Random Priority-Based Thrashing Control for Distributed Shared Memory

Yi-Wei Ci, Michael R. Lyu, Zhan Zhang, De-Cheng Zuo, and Xiao-Zong Yang

Abstract—Shared memory is widely used for inter-process communication. The shared memory abstraction allows computation to be decoupled from communication, which offers benefits, including portability and ease of programming. To enable shared memory access by processes that are on different machines, distributed shared memory (DSM) can be employed. However, DSM systems can suffer from thrashing: while different processes update certain hot data items, the largest amount of effort is spent on data synchronization, and little progress is made by each process. To avoid interference between processes during data updating while providing shared memory at page granularity, more time is reserved for a writer to hold a page in a traditional manner. In this paper, we report on complex thrashing, which can explain why extending the time of holding a page might not be sufficient to control thrashing. To increase the throughput, we propose a thrashing control mechanism that allows each process to update a set of pages during a period of time, where the pages compose a logical area. Because of the isolation of areas, updates on different areas can be performed concurrently. To allow the areas to be fairly well used, each process is assigned with a random priority for thrashing control. The thrashing control mechanism is implemented on a Linux-based DSM system. Performance results show that the execution time of the applications that are apt to cause system thrashing can be significantly reduced by our approach.

Index Terms—Distributed shared memory, inter process communication, thrashing control

1 Introduction

In a distributed computing system, to enable data sharing and coordination between processes, shared memory [1], [2], [3], [4], [5] and message passing [6], [7], [8], [9] techniques are often used. By orchestrating the interactions between processes, message passing can provide a high-performance implementation. In contrast, shared memory hides the details of the interactions between the processes. The update of data items can be automatically propagated to the related processes by the underlying system. Because in distributed shared-memory (DSM) systems [10], [11], [12], [13], [14], [15], [16], [17] a uniform address space can be used by the processes, it is possible for processes to seamlessly access the data distributed across multiple machines, which also provides an easy way to refer to data items that have complex data structures without considering the location of each data item. However, DSM systems can suffer from thrashing [18], [19], [20] that results from data contention among processes on different machines, which not only introduces intensive communication but also causes processes to be frequently interrupted from their computations.

To address this problem, one method is to reduce the granularity of memory unit, which avoids having data items that are stored in a memory unit being visible to the processes that are unlikely to use them; another method is to make processes keep writing to reduce the frequency of synchronization. One challenge in thrashing control lies in determining the time to synchronization. In some consistency models, the synchronization points can be determined according to some specific operations. For example, in the release consistency model [21], [22], the exclusive access of memory is achieved by the use of acquire and release operations, and coherence actions can be performed on acquire or release operations.

Because of the requirement of synchronization, the management of shared memory is not totally transparent to applications. The eventual consistency model [23] provides another way of reducing the frequency of synchronization, which is to assume that the state of memory can be converged after a sufficiently long time. Because the view of memory for each process can diverge over a period of time, it is not straightforward to utilize this model to implement an application in which dependencies among operations are sensitive, such as Lamport's Fast Mutual Exclusion [24]. Sequential consistency remains desirable because it can simplify programming and it makes the reuse of existing shared-memory applications easy. To achieve consistency efficiently without sacrificing ease of programming, the synchronization points can be automatically determined by the shared memory system.

Grappa [32] demonstrates that the performance of the data-intensive applications can be improved through the frameworks built on top a DSM system. By using Grappa,

- *Y. W. Ci is with the Institute of Software, Chinese Academy of Sciences, Beijing 100190, China. E-mail: yiwei@iscas.ac.cn.*
- *M. R. Lyu is with the Department of Computer Sciences and Engineering, The Chinese University of Hong Kong (CUHK), Hong Kong. E-mail: lyu@cse.cuhk.edu.hk.*
- *Z. Zhang, D.-C. Zuo, and X.-Z. Yang are with the School of Computer Science and Technology, Harbin Institute of Technology, Harbin, Heilongjiang 150001, China. E-mail: zz@ftcl.hit.edu.cn, {zuodc, xzyang}@hit.edu.cn.*

Manuscript received 28 Dec. 2018; revised 8 Aug. 2019; accepted 8 Sept. 2019. Date of publication 19 Sept. 2019; date of current version 10 Jan. 2020. (Corresponding author: Y.-W. Ci.) Recommended for acceptance by C. Krintz. Digital Object Identifier no. 10.1109/TPDS.2019.2942302

the complexity of the data management can be hidden by an user-level DSM system. In Grappa, to optimize the low-locality data access, delegation operations [33] are employed to perform the updates of the shared objects at certain locations. As a result, it is not necessary to obtain the latest version of a shared object for tasks. However, to enable the shared memory to be accessed without additional primitives as the conventional memory, it is desired that any object contained in the shared memory can be accessed via a pointer. In addition, the knowledge of the objects contained in the shared memory is not assumed to be known to the underlying system.

Because different processes can keep updating certain data items, it is possible for the processes to make no progress if each process cannot efficiently obtain the data items for a sufficiently long time. A traditional approach [12] is to delay synchronization points to provide each process with more time to perform the updates. In this paper, we explore memory management with respect to variable granularity. We have found that extending the time to update a memory unit for each process is not sufficient for thrashing control because processes can wait before updating the memory units which are no longer updated by their holders. If a circular wait appears (i.e., each of the two processes waits for the page kept by the other process), although the frequency of synchronization can be reduced, page thrashing is not truly under control. In other words, a process can request to update a set of memory units during a period of time, and it is unwilling to be interrupted in the course of updating. We show, through evaluation, that the write performance can be improved for the thrashing-prone applications by controlling the time to access memory units.

In this paper, we made the following contributions on the prevention of thrashing:

1. An approach is present to determine a logical area which is frequently used by a process so that processes can keep updating a set of pages during a period of time, which can be used to reduce page thrashing.
2. The writer of each area is selected based on random priorities, which enable the fair use of memory. Because no control message is required for choosing the writer of an area, the cost of changing the writer can be low.
3. An OS level implementation of the thrashing control mechanism proves that the performance of the distributed shared memory system can be improved through the underlying system.

The remainder of this paper is organized as follows. The related work is described in Section 2. The background is presented in Section 3. The priority-based method for controlling the access of memory is given in Section 4. The thrashing control is elaborated in Section 5. Optimizations are given in Section 6. The implementation of the thrashing control mechanism is given in Section 7. The evaluation of the thrashing control mechanism is given in Section 8. Conclusions are given in the last section.

2 Related Work

There is often a trade-off between the latency and the consistency. A weaker consistency model can be used to improve the system performance. However, the state of memory could diverge temporarily or permanently.

In the eventual consistency model, because updates to memory are eventually visible to the other processes without the ordering restriction, non-commutative operations cannot be supported. RedBlue consistency [25] provides a way to enhance consistency when necessary by mixing strong consistency and eventual consistency models. The non-commutative operations and the operations that are not invariant safe are colored red. When red operations are encountered, a stronger consistency is required to ensure that the red operations can be perceived in the same order. The operations that are commutative and invariant safe are colored blue. Blue operations appear to take effect according to eventual consistency. They can be executed locally without coordination. Thus, a performance gain can be achieved if the blue operations account for a large proportion of the operations. In this consistency model, additional information is required to help the system determine the colors of the operations. The efforts of collecting such information and marking operations are non-trivial. Paper [26] gives an enhancement of a causal consistency model [27] to avoid the permanent divergence of a storage state. In a traditional causal consistency model, although the order of the operations that have causal dependencies is maintained, updates to storage could conflict because the concurrent operations can be executed in any order. To ensure convergent behavior, a conflict resolution rule (e.g., the rule of the last-writer-win) is required to handle the conflict updates. The causal+ consistency is an enhanced version of causal consistency, but it is still weaker than sequential consistency.

