LOTS: A Software DSM Supporting Large Object Space*

Benny Wang-Leung Cheung, Cho-Li Wang, and Francis Chi-Moon Lau
Department of Computer Science, The University of Hong Kong
{wlcheung, clwang, fcmlau}@cs.hku.hk

Abstract

Software DSM provides good programmability for cluster computing, but its performance and limited shared memory space for large applications hinder its popularity. This paper introduces LOTS, a C++ runtime library supporting a large shared object space. With its dynamic memory mapping mechanism, LOTS can map more objects, lazily from the local disk to the virtual memory during access, leaving only a trace of control information for each object in the local process space. To our knowledge, LOTS is the first pure runtime software DSM supporting a shared object space larger than the local process space. Our testing shows that LOTS can utilize all the free hard disk space available to support hundreds of gigabytes of shared objects with a small overhead. The scope consistency memory model and a new mixed coherence protocol allow LOTS to achieve better scalability with respect to problem size and cluster size.

Keywords: *distributed shared memory, large object space, dynamic memory mapping mechanism.*

1. Introduction

Cluster computing has emerged as a more popular platform for high performance computing. By connecting commodity PCs or workstations together with a communication network, the cluster is able to provide computing power comparable to supercomputers at a fraction of the cost. This increase in computing capability also allows more complex problems to be solved.

Over the years, two major cluster programming paradigms have been developed, namely *message passing* and *distributed shared memory* (DSM). The latter provides the virtual view of globally shared memory among machines in the cluster (Figure 1), and hides the fact that each machine can only access its own physical memory. When a machine wants to access an object (a variable) not stored in its own memory, the DSM system automatically brings in the object through the network. Application programmers need not insert explicit send and receive commands in the programs, as they would in message-passing systems. It is widely agreed that programs written following the DSM paradigm have better resemblance to sequential ones.

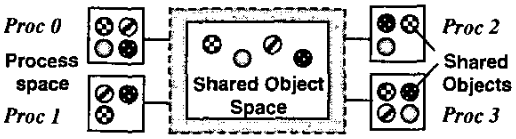


Figure 1. The virtual shared address space provided by DSM (object-based view)

However, over the years, distributed shared memory has not been gaining as wide an acceptance as it should be. One reason is performance. Most of the early-day DSM systems were not efficient, as their machines tend to send too much redundant data to each other. The situation has been improved since the invention of certain relaxed *consistency models* [1, 2, 3, 4] and *coherence protocols* [5, 6, 7]. We believe that it is not completely fair to compare the performance of DSM systems with their message passing counterparts directly, since it is unlikely that the DSM systems will ever be able to send the exact amount of data needed by a machine during execution, while the application programmer has every control over message passing. Thus some of this DSM overhead should be regarded as part of a tradeoff to gain programmer-friendliness.

Another problem of equal significance, if not more, which exists in virtually every DSM system to date, is the limited shared object (memory) space provided by the system. Most of the existing systems try to map all the shared objects to the virtual address space of every process, regardless the process is going to access them or not. Moreover, the same object must be mapped to

[*] This research is supported by Hong Kong RGC Grant HKU-7030/01E and HKU Large Equipment Grant 01021001.

the same virtual addresses in every process, and two shared objects cannot map to the same addresses. For example, in JIAJIA [8], a variable mapped to address `0x4000abcd` in one process must also be mapped to address `0x4000abcd` in other processes running the DSM application. The maximum amount of shared memory that can be created is thus bounded by the size of the process space. For 32-bit machines, at most 4GB of shared memory can be claimed. The actual size is often much smaller, since the upper 1GB of the process space is reserved for the kernel, and the heap and stack areas also occupy some space. Worst of all, this limit is fixed, regardless of how many machines we are using. This violates one original objective of DSM: To provide more memory for each machine by combining the memory available in each machine together.

Such limitation in shared object space hinders the usage and popularity of DSM in two ways: Either the application is too large to fit in the system, or the programmer needs to use less memory-consuming data structures and slower algorithms, so that the application can execute using DSM. Intuitively, one can solve the address space limitation by increasing the number of addressable bits. Some 64-bit machines are available in the market nowadays. However, we firmly believe this should not be the preferred solution due to two main reasons: First, 32-bit machines are still the mainstream in the current commodity market, and they should not be abandoned as they are far more cost-effective than 64-bit counterparts. Second, we believe that scientific problems in real life are becoming more complex, with ever-growing memory consumptions that may not even be able to fit into the 2^{64}-byte address space. Consider some NP-complete problems: If we want a full analysis of all possible moves in a Weiqi (Go) game (about 3^{361} configurations), or an optimal solution to the Rush Hour problem [9], the address space needed to hold the data is enormous. Thus we need an alternative solution that can increase the size of the shared object space beyond the limit posed by the process space.

This paper introduces LOTS (Large ObjecT Space), a DSM system whose main objective is to provide a large shared object space, with the use of the local disk of each machine as backing store. LOTS can provide more than 4GB of shared object space, with an upper bound being the space available in the local hard disk. To let the virtual address space store more objects, only a trace of control information for each object is needed to be resident in the virtual address space, while the actual object data are dynamically but lazily mapped into the virtual memory when being accessed.

