Run-Time Support for Distributed Sharing in Safe Languages

Y. CHARLIE HU
Purdue University
and
WEIMIN YU, ALAN COX, DAN WALLACH, and WILLY ZWAENEPOEL
Rice University

We present a new run-time system that supports object sharing in a distributed system. The key insight in this system is that a handle-based implementation of such a system enables efficient and transparent sharing of data with both fine- and coarse-grained access patterns. In addition, it supports efficient execution of garbage-collected programs. In contrast, conventional distributed shared memory (DSM) systems are limited to providing only one granularity with good performance, and have experienced difficulty in efficiently supporting garbage collection. A safe language, in which no pointer arithmetic is allowed, can transparently be compiled into a handle-based system and constitutes its preferred mode of use. A programmer can also directly use a handle-based programming model that avoids pointer arithmetic on the handles, and achieve the same performance but without the programming benefits of a safe programming language. This new run-time system, DOSA (Distributed Object Sharing Architecture), provides a shared object space abstraction rather than a shared address space abstraction. The key to its efficiency is the observation that a handle-based distributed implementation permits VM-based access and modification detection without suffering false sharing for fine-grained access patterns. We compare DOSA to TreadMarks, a conventional DSM system that is efficient at handling coarse-grained sharing. The performance of fine-grained applications and garbage-collected applications is considerably better than in TreadMarks, and the performance of coarse-grained applications is nearly as good as in TreadMarks. Inasmuch as the performance of such applications is already good in TreadMarks, we consider this an acceptable performance penalty.

Categories and Subject Descriptors: D.4.2 [**Software**]: Operating Systems—*storage management*; D.3.4 [**Software**]: Programming Languages—*processors*

General Terms: Languages, Management, Performance

Additional Key Words and Phrases: Communications, distributed sharing, memory consistency, safe programming languages

1. INTRODUCTION

In recent years there has been increasing interest in supporting scientific computing in modern languages, particularly in Java (e.g., JavaGrande).

Authors' addresses: Y. C. Hu, School of Electrical and Computer Engineering, Purdue University, 1285 Electrical Engineering Building, West Lafayette, Indiana 47907; email: ychu@purdue.edu; W. Yu, A. Cox, D. Wallach, and W. Zwaenepoel, Department of Computer Science, Rice University, 6100 Main Street, Houston, Texas 77005; email: {weimin,alc,dwallach,willy}@cs.rice.edu.

Among these efforts are numerous attempts at building fast Java virtual machines (e.g., Adl-Tabatabai et al. [1998], Burke et al. [1999], and Wilkinson [1996]), compilation into native code (e.g., Timber and Tower Technologies), compiler optimizations specifically for scientific codes (e.g., Budimlic and Kennedy [1997, 1999]), development of suitable scientific run-time libraries (e.g., Casanova et al. [1997]), and support for parallel and distributed computing (e.g. Christiansen et al. [1997]) and Fox and Furmanski [1996]). This article falls in the latter category.

We investigate run-time support for executing multithreaded scientific codes in modern languages on a cluster of PCs. The presence of a network between the PCs in the cluster is transparent to the programmer. Specifically, object references can be followed across machine boundaries, and no special API (such as, e.g., Java RMI) is necessary to access objects on another machine. The rationale is similar to the rationale for conventional software distributed shared memory (DSM) (e.g., Li and Hudak [1989]): it allows for easier development and faster migration of multithreaded codes to cluster environments.

This article addresses the issue of building such a distributed sharing system that exhibits good performance for a wide range of sharing patterns, from fine-grained to coarse-grained. Building a single system that covers such a wide range has proven to be a vexing problem. Indeed, DSM systems have been divided into those offering support for coarse-grained sharing or for fine-grained sharing. Coarse-grained sharing systems are typically page-based, and use the virtual memory hardware for access and modification detection. Although relaxed memory models [Gharachorloo et al. 1990; Keleher et al. 1992] and multiple-writer protocols [Carter et al. 1995] relieve the impact of the large page size, fine-grained sharing and false-sharing remain problematic [Amza et al. 1997]. Fine-grained sharing systems typically augment the code with instructions to detect reads and writes [Scales et al. 1996; Veldema et al. 2001a], freeing them from the large size of the consistency unit in virtual memory-based systems, but introducing per-access overhead that reduces performance for coarse-grained applications [Dwarkadas et al. 1999]. Both types of systems experience problems when it comes to efficiently supporting garbage collection, because they do not distinguish memory updates by the application from a memory update caused by the garbage collector [Yu and Cox 1996].