In a stronger consistency model, simultaneous updates to the storage can also be ordered. However, the cost of ordering the updates from different processes is often expensive. To address the performance issue, a release consistency model is proposed. In this model, updates from an individual process can be performed locally. They are propagated to other processes only when explicit synchronization points are encountered. Because specific primitives are required to mark the synchronization points, additional programming complexity is required.

In release consistency, the synchronization points show the boundaries of execution, and the boundaries are static. BulkSC [28] provides a way to mark the boundaries in a dynamic manner by hardware support. BulkSC enforces sequential consistency with a coarse grain of chunks, each of which is a set of consecutive instructions that are dynamically grouped. In BulkSC, the synchronization points, which are transparent to applications, are located at the boundaries of the chunks. Because data synchronization can be less intensive and updates can be batched, BulkSC achieves a similar performance gain compared with release consistency. However, BulkSC does not guarantee the advancement of computation. Disambiguation operations are required to avoid the violation of sequential consistency. In these operations, chunks can be re-executed.

To support conventional shared memory applications even if there are no explicit synchronization points, sequential consistency is considered in this paper. Thrashing often occurs when processes all want to hold some data object for

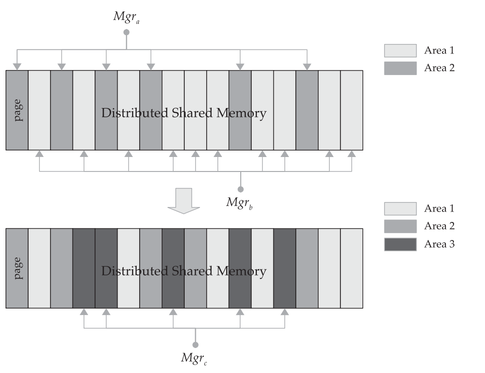


Fig. 1. Memory areas.

updating but no one has the patience to wait. The central problem is that there is a lack of mechanism to guide processes in memory access. A method that uses specific primitives to guide the memory usage provides an explicit way to perform thrashing control. Another method is to determine the writer of the memory and the time for updating implicitly, which is analogous to the work of process scheduling. To achieve thrashing control, the scheduling must improve the overall writing speed of the system. When a process is scheduled to be a writer, sufficient time should be allocated to guarantee a sustained write. A straightforward method is to select each process to be a writer in a round-robin fashion [29], which allows the processes to be scheduled in a circular order. If the time slots are evenly allocated to the processes, fair usage of the memory can be achieved. For determining a writer, all of the requests can be sent to a scheduler first; the scheduler will know who the next writer is. However, the centralized management could lead to a performance bottleneck. To alleviate the load of the scheduler, a distributed method for the memory management is desired. Token passing [30] allows each process the opportunity to be scheduled. Once a process holds the token, it can schedule itself to be a writer. After a period of time, the scheduler must be altered, and the token is passed to the next process. If a process is idling, then it can also reject being scheduled, and it can pass the token to the next process directly. However, with an increase in the rejections, the efficiency of scheduling will be impaired.

In this paper, a scheduling scheme that has a distributed design is proposed to reduce thrashing. The page accessing of the processes can be controlled by a set of managers, each of which manages an area of memory. Each area contains a set of pages that might attract the interest of a group of processes. The area managers provide the capability to suppress the interference between processes that access different areas. If a process has an interest in several areas, then its requests should be processed by the managers of those areas in a coordinated manner. Without such coordination, processes can still suffer from the thrashing effect. To avoid sending control messages for coordination, we propose a random priority-based approach, which provides a fair use of the shared memory.

3 BACKGROUHD

We assume a distributed system with a set of nodes. Each process can be deployed on any of the nodes. Each shared

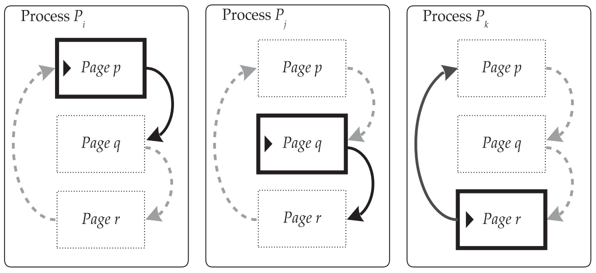


Fig. 2. Complex thrashing.

memory can be divided into N_a logical areas at most. Each area comprises one or more pages and is managed by a manager. Here, the number of managers can be changed with changes of the areas (as shown in Fig. 1). Sequential consistency requires that the order of updates can be perceived by all of the processes, and the ordered updates are extended from the updates that are performed by each process. For each memory unit (e.g., a page), sequential consistency is maintained. Concurrent updates on different memory units are allowed. Thrashing control can perform at page granularity or at the granularity that is greater than a page.

In page-granularity thrashing control, extending the time to keep updating a page can reduce the probability of being interrupted. The time window should not be longer than the time required for updating. However, it is often difficult to know when a process stops writing. Even if the status of each page can be checked periodically and a successor can efficiently be found to take the place of the current writer, thrashing can also appear. This scenario can be explained by the following example (as shown in Fig. 2). Processes P_i and P_j update pages p and q repeatedly. When process P_i updates page p, process P_j updates page q. After a while, P_i shifts to updating page q, and P_j shifts to updating page p. Each of the two processes requires the page that has been updated by the other process. In this example, extending the time for a writer to hold a page cannot reduce the overall communication cost. Let W_i denote the set of memory units that are updated by process P_i in a period of time. Here, the memory unit is not restricted to a page. To represent the dependencies between the worksets required by different processes, the corelation between memory units is defined as follows, denoted by $\leftrightarrow$.

Definition 1. *Suppose that there is a memory unit m_u, such that $m_u \in W_i$. m_u is related to another memory unit m_v (i.e., $m_u \leftrightarrow m_v$), iff the following condition holds:*

$$\exists W_j : \ (m_v \in W_j) \wedge (W_i \cap W_j \neq \emptyset).$$

Let $\leftrightarrow^*$ denote the transitive closure of relation $\leftrightarrow$. The complex thrashing, which appears when processes have the same interest in a set of memory units, can be defined as follows.

Definition 2. *Complex thrashing occurs in processes P_i and $P_j\,(i \neq j)$ iff P_i and P_j access two memory units, m_u and $m_v\,(u \neq v)$, that satisfy the following condition:*

$$(m_u \leftrightarrow^* m_v) \wedge (m_u \in W_i) \wedge (m_v \in W_j).$$

If two or more processes are involved in the interdependence between memory units in a period of time, thrashing can occur.

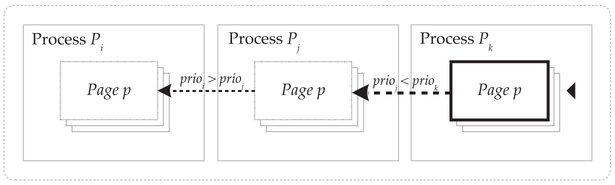


Fig. 3. Priority degradation.

4 Priority-Based Method

To prevent complex thrashing, the memory units that have interdependencies cannot be managed independently. If memory units are grouped for accessing and one process is allowed to access a group for each period of time, more memory units can be updated by the process in a period of time. As there is the chance that each of the two processes waits for some of the grouped pages that are kept by the other process, each process can be assigned a priority to avoid circular wait. If the priority of a process is sufficiently high, the process can hold all the required memory units.

4.1 Priority Degradation

As the processes that access shared memory have the latest copies of the content of memory units, they can be treated as the content holders of the shared memory. The use of priority makes the process that has the higher priority keep updating. However, if each content holder also controls the access of the corresponding memory units, the process with high priority can be blocked because of priority degradation, which appears when the processes fail to preempt at the content holders. The definition of priority degradation is as follows.