For all applications we have tested, the large shared object space support only incurs a small but acceptable overhead of about 5-15% of the total execution time. Such overhead can be cancelled out by using a more efficient memory consistency model and coherence protocol. We shall also demonstrate the large shared object space support. In our test, LOTS can allocate a shared object space of 117.77 GB, fully utilizing the free local hard disk space available.

For the rest of the paper, Section 2 overviews the previous related work. Section 3 discusses the design and implementation of LOTS. Section 4 describes the system testing procedures and the results analysis. The conclusion and future work form Section 5.

2. Related work

The first software DSM system, IVY [10], emerged in 1989. However, its performance was poor due to the use of a strict *Sequential Consistency model* [1]. Later systems tried to improve the efficiency by introducing numerous *relaxed models*, so as to reduce the amount of redundant data sent through the network. The most successful one is TreadMarks [11], which adopts the *Lazy Release Consistency model* [2]. Another one is JIAJIA [8], using *Scope Consistency* [3] as proposed by Princeton University. JIAJIA's open-source-ness provides the base for further DSM research, such as our previous work, JUMP [12], using the *migrating-home protocol* [6] instead of a *home-based* one [7]. The migrating-home protocol allows the home, that is, the master copy of a shared page or object to transfer from one process to another. It differs from the home-based protocol, where the home is fixed, or the *home-less* protocol [11], where there is no notion of home.

One of the latest DSM systems worth mentioning is DOSA [13]. Developed by the same group that introduced TreadMarks, DOSA aims at improving DSM efficiency for both fine-grained and coarse-grained applications. It uses a *handle-based* implementation. Shared object accesses are redirected through an object table to actual objects. By limiting the application programming language to be type-safe (that is, no pointer arithmetic is allowed on the shared objects), together with certain compiler optimizations, DOSA is able to outperform TreadMarks.

All the DSM systems mentioned above do not address the critical issue of the need of a large shared object space. The maximum size of shared memory that can be obtained by using these systems is thus very limited. For example, the maximum amount of shared memory obtainable in TreadMarks is just equal to the minimum RAM size among all machines, while JIAJIA only allows a maximum of 128MB of shared memory. For JIAJIA, the limit can be relaxed, but it will still not be able to store more than 4GB of shared data.

The need for a large shared object space has not been well addressed in the DSM or parallel computing area, but effort has been made by database systems people to implement a persistent store to accommodate very large objects. One of their most well-known techniques is *pointer swizzling* [14], in which artificial, invalid addresses or pointers are translated or swizzled to machine-addressable form during access. Assume we are using 32-bit machines, one most popular way to achieve such translation is to convert source addresses of larger than 32 bits to 32-bit target addresses at page fault time. The generation of the source addresses needs compiler support. When page fault occurs, the runtime system will trigger the page-fault handler to convert source addresses to machine-valid target ones, to allocate memory and to map the actual data to that segment. Objects swizzled into the memory but are unused for a long period of time will be removed (unswizzled), and the contents will be sent back to the local disk or remote server. Systems such as QuickStore [15], ObjectStore [16] and Thor [17] use pointer swizzling or its variant to support a distributed database with size larger than the logical address space.

Although the authors of [14] state that their approach can benefit DSM systems, we have not found a system making use of such strategy to increase the shared object space. As a large shared object space is vital to the usability of the DSM system, we try to implement LOTS by using a strategy similar to pointer swizzling for supporting large object space. We adopt a pure runtime strategy, which can enhance the system's portability as well as reduce software dependency.

We should point out the difference between our large object space support and the *Physical Address Extension* (PAE) feature [18] of Intel chips supported by many operating systems. PAE allows machines with 4GB to 64GB of RAM to be addressable by multiple processes, but most operating systems supporting PAE only allow each process to address 4GB of RAM due to the 32-bit process space limit. The only exception is Solaris 7, which maps the RAM to the process space dynamically during runtime, so that a process can address more than 4GB of RAM. Our large object space support allows more than 4GB of objects to be mapped in the process space, and we need not concern how much RAM is available; the OS virtual memory system takes care of this issue. Theoretically, our large object space support can work whether PAE is enabled or not.

3. Design and implementation of LOTS

This section looks into the design of LOTS, addressing the major features and how they are achieved.

3.1. An overview of LOTS

As mentioned in Section 1, the main objective of LOTS is to provide a large shared object space for cluster applications. Figure 2 shows the simplified scheme describing how this can be done. LOTS is implemented as a C++ language runtime library in Linux. The cluster application making use of LOTS will be compiled with this library using ordinary g++. Each machine runs a copy of the application binary. When the process in one of the machines wants to access the shared object, LOTS will check the status of the object by examining the object control information resident in the virtual memory. If the local copy of the object is not clean, a valid copy of the object will be brought in from a remote machine. On the other hand, if the object data is not mapped to the local virtual memory, it will be brought in from the local disk. The address of the object to be accessed will be returned to the application. We call this *dynamic virtual memory mapping*, the key to allow the total size of all shared objects to exceed the size of the process space.