Modern languages all offer some notion of language safety. For the purposes of this article language safety can be narrowly interpreted to mean absence of pointer arithmetic. The key result of this work is that, when a safe language is used, a single system can transparently support sharing of both fine- and coarse-grained objects across machine boundaries. Underlying the efficiency of our approach is the observation that in the absence of pointer arithmetic the compiler can transparently redirect object accesses through a handle table. This in turn allows inexpensive *per-object* access and modification detection using virtual memory protection [Brecht and Sandhu 1999; Itzkovitz and Schuster 1999]. Another contribution of this article is that handle-based systems interact much better with garbage collection algorithms present in safe language implementations, leading to good performance for garbage-collected applications.

Our results are more broadly applicable and not necessarily restricted to safe languages. For programs in unsafe languages, the same results apply but in a nontransparent manner. The programmer must use a restrictive programming model, using a handle to access all objects and refraining from pointer arithmetic on the handles. Viewed in this light, this article also contributes to the area of distributed shared memory systems. We demonstrate that, when a handle-based programming model is used without pointer arithmetic on the handles, a single system can in fact support both fine and coarse granularities with good efficiency.

We have implemented these ideas in a system we call DOSA (Distributed Object System Architecture). To evaluate its performance, we have compared the performance of DOSA against that of the TreadMarks coarse-grained DSM system [Amza et al. 1996] for a number of applications. We chose TreadMarks as the baseline for performance comparison for two reasons: it is the most widely used coarse-grained DSM system, and believed to be the most efficient; and DOSA is derived from the same code base as TreadMarks, minimizing the differences due to coding. Unfortunately, there is no similarly available implementation of fine-grained shared memory, so we cannot make an explicit comparison with such a system. Our performance evaluation substantiates the following claims.

1. The performance of fine-grained applications is considerably better than in TreadMarks.
2. The performance of coarse-grained applications is nearly as good as in TreadMarks.
3. The performance of garbage-collected applications is considerably better than in TreadMarks.

For the applications used in this article, we observe performance improvements as high as 98% for fine-grained applications and 65% for garbage-collected applications, whereas the maximum performance degradation for coarse-grained applications is 6%.

The outline of the rest of this article is as follows. Section 2 explains the key technical idea: how handle-based systems allow efficient support for fine- and coarse-grained objects. Section 3 describes the API and the memory model of the DOSA system. Section 4 describes its implementation and various optimizations to the basic handle-based approach. Section 5 discusses the experimental methodology that we have used. Section 6 presents overall results for fine-grained, coarse-grained, and garbage-collected applications. Section 7 presents a breakdown of the contributions of the various optimizations. Section 8 discusses related work. Section 9 concludes the article.

2. KEY TECHNICAL CONCEPT

The key insight in this work is that a handle-based implementation enables efficient and transparent sharing of data with both fine- and coarse-grained access patterns (see Figure 1). We define a handle-based system as one in which

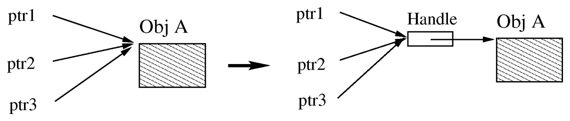

Fig. 1. Pointer safety in a safe language enables a handle-based implementation.

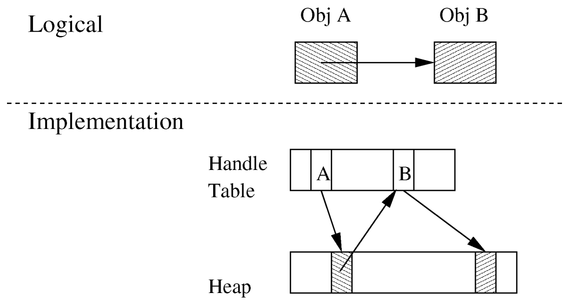

Fig. 2. Objects with handles.

all references to objects are redirected through a handle for the object. No direct references to the object are allowed, other than through the handle, and no pointer arithmetic is allowed on the handles. A *handle table* contains all the handles (see Figure 2). Each object in the language is uniquely identified by an object identifier (OID) that also serves as an index into the handle table for that object. A safe language in which no pointer arithmetic is allowed can easily be compiled into a handle-based system (e.g., Sun's classic JVM for Java, and early implementations of Smalltalk [Goldberg and Robson 1983; Deutsch and Schiffman 1984]), and constitutes the preferred use of the proposed run-time system.