Definition 3. *The priority of a process P_i is degraded iff $\exists P_j : P_i$ sends a request to update the content that is provided by P_j and P_j satisfies the following:*

1) *$\exists P_k$: The priority of P_j is lower than that of P_k.*
2) *The priority of P_i is higher than that of P_j, and P_j is blocked since it requests to update the content that is held by P_k.*

An example (as shown in Fig. 3) is given as an illustration. In this example, each page holder can determine whether a process is allowed to update the corresponding page in terms of the priority of the process. If the priority of a process is higher than that of the current writer of a page, then the process can update the page. Suppose that the current priorities of processes P_i, P_j, P_k are $prio_i, prio_j$, and $prio_k$, respectively $(prio_i > prio_j, prio_j < prio_k)$. Processes P_i and P_j both request to update page p, which is held by process P_k. P_j tries to become a new holder of page p. Because $prio_j < prio_k, P_j$ cannot finish retriving page p. When P_i sends request to the new holder for updating page p, because P_j is retriving page p, even if $prio_i > prio_j$, P_i must wait also. There can be a chain of possible content holders that request updating the same memory unit. Preemption fails to be performed when the dependent requests are processed independently. Although one process's priority is higher than that of the requested process, if processes request to update the same page, there is the chance that the process with higher priority will require considerable time to retrieve the required page.

4.2 Preemption

If there is a manager to control the access of memory units, the write requests from the processes that have lower priorities can be delayed by the manager, which can be used to avoid priority degradation. In addition, if a process is allowed to keep updating a set of memory units without interruption for a period of time, then the interdependencies among memory units can be hidden from the other processes during that period. As a result, thrashing can be inhibited. For example, as shown in Fig. 2, if pages p, q and r can only be updated by a process whose priority is sufficiently high, the processes with low priorities cannot block a writer from updating.

To prevent a bottleneck of the manager, the memory units can be grouped into a number of areas, each of which has a corresponding manager. These memory units can be dynamically allocated to an area according to the memory access pattern. For example, if unrelated data items are allocated to different areas, then the concurrency of the applications can be improved because updates on different areas can be performed concurrently.

To avoid direct coordination when determining a writer, a priority assignment method is adopted. Suppose that the priority of each process is updated after each T_{up} time interval. Let $prio_i(t)$ be process P_i's priority as perceived by an area manager at time t, where t is the local time of the area manager. The area manager can detect whether a priority expires according to the live-time that is assigned by the writer. Let $curr(t)$ be the highest unexpired priority that is perceived by an area manager when the local time of the manager reaches t. If all of the perceived priorities expire, then $curr(t)$ is set to 0. The access control of shared memory can be modeled as follows.

Rule 1. Process P_i is allowed to update an area when the local time of the area manager reaches t if the area manager perceives that $prio_i(t)$ is higher than or equal to $curr(t)$.

Because the process with high priority can keep updating, the unnecessary interleaving of updates from different processes can be reduced. Note that if a memory unit is marked as read only, any process can read the unit directly without blocking. When processes try to update an area, the manager of the area must select a process to be a writer. Given that a process can access many areas over a period of time, the managers of these areas should provide the same permission of memory access to the process. Otherwise, an area that is opened to a process could turn to starvation status because the requests of that process can be blocked by the managers of the other areas. To ensure fair usage of the memory, the priority of each process can be updated in a round-robin fashion.

Let T_{up} denote the period of the priority update, and $I(t)$ the priority update interval when time reaches t, where $I(t) = \lfloor t \,/\, T_{up} \rfloor$. Suppose that there are at most N processes for accessing memory. The priority of P_i can be determined as follows:

$$prio_i(t) = (i + I(t)) \, mod \, N.$$

Table 1 shows an example for the priority assignment. Suppose that processes P_1 and P_2 both try to update a page during time 0 to $4T_{up}$. Because there is only one interval in which the priority of P_1 is higher than that of P_2, P_1 can

TABLE 1
Priorities of Processes $(N = 4)$

Time	Process P_1	Process P_2	Process P_3	Process P_4
$0 - T_{up}$	0	1	2	3
$T_{up} - 2T_{up}$	1	2	3	0
$2T_{up} - 3T_{up}$	2	3	0	1
$3T_{up} - 4T_{up}$	3	0	1	2

TABLE 2
Priorities of Processes $(N = 4)$

Time	Process P_1	Process P_2	Process P_3	Process P_4
$0 - T_{up}$	0	3	1	2
$T_{up} - 2T_{up}$	2	1	3	0
$2T_{up} - 3T_{up}$	3	0	2	1
$3T_{up} - 4T_{up}$	1	2	0	3

spend at most T_{up} time for updating that page. In contrast, P_2 can consume at most $3T_{up}$ time for its updating. Clearly, the method does not provide sufficient fairness for appropriate memory access. To improve the fairness, a random priority-based approach can be adopted, which will be introduced in the next section.

4.3 Random Priority

To provide the fair use of memory, the system must ensure that for any two processes P_i and P_j, they have the same chances to access an area. In other words, the possibility that P_i has a higher priority compared to P_j should be equal to the possibility that P_j has a higher priority compared to P_i. It is not desirable to assign the same priority to different processes because this strategy could allow more than one process to update an area.

Assigning different priorities to processes guarantees that a process is allowed to update an area while the processes with the higher priorities remain idle. Suppose that N priorities can be assigned to processes in each round. To ensure fairness, each process must have the same opportunity to be assigned with any priority. The priority of each process can be generated in a random manner. The priority of a process P_i can be generated as follows:

$$prio_i(t) = rand(i, I(t)),$$

where *rand* is a random function. The random function can have different forms. Here, the *rand* function provides a mapping of a random matrix *Rand*. *Rand* can be used to generate priorities for each priority update interval. Suppose that there are N rows of $Rand. rand(i,j) = Rand_{i,j \bmod N}$. Each row of $Rand_{i,j}\ (j = 0, 1, \ldots, N-1)$ can be constructed as follows:

1) Initially, there are N different priorities to be selected for process P_i, and each item of $Rand_{i,j}$ is set as uninitialized.
2) For each time, a priority is randomly selected from the remaining priorities of process P_i, and an uninitialized item of $Rand_{i,j}$ is set to the selected priority.
3) After N times, the setting is finished, and each item of $Rand_{i,j}$ is initialized with a priority for process P_i.

The random function generated according to this method guarantees the fairnes priorities, which are proved in Appendix A. An example of generating priorities is given in Table 2. It can be seen that every process has the same opportunity to have the highest priority during an interval of $4T_{up}$. Processes P_1 and P_2 have the same opportunity to preempt each other.

5 Thrasing Control

A priority collision occurs when a priority is assigned to more than one process at the same time. A requirement for the priority assignment is to ensure that priority collisions are controllable. In the priority assignment, because it is difficult to synchronize the local timing of the processes, processes might become aware of the new priority of a process at different time points. If a priority is utilized by different processes at the same time, then thrashing could occur. To generate the priorities of the processes, an approach, called random priority-based thrashing control (RPTC), is utilized.

In RPTC, the priority of each process is updated periodically, and the processes have the same opportunity to use any priority. When a request arrives at an area manager, the manager checks the priority of the requester for scheduling. The priority can either be determined on the requester side or on the manager side. In the priority-based approach, a process with a high priority has the permission to access the memory during a period of time. To avoid direct coordination, the priority of each process is converted according to time. Determining the priority of each process is not restricted on the manager's side. However, it requires that the managers can be aware of the change of priorities in an effective way when the variations in the local time and the transmission delays are taken into account.