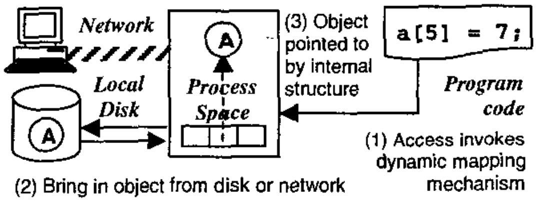

Network
A
(3) Object pointed to by internal structure
a[5] = 7;
Local Disk
Process Space
Program code
A
(1) Access invokes dynamic mapping mechanism
(2) Bring in object from disk or network

Figure 2. Illustrating the dynamic virtual memory mapping mechanism in LOTS

In this paper, we shall mainly focus on the various memory-related features in LOTS that contribute to the large shared object space support and performance improvement. Other components, such as synchronization facilities provided and the communication method used by LOTS, are more straightforward, and we shall only briefly describe them in Section 3.6.

3.2. The memory allocator

A LOTS application first declares shared objects through the LOTS memory allocator, which is a C++ class called `Pointer` with type template. For example, to declare a shared integer pointer or array, we use the code `Pointer <int> i_ptr`. By declaring a shared object, a unique, known-to-all-machines object ID will be generated internally. This object ID will be the key to access all internal data structures for the object. Note that physical memory is not allocated at this moment. Shared memory is allocated to the objects when the application calls the `alloc()` member function in the `Pointer` class, which is analogous to the `malloc()` in C or `new` in C++. The statement `i_ptr.alloc(50)`, for example, allocates memory for 50 integers in each machine, which is pointed to by the variable `i_ptr`.

In LOTS, 1-dimension array is treated as a single object. For pointer of pointers or 2-dimension arrays, LOTS treats each pointer or row as a separate object. Hence the allocation of shared memory follows that in traditional `malloc()` in C or `new` in C++.

The `alloc()` function allocates shared memory by mapping a piece of memory in each machine to any virtual address. The Doug Lea memory allocator [19] which is originally used in the C++ `new` function does not exactly match the requirement of LOTS. Thus the LOTS allocator uses the Linux `mmap()` system call to bypass the Doug Lea allocator. The `alloc()` function also assigns a unique shared memory ID to the claimed piece of memory, and the control information of the shared object is set to point to the start of this allocated address. The mapping state of the memory area is set to "mapped", and the shared state is set to "initial".

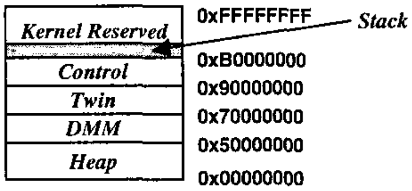


Figure 3. Process space utilization by LOTS

As mentioned, the LOTS allocator uses a different memory allocation strategy from that of the Doug Lea allocator. Figure 3 shows that LOTS partitions the process space into several regions for efficient memory management. The heap, stack and kernel-reserved areas are unchanged. The middle part of the process space from address `0x50000000` to `0xAFFFFFFF` is used for storing shared objects. This area is further subdivided into three equal segments: The *dynamic memory mapping* (DMM) area, for mapping shared object data dynamically during access; the *twin area*, storing a copy of the mapped object before synchronization, so that the actual updates can be calculated at the next synchronization point; and the *control area*, for storing the timestamp and lock information associated with the objects. To simplify the implementation, an object occupying an address A in the DMM area will also occupy the corresponding address (A+`0x20000000`) in the twin area and the control area (A+`0x40000000`).

LOTS is designed to allocate memory in a space-efficient manner, and potentially takes advantage of possible spatial locality. Small objects are assigned to the upper half of the DMM area, while medium-sized objects are assigned in decreasing addresses of the lower half, and large-sized objects are allocated in increasing addresses of the lower half. For small objects of the same size, LOTS tries its best to allocate them in the same page. This will reduce the number of page faults invoked by the virtual memory subsystem of the operating system if these objects are accessed one after the other. Such access patterns can occur frequently, for example, when an application is traversing a linked list, in which every element is of the same size.

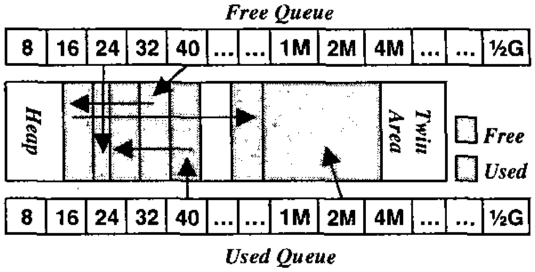


Figure 4. Used and free virtual memory blocks inside DMM area are organized in linked lists, with the head pointed to by various queues

LOTS makes use of an approximation of the best-fit algorithm to map the shared objects to the DMM area. To implement this algorithm, 1024 queues are used, each of them storing either unused or allocated blocks of size within a specified range (Figure 4). If the size of an object is larger than the current contiguous space available in the DMM area, LOTS will trigger the swapping routine to swap out some of the mapped objects. These objects will be sent to the local disk. We shall discuss swapping in more detail in Section 3.3.

3.3. The dynamic memory mapper

After the shared object is declared and memory is allocated, the application is able to access the objects in the same way as a sequential program. It is clear that during each access, LOTS should check the status of the object, so that if the object is currently not mapped in the DMM area, it should be brought in from the local disk. To maintain shared memory consistency, the clean copy should also be brought from the remote machine. As LOTS uses a pure runtime approach, it does not need to add status checking code before each shared object access during compile time. We do not ask application programmers to explicitly insert such code either, since it is error-prone and user-unfriendly. The solution is to make use of the C++ *operator overloading* facility to invoke this status checking.

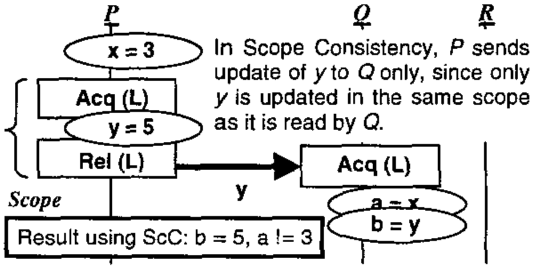
LOTS provides a large collection of operator overloading functions, which are to be invoked before the actual object data is accessed. For example, when a program statement a[5]=1 is executed, where a is a shared array, the [ ] operator will be overloaded. The overloading function then checks if the object a is mapped in the DMM area, and whether the local copy is clean. Consequently, a clean copy will be brought in to the DMM area, and the correct virtual address is returned to the application. The status checking routine is made very light-weight. In most cases, when the object is mapped to the DMM area and the copy is clean, the checking routine is just a table lookup, converting the object ID to the address pointer to be returned.

If the shared object is not mapped, and the DMM area does not have a large enough contiguous space to dynamically map the shared object, swapping has to be performed, as described in Section 3.2. However, there may also be other shared objects going to be accessed. If they are swapped out, incorrect results may be obtained. Consider a program statement a[5]=b[5]+c[5]. If a, b, c are all shared objects, most g++ compilers will overload the [ ] operator associated with object a first, followed by that associated with objects b and c. After all the addresses are resolved, the values of b[5] and c[5] will be read, and the result is stored in the address pointed to by a[5]. If object a is swapped out of the DMM area (e.g., to allow object c to be swapped in) before it is written, a[5] will get the wrong result. LOTS requires all objects referenced in a single statement be resident in the DMM area until the statement is completed. Thus, we introduce the *pinning* mechanism, which is actually a timestamp on each object recording its latest access. Objects with a more recent timestamp are less likely to be swapped out of the DMM area. Hence the swapping mechanism is a combination of the *least-recently-used* (LRU) and the *best-fit* strategy.

The data structure used for table lookup in LOTS is in fact very similar to the handle-based implementation in DOSA, and the table lookup works in a fashion very similar to pointer swizzling. The main difference is that pointer swizzling directly copies the mapped address to the control field of the object. LOTS does not do that, because we want to keep the size of the Pointer class to be the same as that of a pointer, which is four bytes. The Pointer class contains only the object ID, which fits the size of a pointer. This makes pointer arithmetic possible. Users can perform pointer arithmetic on the shared objects, as if they are normal C++ pointers. In fact, the language supported by LOTS is not only type-safe; LOTS supports a limited set of pointer operations. For example, with a shared array a, pointer dereference statements such as *(a+4)=1 are valid in LOTS. This adds user-friendliness to programming as the language syntax of LOTS is very close to the original C++.

3.4. Memory consistency issue

Any DSM system must guarantee that the different copies of objects in different machines are consistent. The rules (*consistency model*) and the implementation mechanisms (*coherence protocol*) both affect very much the efficiency and user-friendliness of the DSM system. LOTS takes much care in these two aspects. It uses *Scope Consistency* as the memory model. Scope Consistency states that "When a process *Q* acquires a lock *L* previously occupied by another processor *P*, all the updates made inside the critical section guarded by lock *L* is made known to processor *Q*". This is a very relaxed memory model, since all updates outside the critical section guarded by lock *L* do not need to propagate to processor *P*, and processes other than *P* and *Q*, such as *R* in Figure 5, are not involved. This would not affect the programming, since the program behavior will be correct as long as the same lock is used to guard the access of the same object. An example of Scope Consistency is shown in Figure 5.



In Scope Consistency, *P* sends update of *y* to *Q* only, since only *y* is updated in the same scope as it is read by *Q*.

Figure 5. Illustrating the behavior of Scope Consistency (ScC)

A coherence protocol that works well with the memory consistency model is also critical in enhancing the efficiency of the DSM system. LOTS combines two well-known coherence protocols to adapt to most of the shared memory access patterns: A *homeless, write-update* protocol for propagating object updates during lock synchronization, and a *migrating-home, write-invalidate* protocol for propagating object updates at a barrier operation. An example of this mixed protocol is shown in Figure 6. The following is the rationale for such a combination: Locks are usually used for objects following a migratory or producer-consumer access pattern. Using write-update will reduce the number of object update requests by the processes accessing the objects, and a homeless approach has proven to be efficient in the case of TreadMarks [11]. For barriers, however, write-update is obviously unfavorable, since this will lead to very heavy all-to-all traffic during barrier synchronization. Thus write-invalidate is more preferred. Moreover, a migrating-home approach will benefit during global synchronization in three ways: First, if there is only a single process writing an object before the barrier, there is no need to propagate the updates. The home is simply migrated to the writer, and this information can be piggybacked on the barrier exit message. Second, the presence of a home avoids the updates of an object to be scattered by two or more processes. This means after the barrier, a process requesting an object only needs to send the message to the home process. Third, after barrier synchronization, all updates are propagated to the home process. Other processes, like *P0*, *P2* and *P3* in Figure 6, are then able to invalidate their own copies of the non-home objects, and free the memory storing the updates. This allows simpler and more efficient bookkeeping.