To avoid the occurrence of priority degradation, the requests for updating memory can be sent to the corresponding managers first. The priority of a process can be generated before sending a request. The generated priority is piggybacked on the request. In each priority update interval, the manager can select requests that are piggybacked with higher priorities to be processed first. Let req_i be the request that is sent from processes P_i. Let $t(req_i)$ denote the local time of process P_i when request req_i is sent. Suppose that requests req_i and req_j are received by a manager during the same priority update interval. If requests req_i and req_j are sent from different priority update intervals (i.e., $I(t(req_i)) \neq I(t(req_j))$), there could be a collision between the priorities that are piggybacked on req_i and req_j. Priority collisions bring a risk of thrashing.

To reduce the above risk, one way is to extend the range of the priorities that can be selected; namely, the maximum priority can be far greater than the number of requesters. If the range is sufficiently large, the possibility of priority collisions can be small. However, it is possible that a process is blocked by itself because of the uncertain offset in the local time between processes. For example, as shown in Fig. 4, if requests are sent by a process in different priority update intervals but are received by a manager in the same priority update interval, then the process can be blocked by itself. This problem can be addressed by notifying the manager of the live time of each priority. Managers determine whether a priority expires according to the livetime that is specified by the requester. Let $req_i.live$ denote

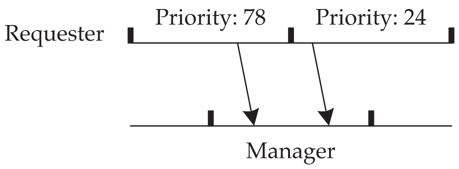


Fig. 4. Blocking effect.

the live time that is piggybacked on request req_i. $req_i.live$ is the duration of the current priority of P_i. Here, $req_i.live$ can be expressed as follows:

$$req_i.live = (I(t(req_i)) + 1)T_{up} - t(req_i).$$

Suppose req_i is a request that hits a memory area. If the manager of the memory area finds that the priority of the process P_i is higher than the priority of the process accessing the memory area, P_i is allowed to be a writer. Here, $req_i.live$ is the time that is reserved for writing. If req_i is not processed because the priority of P_i is not sufficiently high, it will be rescheduled after a period of time.

Although live time can be used to avoid interference from the same process, it is still difficult to avoid interference from different processes. The requests from the processes with higher priorities can arrive at any time, which could prevent a process from updating. We show this in Fig. 5, where there are two processes, P_i and P_j, that both try to update an area. First, the request of process P_i arrives at the manager of the area. The manager reserves time for P_i. During this period, the perceived priority is 27. When the request of process P_j has arrived, because the priority of process P_j is higher, P_j is allowed to update the area. The reserved time is recalculated according to the live time piggybacked on the request of P_j, and the perceived priority of the current interval is updated to 138. Process P_j also fails to use the whole reserved time because a new request from P_i is received by the manager and the perceived priority of the current interval is changed to 156. To guarantee the writing time for each process, the deviations of local time and transmission delays should be controllable. Let T_{off} be the sum of the maximum transmission delay and the maximum deviation of local time. Assuming the execution delay can be ignored, if $2T_{off} < T_{up}$, then there exists $T_{up} - 2T_{off}$ time to safely use a priority.

To guarantee the updating of each process, the live time of the priorities can also be determined on the manager side. When a process is assigned with the highest priority, it can update memory without any interruption. Because the expected interval for being assigned the highest priority is NT_{up}, it is possible for each process to safely update memory in a timely manner. Suppose that when the manager of a memory area receives a request req_i, the local time of the manager is t. Here, $prio_i(t)$ is the priority generated by the

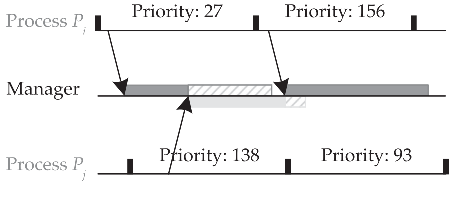


Fig. 5. Access interference.

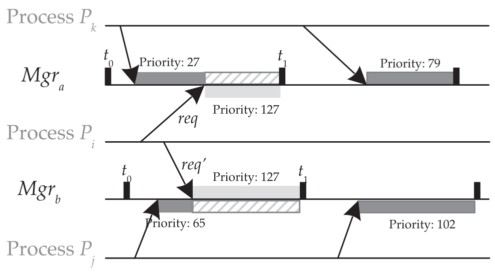


Fig. 6. Cross-area access.

manager. If the manager perceives that the priority of P_i is higher than that of any other requester in the interval $(I(t)T_{up}, (I(t) + 1)T_{up})$, then it allows P_i to update the memory area. Otherwise, request req_i will be rescheduled until the priority of process P_i is sufficiently high.

To provide a sustained write for a cross-area update, managers should make the same decisions for the priority assignment, which can be achieved by making the local time of the managers synchronized. As shown in Fig. 6, for example, managers Mgr_a and Mgr_b determine the writing time of processes independently. Because requests *req* and *req*′ are received during the same time interval (t_0, t_1) and process P_i has a higher priority compared with processes P_j and P_k in that interval, the two requests can be processed by both managers. While there is a considerable deviation between the local time of the managers, the requests sent from the same process to different managers still have a chance to be concurrently processed if the managers perceive that the priority of the process is higher compared with the other requesters.

6 Optimizations

6.1 Priority Reusing

In a priority update period T_{up}, the process that has the higher priority can be accepted to access an area. To make each area accessed in a fair manner, the priority of each process is updated after each T_{up} time. With the increase in T_{up}, because the time required by a process to access an area is uncertain, the possibility that an area is not used by any process could increase. In contrast, if T_{up} is too short, then there could be insufficient time to retrieve the required pages for a process, which impairs the writing performance.

A method that allows the writer to be determined according to the memory access pattern is adopted. In this method, T_{up} remains short. To provide a writer with more time when a sustained write is required, the priority of each process is updated to a different value after αT_{up} time. α is called priority duplication rate. This approach is different from extending T_{up} to α times because if the process with the higher priority turns from busy to idle during αT_{up} time, then the process with the lower priority can take its place. As an illustration in Fig. 7, the priorities of processes can stay for a time interval $3T_{up}$. From time t_1 to t_2, because process P_i does not update the memory, process P_j, which has a lower priority, is allowed to be a writer. After time t_2, process P_i tries to update memory again; then, P_i has the chance to be a writer because its high priority has been retained.

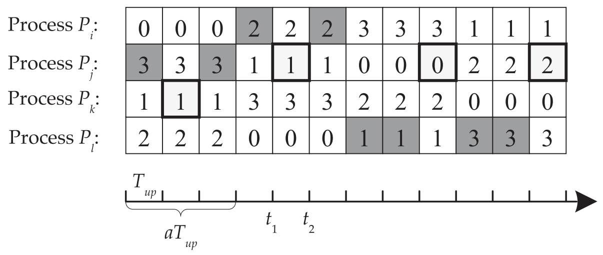

Process P_i: 0 0 0 2 2 2 3 3 3 1 1 1
Process P_j: 3 3 3 1 1 1 0 0 0 2 2 2
Process P_k: 1 1 1 3 3 3 2 2 2 0 0 0
Process P_l: 2 2 2 0 0 0 1 1 1 3 3 3
T_{up}
aT_{up}
t_1
t_2

Fig. 7. Priority duplication ($\alpha = 3$).

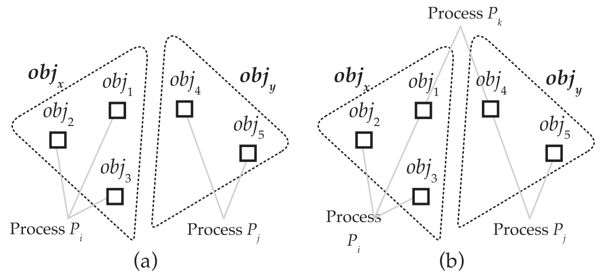


Fig. 8. Object finding.