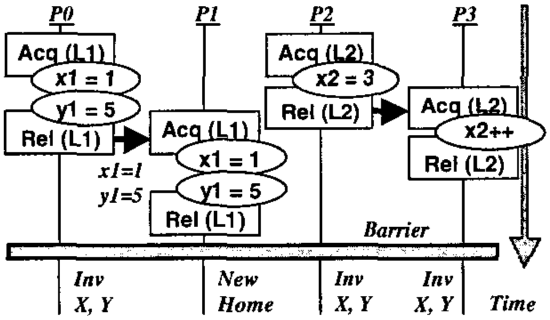


Figure 6. Illustrating the mixed coherence protocol used in LOTS

3.5. Solution to the diff accumulation problem

As discussed, LOTS follows TreadMarks' homeless protocol to propagate object updates synchronized by locks. LOTS also follows TreadMarks' approach in sending diffs (runtime encoding of updates) instead of the whole object. If the object update is sparse, sending diffs is more favorable than sending whole objects as diffs will be much shorter. However, different copies of diffs, representing updates at different times, have to be sent to the object requester in order to obtain the clean copy of the object. In such case, timestamps need to be implemented. Moreover, since diffs with the same timestamp must be sent as a whole, some redundant information may be sent as well, as depicted in Figure 7a. If the object is updated frequently, such as when the object has a migratory update pattern, the redundant traffic will slow down the DSM. This is a phenomenon known as *diff accumulation* [20].

LOTS solves this problem by associating the lock and timestamp information to each field of the shared object in the control area. As shown in Figure 7b, the actual diff is calculated on demand by comparing the timestamp and lock ID information with that provided by the requester, hence eliminating outdated data being sent. This way of calculating diffs may not be favorable a decade ago when TreadMarks was developed. But since the CPU speed has increased at a faster rate than the network speed during these years, as CPU speed for commodity PCs increases from 75MHz to over 2GHz in ten years, while 100-based Ethernet is still the most popular network facility now, we believe that shifting the burden from the network to the CPU is justified.

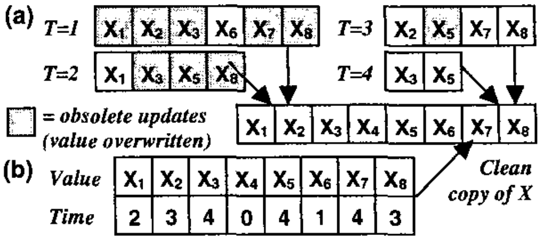


Figure 7. (a) Traditional way of handling diffs, as adopted by TreadMarks, (b) eliminating the diff accumulation problem by storing the last updated time for each field of the object

3.6. Other components in LOTS

Apart from the memory-related features just discussed, LOTS provides users with barriers and locks as synchronization facilities. Their associated memory consistency behaviors are described in Section 3.4. In addition, LOTS provides an alternative type of barrier, `nm_barrier()`, which performs event synchronization only, without any memory synchronization effect. The `nm_barrier()` will be a good option if the application program uses the same lock to guard all accesses of the same shared object before and after the barrier.

Machines executing LOTS communicate with each other through dedicated point-to-point socket channels, following the UDP/IP protocol. LOTS also adopts a simple flow control algorithm, which is slightly more efficient than that of the TCP protocol. SIGIO handlers are implemented to handle incoming messages.

4. Testing and results

As the main objective of LOTS is to provide a large shared object space for cluster computing with reasonable performance, the testing should aim at both the efficiency of the system and the capability of providing a large shared object space. Hence we performed two different tests. First, we compared the execution time of four applications under LOTS with that under JIAJIA V1.1, a page-based DSM using a home-based protocol. JIAJIA was chosen since it is open-source, allowing us to trace how the memory consistency model, coherence protocol and network communication work, so that we can analyze the testing results in detail. Second, we ran an application that needed more virtual memory than is locally available, to see if the application worked correctly.

4.1. Performance testing and results analysis

For the first test, since both LOTS and JIAJIA use UDP/IP protocol with similar flow control mechanisms as discussed in Section 3.6, any difference in execution time can be attributed to one or more of the following three factors: (1) The coherence protocol efficiency, (2) the overhead due to the difference in object-based and page-based DSM. Such overhead is larger in object-based DSM since the system must check the state of an object at every access, and (3) the overhead for supporting a large object space in LOTS. The size of (3) can be easily determined by disabling the large object space support as well as the pinning mechanism in LOTS. The size of (2) can also be found by measuring the amortized time used in calling the access checking routine. The remaining difference in time, if any, can be classified as the difference in protocol efficiency.

The first test was performed on a cluster of 16 Pentium IV 2GHz machines, connected by a 100-based Ethernet using a 24-port Fast-Ethernet switch. Each machine had 128 MB of RAM and ran a copy of Linux Fedora. We used four common scientific applications: ME (merge sort), LU (LU factorization), SOR (successive red-black iterations) with 256 iterations, and RX (radix sort). Small problem sizes were chosen so that the programs could work on both JIAJIA and LOTS to compare the performance. We also ran the programs under LOTS-x, LOTS without the large object space support, to determine how large factor (3) is.

The testing results for Test 1 are shown in Figure 8. The graph shows that LOTS runs faster than JIAJIA on most data points, except RX, where LOTS becomes slightly slower than JIAJIA as more processes are used. We shall look at the reasons why in more detail.

LOTS runs ME faster than JIAJIA in general. The reason is that objects in ME share a migratory access pattern. When two sorted sub-arrays are merged together in one of the merging phases, one of the processes handles the merging. Thus at any time, half of the total data is migrated. With the home-based protocol in JIAJIA, only *1/p* of all the data will be accessed locally in the home node, since JIAJIA uses a round-robin home allocation on pages. In LOTS, however, using the migrating-home protocol (ME only uses barriers to synchronize), half of all the data are accessed in the migrated home node. So LOTS can take advantage of that to access more home objects, thus reducing data traffic and improving the performance. Note that ME does not show a speedup for increasing number of processes, because only the merging time is counted while the local sorting time is excluded. Thus more processes mean more stages, hence longer time.

The LU program demonstrates the adverse effect of false sharing on page-based DSM. In LU, one process always updates a row in the source matrix to do the factorization, while all others will read the result of that row to update the rows they are responsible to update. If the row size does not fit an integral multiple of pages, both read-write and write-write false sharing, i.e., two or more processes accessing different part of the same page, can occur. Even if false sharing does not occur, the home-based protocol still suffers from the heavy network traffic due to excessive page requests by all processes (readers). In LOTS, each row is a unique object. False sharing will not happen, since only one process will write to a particular row at any time. By eliminating false sharing, LOTS improves the overall performance drastically by up to about 80%.

LOTS also outperforms JIAJIA on SOR, a program used to approximate engineering problems that involve integrations. In SOR, the two matrices (red and black) are divided into *p* horizontal "slices", and each process is responsible to update its own "slice" in each of the two matrices, according to the values of the adjacent positions in the other matrix. Thus it is clear that each object (row) is updated by a single process throughout the whole program, and only the rows at the edge of the "slides" are read-shared by two processes. This access pattern certainly favors the migrating-home protocol in LOTS, and testing results support the claim.

RX performs better on LOTS when two or four processes are used, but it runs slightly slower than JIAJIA in the *p*=8 case. This can also be explained by the object access patterns. In our implementation of RX, 256 shared buckets (objects) are initialized to store the numbers during sorting. Each bucket, of size an integral multiple of a page, is accessed by a processor at a time (concurrent access is prohibited by barriers). However, during the execution, *1/p* of the total number of buckets are always accessed by a single process, while others are accessed alternatively by two processes. The former access pattern favors LOTS' migrating-home protocol since the accessing process will be home after the first barrier. But for the alternate access pattern, migrating the home to the latest writer during the barrier gives little benefits, since the bucket will be requested next by the process that originally owns it. As the number of processes *p* increases, the portion of buckets (hence shared objects) having this ping-pong access pattern also increases. The performance of LOTS thus degrades and lags behind JIAJIA.

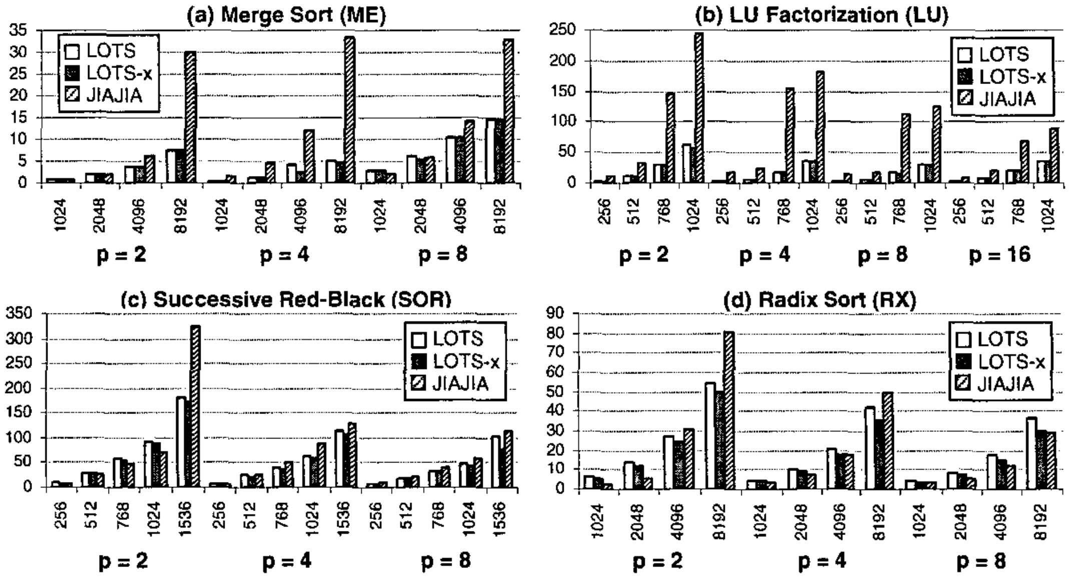


Figure 8. Execution performance of LOTS (with and without large object space support) compared with JIAJIA V1.1. The x-axis is problem size, while the y-axis is time in seconds

4.2. Overhead for large object support

We have also obtained the performance of LOTS-x by disabling the large object support in LOTS. The difference between LOTS and LOTS-x corresponds to the overhead of LOTS supporting large objects. Results show that this overhead depends on the number of shared object accesses occurred in the program, since every time an object is accessed, LOTS has to perform access checking and pinning. For applications with frequent shared object accesses, such as RX, the overhead is around 10-15% of the total execution time. For other applications, the overhead seldom exceeds 5%.