6.2 Detecting Hot Areas

Given that the data items allocated together could be of interest to a different group of processes, each allocated memory is allowed to be divided into areas. Each area can be treated as an integrated data object. The minimum size of an area is one page, which does not ensure that one area is used to describe only one data item. The data items contained in an area can be seen as members of the area. Because the shared-memory system does not differentiate the data items inside the allocated memory, the areas cannot be obtained based on certain boundaries of the data items. In RPTC, the areas are detected in terms of the assumption that the requested content of the data objects eventually composes the requested data objects. To simplify the detection of the data objects, combining the data objects is allowed so that a group of objects can be detected together. Let N_v denote a visiting threshold. The rule for object combination is given as follows:

Rule 2. Objects obj_i and obj_j $(i \neq j)$ can be combined to a new object if obj_i and obj_j are accessed by a process for at least N_v times $(N_v > 0)$.

For example, process P_i has an interest in objects obj_1, obj_2, and obj_3 (as shown in Fig. 8a), while process P_j has an interest in objects obj_4 and obj_5. As a result, the five objects can be combined into two objects obj_x and obj_y, each of which represents an area. Because there is no intersection of the two areas, processes P_i and P_j can access the areas without interference. If another process P_k joins the system (as shown in Fig. 8b) and it has an interest in the objects obj_x and obj_y, the requests of P_k must be permitted by the managers of both areas. The dependency between objects is allowed in the thrashing control system. Even if a data item is divided into parts that belong to different areas, the high priority process has the chance to access these areas during a period of time.

Dividing the memory into areas is utilized not only to reduce the interference between the processes in the course of updating the memory but also to offset the load of the single manager. The method for determining areas is to apply the access history of the processes. Initially, all of the pages are managed by a manager Mgr_0. When a remote requester tries to access a page, if Mgr_0 finds that the page is still managed locally, Mgr_0 allows the requester to manage the page. Eventually, the pages that are managed by the same requester form an area. One issue is that there could be some memory that is initialized by a process but is not used further. If the manager of a page is determined once a process accesses the page, then an area might not describe the data objects of interest, which could introduce more cross-area memory access. The cross-area memory access requires managers to make decisions on priority assignments in a coordinated manner.

To reduce cross-area memory access, there is a strategy that a requester is allowed to be a manager of a page if it has requested to access the page for at least N_v times. Note that a large N_v can slow down the construction of areas. Another issue is that if there are too many areas to manage, although the load of each manager can be alleviated, it is difficult for the managers to track the use of the memory. In the priority update method mentioned above, if a manager of an area fails to detect the memory access behavior of the process with the higher priority, it allows the process with the lower priority to access the area, which is undesired by the busy process with the high priority. To address this problem, the number of areas should be controlled. In this paper, the maximum number of areas is set to N_a.

7 Implementation

The prototype is implemented as a component in a DSM system. Here, we consider variable granularity thrashing control. Basically, there are three roles in RPTC, namely the requester, the holder, and the manager. The requesters in each node issue page fault requests to the corresponding area managers. The manager determines whether a request can be processed according to the priority of the requester. If the manager finds that the priority of the requester is low, it will block the request of the requester until the priority of the process accessing the area expires. When a request can be processed, the manager sends the request to the holders that maintain the content of the page. When all of the required parts of a page are collected, the requester delivers the page to the operating system.

A Linux-based distributed shared memory (LBDSM) system is implemented to provide an operating system level implementation. The memory management can be transparent to the applications. LBDSM is compatible with System V IPC, which provides the system calls to create and release the shared memory. There are three layers in LBDSM, namely, the translation layer, the link layer, and the resource layer (as shown in Fig. 9).

The resource layer provides an abstraction of distributed resources, such as the memory, massage, and semaphore. The positions of the distributed resources are transparent to the applications. The link layer is a bridge between the translation layer and the resource layer. The translation layer translates the requests of the kernel. Next, the link layer passes the translated requests to the resource layer. When the results from the resource layer are obtained, the link layer passes the results to the translation layer. The

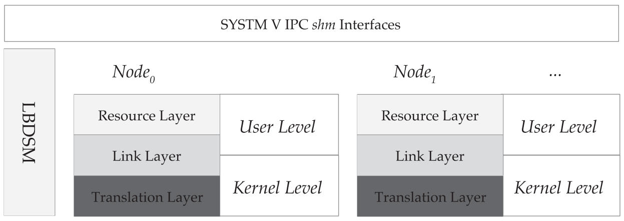


Fig. 9. LBDSM.

translation layer translates these results to the forms in which the kernel can be utilized. For the memory resource, its consistency is maintained at the resource layer.

A sequential consistency model is applied in LBDSM, which allows the shared memory to be employed without synchronization primitives. The underlying system arranges data synchronization. To prevent shared memory from being used in a greedy manner, the thrashing control mechanism is provided. The random priority-based thrashing control is implemented as a part of the resource management in the resource layer.

8 Evaluation

In this section, the performance of RPTC is explored. Some applications that are prone to inducing thrashing are considered to test the effectiveness of the thrashing control. The SPLASH-2 benchmark suite [31] is also used for evaluation. In the evaluation, 17 VMs are used; 16 VMs provide computational and memory resources and one VM is used as the metadata server. The metadata of resources, such as the locations of shared-memory resources, is maintained by this metadata server. All of the VMs run in the VMware ESXi 6.0 environment. Each VM has 1GB of RAM and 4 vCPUs. 10-Gb NICs are used to transfer data between VMs. The virtual machines are deployed on two servers. Each of the servers has one i7-4790 Intel processor and 16 GB of RAM. Let T denote the time required to complete an experiment.

8.1 Thrashing-Prone Applications

If processes have the same interest in certain data items, and each of the processes wants the opportunity to update them, the data items can be frequently transferred between the processes, which results in thrashing. It can be observed that if there are data items that are commonly accessed by the processes, it is likely to induce thrashing. To evaluate the performance of the system when data contention is high, the following applications are employed. Let $size_{mem}$ be the size of distributed shared memory, let $size_{pg}$ be the size of each page, and let $Total_{pg}$ be the number of pages of the shared memory.

8.1.1 Hotspot

An application called Hotspot is developed to simulate the behaviors that occur when processes focus on the same parts of memory. In this application, different sizes of memory will be allocated for testing. In each round of testing for each process, k different areas of the shared memory can be updated, where $k = log_2(Total_{pg})$. Let $Area_n$ be the n-th area that is selected to update in a round of testing, and $start_n$ the start of $Area_n$ $(start_0 = 0)$. The size of $Area_n$, denoted by $Area_n.size$, is $2^{k-n} \cdot size_{pg}$. The start of $Area_n$ $(n = 1, 2, \ldots, k)$ can be expressed as $start_n = start_{n-1} + c_n \cdot Area_n.size$, where c_n is a Boolean value which is randomly assigned in each round of testing.