Note that the above figure does not include the access checking overhead, since being an object-based system, LOTS needs to check the shared object state so that the clean copy of the object can be obtained before it is actually accessed. We inserted timing functions in the applications to record the access checking overhead. Results show that each access check needs an average of 20 to 25 nanoseconds in a 2GHz Pentium IV machine. Although this value is small, it is critical since an application may perform access checks for millions or even billions of times. For example, in our implementation of SOR with problem size of 1024, when four processors are used, the number of object access checks per process throughout the execution is about $1.5*10^9$ times. This means around 30-37 seconds out of 55 seconds of execution time is spent on access checking. If the checking overhead is reduced, we expect a good improvement on the efficiency of LOTS.

4.3. Testing large object space support

The second test, for testing the large object space support, was performed on a cluster of four machines. In the test, the machines try to allocate a shared large 2-dimension integer array of *X* rows, with a total size exceeding 4GB. This means that *X* shared objects are created. Then DSM read/write statements are inserted to access the shared objects. Large object space support is thus invoked. The program is made simple (just adding some numbers held by each process) since we only want to demonstrate the invocation of the large object space support, but there is still a lot of data movement performed when the objects are swapped in and out of the disk. In this program, every object is swapped out once, thus more than 4GB data is written to the disk. It is expected the execution time is to be dominated by the disk access time. We used different machine configurations with different CPU speeds, including a 4-node Pentium III PC cluster, a 4-node Pentium IV PC cluster, and a 4-node 4-way Xeon Pentium III SMP cluster. Different operating system versions, RAM sizes and values of *X* are selected to test the impact of each of the configurations to the execution time, as well as how large the shared object space can be. The results are shown in Table 1. The data shows that using Pentium III 733MHz machines, in RedHat 6.2, the execution time is 1114 seconds, while RedHat 9.0 needs 976 seconds. This suggests that RedHat 9.0 has a better I/O support than RedHat 6.2, but frequent disk I/O to and from the disk still takes up hundreds of seconds. To be more specific, the disk read/write time due to the large object space support in LOTS already takes 1004 seconds in RedHat 6.2, and 666 seconds in RedHat 9.0. This has not included the I/O time caused by the virtual memory swapping invoked by the operating system. By using faster machines with more advanced I/O systems, the execution time will be much reduced. When we performed the same test on four Pentium IV 2GHz machines with Fedora 9.0, the time needed is just 142 seconds. The efficiency of the large object space support thus counts on a fast I/O facility.

Table 1. Testing the large object space support of LOTS on various platforms and various shared object size

Machine/ CPU Speed	OS	RAM Size (MB)	No. of Shared Objs (X) #	Per Object Size (MB)	Total Shared Object Size (GB) #	Exec Time (sec)
P3-733MHz	Redhat 6.2	384	33	128	4.125	1114
	Redhat 9.0	128	33	128	4.125	976
P4-2GHz	Fedora	512	33	128	4.125	142
		512	132	128	16.50	373
Dell P6300*	Fedora	1024	132	128	16.50	507
		1024	132	511	65.87	2112
		1024	201	511	100.30	3227
		1024	236	511	117.77	3839

* The Dell Poweredge 6300 machines, with 4-way SMP, each CPU is of Xeon Pentium III 500MHz type

The number of disk writes in each run = the number of shared objects, and the total amount of data written to disk = the total shared object size

We also performed the experiment on four Dell Poweredge 6300 machines, with 4-way SMP. Though they are not new models, the 4-way SMP configuration allows them to act as file servers for the cluster, as they are equipped with two 72GB SCSI hard disks. In our experiment, we are able to exhaust all the free space available in the hard disks for storing shared objects, obtaining a shared object space of 117.77GB. Thus LOTS is able to support a large shared object space greater than the 4GB process space. This size is only limited by the local hard disk free space, and the single object size is only limited by the size of the DMM area, which is 512MB in our current implementation.

5. Conclusion and future work

We firmly believe that a large object space is essential for the wide acceptance of DSM, as it allows a wider range of scientific and real-life applications to be run, and exploits the cost-efficiency of clusters without sacrificing user-friendliness. This paper demonstrates LOTS, a software DSM capable of enlarging the shared object space beyond the size of the process space through the dynamic object mapping mechanism. All the mapping is done automatically through table lookup at runtime, without any compiler preprocessing, thanks to the C++ operator overloading facility. To the best of our knowledge, this is the first pure runtime DSM system to support a large shared object space. It offers reasonable performance and retains good programmability, as the programming interface is very similar to C++. Only a minimal set of functions, such as memory allocation function, locks and barriers are exported to users for using the DSM system.

From the testing results, we find the performance of LOTS to be quite acceptable. This is attributed to its mixed coherence protocol, which works efficiently on many access patterns (migratory and single-writer-multiple-readers), its object-based nature (eliminating some false sharing in page-based systems) and also the mechanism to eliminate diff accumulation. As a result, LOTS is more scalable than its counterparts, both in terms of problem size and cluster size. Currently, LOTS is designed to support up to 256 processes.