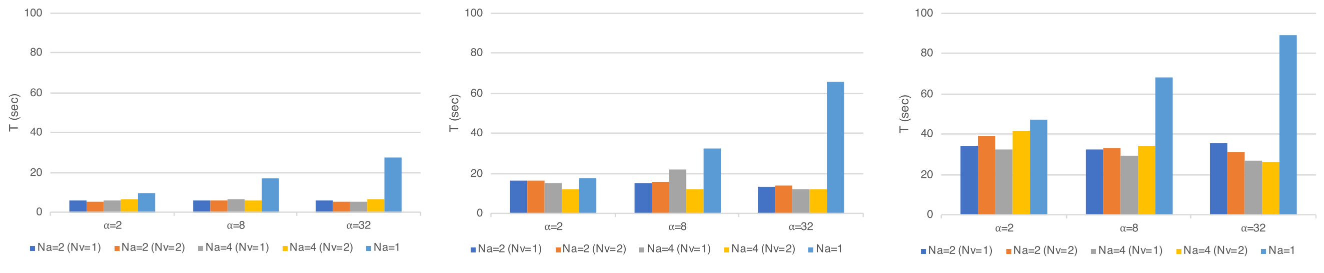
Each area $Area_n$ is utilized as a circular buffer. For each time, a process updates r bytes of the buffer describing $Area_n$, where r is randomly selected and $r < Area_n.size$. The processes update the buffers in a concurrent manner. While there are overlaps of the areas that are being updated by the processes, thrashing can occur. In the experiment, each process performs 10000 rounds of tests, and $size_{mem}$ is 16KB. If the size of the shared memory is larger, there is less chance that the processes access the same areas of the memory.

In each round of testing, a process repeatedly divides the area that it focuses on and randomly selects half of the area to update. If the target area of a process is also accessed by the other processes, the holders of the pages belonging to the area can be altered. However, the process has a chance of becoming the holder of these pages again in the course of rewriting, which can be used to simulate the effect of thrashing. With an increase in the number of processes that participate in updating the shared memory, the probability that two processes update the same area and the probability that the area updated by one process is a part of the area that is accessed by another process can increase. Namely, thrashing is more likely to take place when more processes participate.

In Hotspot, each process has a random manner for memory access. A process can update more than one logical area determined by RPTC. Although the throughput of memory accesses can be improved when using a larger logical area number N_a, there is a risk that the concurrency of updating shared memory is reduced resulting from interdependencies among areas. If the logical areas of shared memory can be detected (as shown in Fig. 10), better performance can be achieved. The visiting threshold N_v determines the speed that logical areas of shared memory can be formed. When a smaller N_v is chosen, the logical areas of shared memory can be formed faster. A high workload that is placed on the initial manager can be efficiently balanced if a small value of visiting threashold N_v is used. For $N_a = 1$, all the memory operations are performed in the same logical area. Namely, though there exist independent operations updating different memory areas, the operations must be interleaved, resulting in performance degradation.

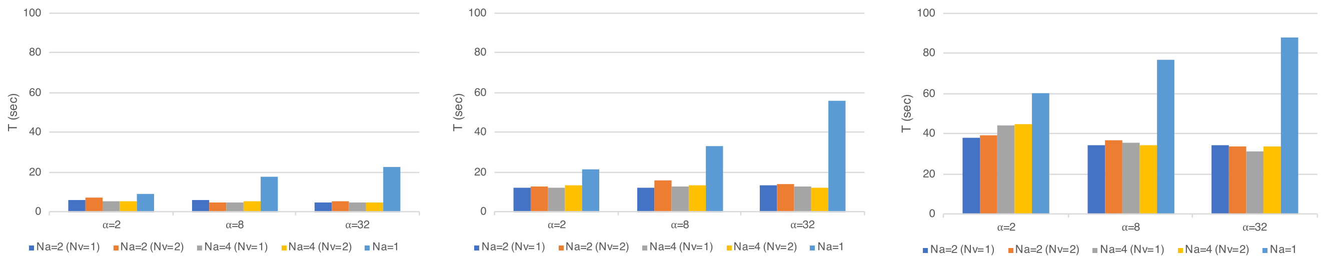
8.1.2 Shared Memory Mutual Exclusion

Mutual exclusion is an important primitive for controlling access to shared resources. In a distributed system, mutual exclusion can be implemented by either shared memory or message passing. Supporting the shared memory version of mutual exclusion makes it possible for the synchronization and update operations to be performed together to test the same shared memory system. Because the memory that is protected by the mutual exclusion operations can be updated by only one process, there is no contention for this part of the memory. In contrast, when executing mutual exclusion operations, the required states, which are stored in distributed shared memory, can be changed by the processes at any time.



(a) 4 Nodes (b) 8 Nodes (c) 16 Nodes

Fig. 10. Hotspot ($size_{mem} = 16KB$).



(a) 4 Nodes (b) 8 Nodes (c) 16 Nodes

Fig. 11. ME $(N_{pg} = 4)$.

An application called ME is developed to simulate the use of shared memory for mutual exclusion and random update. Three operations, *lock*, *update*, and *unlock*, are executed successively in each round of testing. The *lock* and *unlock* operations are implemented based on Lamport's Fast Mutual Exclusion Algorithm [24] for the mutually exclusive memory access. The operation *update* is used to randomly update shared memory. A page of the shared memory is reserved for storing the data that is required by the *lock* and *unlock* operations. This page can be hot because the processes can frequently read and write this part to acquire and release a lock. The other pages can be used by the *update* operation. In the *update* operation, n pages are randomly selected from N_{pg} pages for updating.

In ME, it is desireded that the reserved page of the shared memory can be updated without interruption. In Fig. 11, if memory can be divided into logical areas, better performance can be achieved. When more logical areas are enabled, even if a process does not have a high priority, the process has a chance of accessing the areas that are not accessed by the processes with higher priorities. A larger value of α can help the area mangers reserve more time for a process that requests access to the memory.

8.2 SPLASH-2 Benchmarks

The Stanford Parallel Application for Shared Memory (SPLASH) is a benchmark suite for evaluating shared-memory; it comprises four computational kernels and some applications. To make SPLASH2 support System V IPC, we implemented a set of PARMACS macros which are used by SPLASH2. In this implementation, not all benchmark applications can be supported. Currently, the most of SPLASH kernels can be performed, and they are used for evaluation. In this section, we focus on the performance of the following SPLASH kernels.

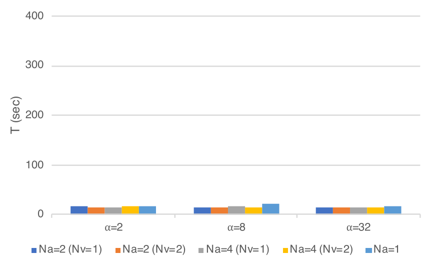
Radix. This kernel implements radix sort, which is a non-comparative integer sorting algorithm. The algorithm is performed iteratively. For each time, keys are sorted according to their ranks, which are determined according to r bits in each key. All-to-all inter-process communication will be encountered when permuting keys to new positions. The number of keys is 262114. The maximum key value is 524288. The radix size is 1024.

FFT. This kernel implements the FFT algorithm in which the matrix transpose steps require all-to-all inter-process communication. This implementation is optimized to minimize the inter-process communication. The data set consists of $1k$ data points to be transformed.

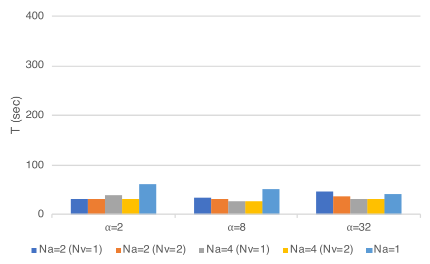
LU. This kernel factorizes a dense matrix as the product of a lower triangular matrix and an upper triangular matrix. The dense matrix is divided into blocks, which can be allocated to different processes to balance the workload. A typical size of each block is 16×16. The size of the matrix is 128x128.

For the Radix kernel, shared memory content can be frequently exchanged between processes. If a process requires the pages distributed around logical areas of the shared memory to be updated, it is better for the process to have more time to access the areas, which can be achieved by increasing priority duplication rate α. Otherwise, there is no enough time for a process to update pages belonging to different areas, and the process can repeatedly request pages that have been written by the other processes. In Fig. 12, if more logical areas are allowed, the system can achieve better performance. For the FFT and LU kernels (as shown in Figs. 13 and 14), shared memory content is not frequently exchanged between processes. When there is light workload for the management of memory access, the shared memory is not necessary to be divided into distinct areas. In this case, the workload imbalance of the area managers is not likely to have significant impact on the performance.