We have performed the testing using the first completed version of LOTS. There is room for further improvement. For example, the code can be further optimized for better performance. In particular, the access checking of LOTS can be made more lightweight. A small improvement can lead to huge performance gain since access checking is called very frequently. The encoding and decoding of messages can also be improved as well. Because sockets are used, the maximum message size cannot exceed 64KB. Hence for large messages, such as carrying a large object, we need to split the message into multiple parts before sending. Currently, although we can send out partial messages during encoding, the receiver side must receive all the message fragments in order to rebuild the original message before decoding. This leads to a performance bottleneck, and is also memory consuming. We should find a way so that the receiver can work on partial messages as soon as they are received.

The pinning mechanism discussed in Section 3.3, although avoids swapping away the accessing objects in some sense, is not a perfect solution. The system can do nothing if all the objects currently mapped in the

DMM area are accessed in the same program statement (this is rare, but can occur if objects are very large). A possible solution is to swap out unused partial objects. Furthermore, the swapping can also be done not only to and from local hard disks, but remote ones as well. Such feasibilities will be further investigated.

Finally, the mixed coherence protocol in LOTS can be refined to further enhance performance. Developing an adaptive coherence protocol is one feasibility. Some possible adaptations include that between using write-invalidate against write-update, as well as sending the whole object verses partial diffs according to the object size and access patterns. With a good adaptation, the network traffic and hence application execution time can be reduced.

References

[1] L. Lamport. How to make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690-691, September 1979.

[2] P. Keleher, A. L. Cox, W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*, pages 13-21, May 1992.

[3] L. Iftode, J. P. Singh and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *Proc. of the 8th ACM Annual Symposium on Parallel Algorithms and Architectures (SPAA'96)*, pages 277-287, June 1996.

[4] B. N. Bershad and M. J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. *CMU-CS-91-170*.

[5] W. Hu, W. Shi and Z. Tang. A Lock-based Cache Coherence Protocol for Scope Consistency. *Journal of Computer Science and Technology*, 13(2):97-109, March 1998.

[6] B. Cheung, C. L. Wang and K. Hwang, A Migrating-Home Protocol for Implementing Scope Consistency Model on a Cluster of Workstations. In the *1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Las Vegas, Nevada, USA.

[7] Y. Zhou, L. Iftode and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Memory Virtual Memory Systems. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 75-88, October 1996.

[8] W. Hu, W. Shi and Z. Tang. JIAJIA: An SVM System Based on A New Cache Coherence Protocol. In *Proc. Of the High-Performance Computing and Networking Europe 1999 (HPCN'99)*, pages 463-472, April 1999.

[9] H. Fernau, T. Hagerup, N. Nishimura, P. Ragde and K. Reinhardt. On the parameterized complexity of a generalized Rush Hour puzzle. In *Proc. of the 15th Canadian Conference on Computational Geometry (CCCG 2003)*, Halifax, Nova Scotia, August 2003.

[10] Kai Li. Shared Virtual Memory on Loosely Coupled Multiprocessors. PhD thesis, Yale University, September 1986.

[11] P. Keleher, S. Dwarkadas, A. L. Cox and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115-131, January 1994.

[12] B. W. L. Cheung, C. L. Wang, and F. C. M. Lau. Building a Global Object Space for Supporting Single System Image on a Cluster, *Annual Review of Scalable Computing*, Chapter 6, Volume 4, Year 2002.

[13] Y. Hu, W. Yu, A. Cox, D. Wallach and W. Zwaenepoel. Run-time support for distributed sharing in safe languages. In *ACM Transactions on Computer Systems (TOCS)*, Vol. 1, Issue 21, pages 1-35, February 2003.

[14] P. R. Wilson. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware. In *Proc. of the Intl. Workshop on Object Orientation in Operating Systems*, pages 364–377, Paris, France, Sept. 1992.

[15] S. J. White and D. J. DeWitt. QuickStore: A High Performance Mapped Object Store (1994). In *ACM SIGMOD Int. Conf. On Management of Data*, pages 395-406, Minneapolis, MN, May 1994.

[16] C. Lamb, G. Landis, J. Orenstein, D. Weinreb, "The ObjectStore Database System", *Communications of the ACM*, Vol. 34, No. 10, October 1991.

[17] M. Castro, A. Adya, B. Liskov and A. C. Myers. HAC: Hybrid Adaptive Caching for Distributed Storage Systems. In *Proc. Of the ACM Symposium on Operating System Principles (SOSP'97)*, Saint-Malo, France, October 1997.

[18] The Intel Extended Server Memory Architecture. Intel Corporation, 1998.

[19] A Memory Allocator by Doug Lea. *http://gee.cs.oswego.edu/dl/html/malloc.html*

[20] A. L. Cox, E. de Lara, C. Hu and W. Zwaenepoel. A Performance Comparison of Home-less and Home-based Lazy Release Consistency Protocols in Software Shared Memory. In *Proc. of the 5th High Performance Computer Architecture Conference*, January 1999.

[21] R. Buyya, K. Branson, J. Giddy and D. Abramson. The Virtual Laboratory: a toolset to enable distributed molecular modeling for drug designing on the World-Wide Grid. In *Concurrency and Computation Pracice and Experience 2003*, 15:1-25.

[22] R. Buyya, S. Date, Y. Mizuno-Matsumoto, S. Venugopal and D. Abramson, Neuroscience Instrumentation and Distributed Analysis of Brain Activity Data: A Case for eScience on Global Grids, Journal of Concurrency and Computation: Practice and Experience, Wiley Press, USA (accepted in Jan. 2004 and in print).