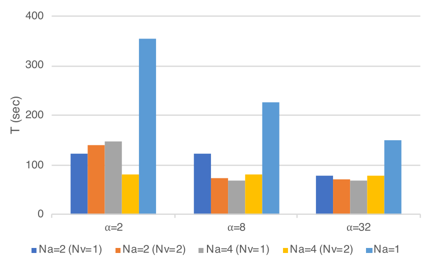
Table 3 shows comparisons of the performance of the shared memory management with thrashing control at the variable granularity (VG) and the fixed granularity (FG).



(a) 4 Nodes

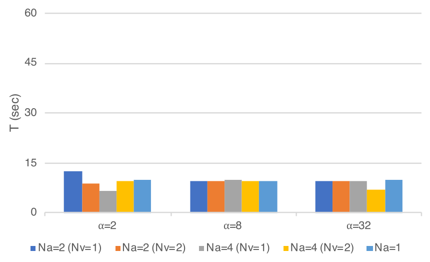


(b) 8 Nodes

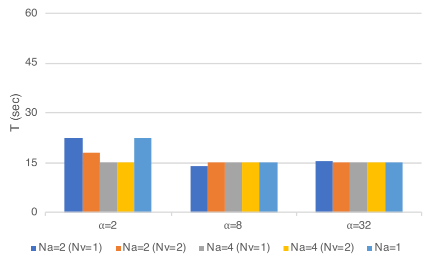


(c) 16 Nodes

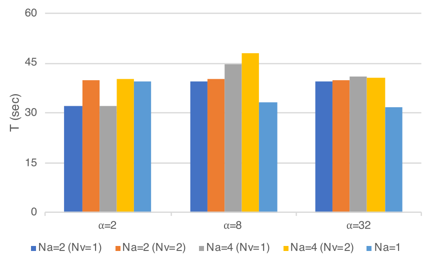
Fig. 12. Radix.



(a) 4 Nodes

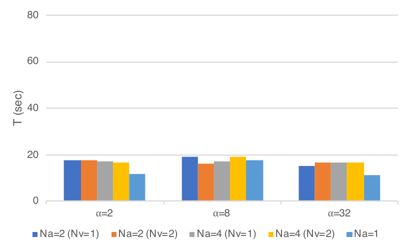


(b) 8 Nodes

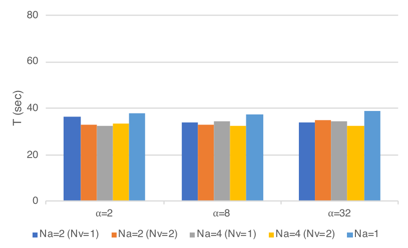


(c) 16 Nodes

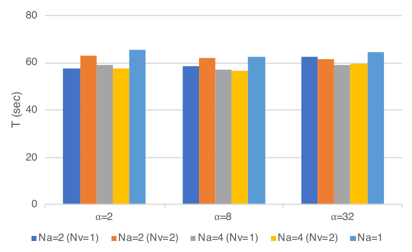
Fig. 13. FFT.



(a) 4 Nodes



(b) 8 Nodes



(c) 16 Nodes

Fig. 14. LU.

TABLE 3
Experiment Time (sec)

Thrashing Control	Hotspot (16 Nodes)	ME (16 Nodes)	Radix (16 Nodes)	FFT (16 Nodes)	LU (16 Nodes)
VG	32.59	44.21	146.31	31.97	58.95
FG	38.58	81.05	336.78	47.22	103.12
VG-noprio	840.62	600.77	215.86	59.78	68.88
FG-noprio	915.00	572.55	474.66	54.16	82.74

For the variable granularity thrashing control, the size of each memory area is not determined, and the RPTC method is utilized. Suppose that $Na = 4, Nv = 1$, and $\alpha = 2$. For the fixed granularity thrashing control, the priority of each process can be maintained in the same way as the RPTC. The difference is that any writer of a page can become the owner of the page. The priority of each accessor is recognized at the page granularity by the owner of the relative page. The location of the owner is tracked in terms of the dynamic distributed manager algorithm [4]. The thrashing-prone applications Hotspot and ME and SPLASH2 benchmark kernels are used. For the Hotspot, $size_{mem}$ is 16 KB. For the ME, $N_{pg} = 4$. The performance results show that for SPLASH kernels FFT and LU, as shared data dose not intensively accessed, thrashing control does not take effect. In addition, there exists additional cost when using priority-based thrashing control at the fixed granularity. In the evaluation of thrashing-prone applications Hotspot and ME and SPLASH kernel Radix, a performance gain can be achieved. Compared with the fixed granularity thrashing control, there is up to 2x performance improvement by using the variable granularity thrashing control. In addition, the experiment results show that after enabling the priority-based thrashing control, the performance of the thrashing-prone programs can be significantly improved.

9 Conclusions

To improve the performance of shared memory systems, synchronization primitives are often employed to guide synchronization, at the cost of sacrificing memory management transparency. Without explicit arrangement of synchronization points, the shared memory system may suffer from thrashing due to data contention. To avoid thrashing in a shared memory system in which synchroni-zation points are transparent to applications, a random priority-based approach, which allows the fair use of memory without sending any control messages to determine the processes that can access memory areas, is proposed. We report that there can be priority degradation because of the exposure of the interdependencies among memory units. To address this problem, the shared memory is divided into logical areas and sustained writes in each area are allowed. Because areas can be formed according to memory access patterns, when unrelated data are allocated to different areas, the data can be visible to the processes that have interest in them and the interference among the processes can be reduced.

ACKNOWLEDGMENTS

We thank colleague Law Hon Man for providing access to computing resources. The work described in this paper was supported by the Hong Kong Scholars Program, the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14210717 of the General Research Fund), and Microsoft Research Asia (2018 Microsoft Research Asia Collaborative Research Award).

APPENDIX A
FAIRNESS

Suppose that the random function *rand* generates N different priorities for N processes in each round. The function is constructed by the method presented in Section 4.3. Because the processes with higher priorities can be idle in a given period of time, it is reasonable to allow the processes with lower priorities to access the shared memory during that period. Fairness of memory access does not imply that each process has the same time available for memory access. Fairness means that for any two processes the probability of preempting one another is the same.

Lemma 1. *For each process, the probability of having any given priority is* $1/N$.

Proof. Suppose that process P_i is the k-th $(1 \leq k \leq N)$ process that is assigned with priority v in a round. Let P be the probability of assigning priority v. If $k = 1, P = 1/N$. Otherwise, if $k > 1$, P can be written as follows:

$$P = \frac{1}{N-k+1} \prod_{n=1}^{k-1} \frac{N-n}{N-n+1} = \frac{1}{N} \quad (k > 1).$$

Thus, the probability of having any priority is $1/N$. □

Theorem 1. *For any two processes, the probability that one process has higher priority than the other process is 1/2.*

Proof. For any two processes P_i and P_j, without loss of generality, suppose that the priority of P_i is generated in advance. By Lemma 1, the probability of selecting priority v for P_i is $1/N$. Suppose that v is the n-th lowest priority. Let x be the priority that is selected from the remaining $N-1$ priorities for process P_j. The probability $P(x > v \,|\, v)$ is $(N-n)\,/\,(N-1)$. The probability that P_i has a higher priority than P_j can be written as follows:

$$P(x > v) = \frac{1}{N} \sum_{n=1}^{N-1} \frac{N-n}{N-1}.$$

Therefore, the probability that one process has higher priority than the other process is $P(x > v) = 1\,/\,2$. □

If the highest priority is assigned to a process, then the process can access memory without preemption for a given period of time. The system must guarantee that each process is assigned the highest priority at least once after a finite expected number of rounds.

Theorem 2. *For each round, if the probability of the process being assigned a given priority is* $1/N$, *the expected number of rounds for which it has that priority is* N.

Proof. The probability of being assigned a given priority after *n* rounds is $(N-1)^{n-1}/N^{\text{n}}$. The expected number of rounds is as follows:

$$E(n) = \sum_{n} \frac{n(N-1)^{n-1}}{N^n}.$$

Let $f(x) = \Sigma x^n$. As $N \cdot E(n) = f'(x)|_{x=(N-1)/N}$. For $x \in [0, 1)$, $f'(x) = 1/(1-x)^2$. Therefore, $E(n) = N$. □

REFERENCES

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial," *Comput*, vol. 29, no. 12, pp. 66–76, 1996.

[2] G. Graunke and S. Thakkar, "Synchronization algorithms for shared-memory multiprocessors," *Comput.*, vol. 23, no. 6, pp. 60–69, 1990.

[3] R. M. Karp, "A survey of parallel algorithms for shared-memory machines," *Technical Report*, University of California at Berkeley Berkeley, CA, USA, 1988.

[4] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Syst.*, vol. 7, no. 4, pp. 321–359, 1989.

[5] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Proc. 17th Annu. Int. Symp. Comput. Archit.*, 1990, pp. 15–26.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Comput.*, vol. 22, no. 6, pp. 789–828, 1996.

[7] G. Burns, R. Daoud, and J. Vaigl, "LAM: An open cluster environment for MPI," in *Proc. Supercomputing Symp.*, 1994, pp. 379–386.

[8] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proc. Eur. Parallel Virtual Mach./Message Passing Interface Users' Group Meeting*, 2004, pp. 353–377.

[9] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," in *Proc. 11th Australian Comput. Sci. Conf.*, 1988, pp. 56–66.

[10] U. Ramachandran, M. Yousef, and A. Khalidi, "An implementation of distributed shared memory," *Softw.: Practice Experience*, vol. 21, no. 5, pp. 443–464, 1991.

[11] K. Li., "Ivy: A shared virtual memory system for parallel computing," in *Proc. Int. Conf. Parallel Process.*, 1988, pp. 94–101.

[12] B. Fleisch, G. Popek, "Mirage: A coherent distributed shared memory design," in *Proc. 12th ACM Symp. Operating Syst. Principles*, 1989, pp. 211–223.

[13] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed shared memory based on type-specific memory coherence," in *Proc. 2nd ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 1990, pp. 168–176.

[14] R. Bisiani, M. Ravishankar, "PLUS: A distributed shared-memory system," in *Proc. 17th Annu. Int. Symp. Comput. Archit.*, 1990, pp. 115–124.

[15] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Techniques for reducing consistency-related communication in distributed shared-memory systems," *ACM Trans. Comput. Syst.*, vol. 13, no. 3, pp. 205–243, 1995.

[16] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "Treadmarks: Distributed shared memory on standard workstations and operating systems," in *Proc. USENIX Winter Tech. Conf.*, 1994, Art. no. 10.

[17] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: Shared memory computing on networks of workstations," *Comput.*, vol. 29, no. 2, pp. 18–28, 1996.

[18] B. Nitzberg, V. Lo, "Distributed shared memory: A survey of issues and algorithms," *Comput.*, vol. 24, no. 8, pp. 52–60, 1991.

[19] J. Protic, M. Tomasevic, and V. Milutinovic, "Distributed shared memory: Concepts and systems," *IEEE Parallel Distrib. Technol.: Syst. Appl.*, vol. 4, no. 2, pp. 63–71, Summer 1996.

[20] V. W. Freeh and G. R. Andrews, "Dynamically controlling false sharing in distributed shared memory," in *Proc. 5th IEEE Int. Symp. High Perform. Distrib. Comput.*, 1996, pp. 403–411.

[21] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," in *Proc. 19th Annu. Int. Symp. Comput. Archit.*, 1992, pp. 13–21.

[22] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Proc. 17th Annu. Int. Symp. Comput. Archit.*, 1990, pp. 15–26.

[23] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[24] L. Lamport, "A fast mutual exclusion algorithm," *ACM Trans. Comput. Syst.*, vol. 5, no. 1, pp. 1–11, 1987.

[25] D. Porto, A. Clement, J. Gehrke, N. Preguica, and R. Rodrigues, "Making geo-replicated systems fast as possible," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 265–278.

[26] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS," in *Proc. 23rd ACM Symp. Operating Syst. Principles*, 2011, pp. 401–416.

[27] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: Definitions, implementation, and programming," *Distrib. Comput.*, vol. 9, no. 1, pp. 37–49, 1995.

[28] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: Bulk enforcement of sequential consistency," in *Proc. 34th Annu. Int. Symp. Comput. Archit. Conf.*, 2007, pp. 278–289.

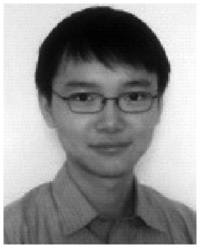
[29] R. V. Rasmussen and M. A. Trick, "Round robin scheduling–a survey," *Eur. J. Operational Res.*, vol. 188, no. 3, pp. 617–636, 2008.

[30] A. Israeli, M. Jalfon, "Token management schemes and random walks yield self-stabilizing mutual exclusion," in *Proc. ACM Symp. Principles Distrib. Comput.*, 1990, pp. 119–131.

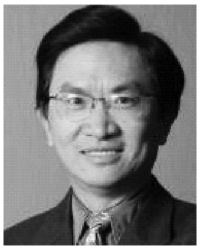
[31] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. 22nd Annu. Int. Symp. Comput. Archit.*, 1995, pp. 24–36.

[32] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Latency-tolerant software distributed shared memory," in *Proc. ATC*, pp. 291–305, 2015.

[33] R. Lublinerman, J. Zhao, Z. Budimlić, S. Chaudhuri, and V. Sarkar, "Delegated isolation," *ACM SIGPLAN Notices*, vol. 46, no. 10, pp. 885–902, 2011.



Yi-Wei Ci is currently an assistant professor of the Institute of Software, Chinese Academy of Sciences. He received his B.S., M.S. and Ph.D. degrees in computer science from Harbin Institute of Technology, PR China, in 2003, 2005, and 2010 respectively. His research interests include distributed computing and fault-tolerant computing.



Michael R. Lyu is currently a professor of Department of Computer Science and Engineering, The Chinese University of Hong Kong. He received the B.S. degree in electrical engineering from National Taiwan University, Taipei, Taiwan, R.O.C., in 1981; the M.S. degree in computer engineering from University of California, Santa Barbara, in 1985; and the Ph.D. degree in computer science from the University of California, Los Angeles, in 1988. His research interests include software reliability engineering, distributed systems, fault-tolerant computing, and machine learning. Dr. Lyu is an ACM Fellow, an IEEE Fellow, an AAAS Fellow, and a Croucher Senior Research Fellow for his contributions to software reliability engineering and software fault tolerance.

Zhan Zhang is currently an associate professor of the School of Computer Science and Technology at the Harbin Institute of Technology. He received the Ph.D. degree in computer systems from Harbin Institute of Technology, PR China, in 2008. His main research interests are fault-tolerant computing and mobile computing.

De-Cheng Zuo is currently a professor of the School of Computer Science and Technology at the Harbin Institute of Technology. He received the Ph.D. degree in computer systems from Harbin Institute of Technology, PR China, in 2001. His main research interests are fault-tolerant computing and mobile computing.

Xiao-Zong Yang is currently a professor of the School of Computer Science and Technology at the Harbin Institute of Technology, PR China. His main research interests are fault-tolerant computing and mobile computing.