In a handle-based run-time system, it is easy to relocate objects in memory. It suffices to change the corresponding entry in the handle table after a relocation. No other changes need to be made, as all references are redirected through the handle table. Extending this simple observation allows an efficient distributed implementation of these languages. Specifically (see Figure 3), a handle table representing all shared objects is present on each processor. A globally unique OID identifies each object, and serves as an entry in the handle tables. As before, each handle table entry contains a pointer to the location in memory where the object resides on that processor. The consistency protocol can then be implemented solely in terms of OIDs, because these are the only references that appear in any of the objects. Furthermore, the same object may be allocated at different virtual memory addresses on different processors. It suffices for the handle table entry on each processor to point to the proper location. In other words, although the programmer retains the abstraction of a single object space, it is no longer the case that all of memory is virtually shared, and that all objects have to reside at the same virtual address at all processors, as is the case in conventional DSM systems.

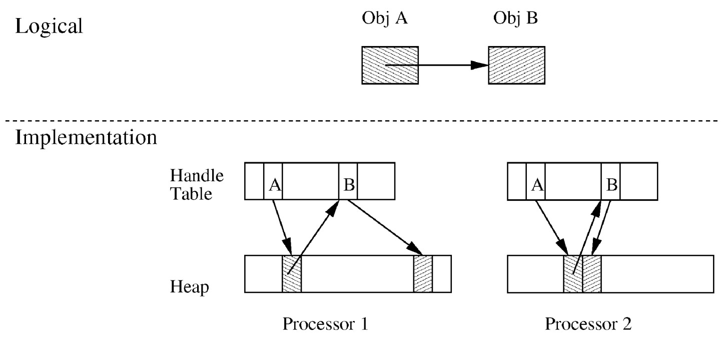

Fig. 3. Shared objects identified by unique OIDs.

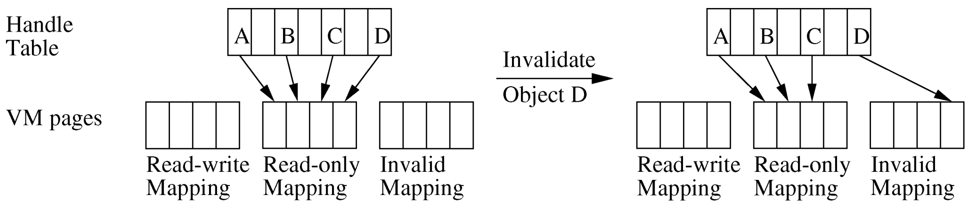

Fig. 4. Access detection using the handle pointers. All three regions of virtual memory point to the same region of physical memory, but the protection on each of the regions is different: read-write, read-only, and invalid. Switching the handle pointer of an object among the three regions causes the access protection of the object to be changed accordingly, without affecting the access protection of other objects in the same page.

In order to provide good performance for coarse-grained applications, we continue to use the virtual memory system for access detection, thereby avoiding the overhead of instrumentation incurred in previous fine-grained systems. Fine-grained access using VM techniques is then provided as follows. Although only a single physical copy of each object exists on a single processor, each object can be accessed through three VM mappings. All three point to the same physical location in memory, but with three different protection attributes: invalid, read-only, or read-write. An object's access protection is changed by switching the object's handle among the different mappings. This changes the access protection *for that object only*. The access protection of the other objects in the same page remain unaffected. Consider the example in Figure 4. A physical page on a processor contains four objects A, B, C, and D, one of which, D, is written on a different processor. This modification is communicated between processors through the consistency protocol, and results in the invalid mapping being set for this object. Access to other objects can continue, unperturbed by this change, thus eliminating false sharing between objects in the same page.

3. API AND MEMORY MODEL

We now turn to a detailed discussion of the system that we have implemented to evaluate the efficiency of the handle-based implementation of shared object systems. We first discuss the system's interface. Its implementation is discussed in the next section.

3.1 API

The programming model is a shared space of objects, in which object references are safe. The fields of an object must be accessed through a reference to the object. No other access is allowed. In particular, no pointer arithmetic is allowed on the references to access the fields of an object.

The programmer is responsible for creating and destroying threads of control, and for the necessary synchronization to ensure orderly access by these threads to the object space. Various synchronization mechanisms may be used, such as semaphores, locks, barriers, monitors, and so on.

An individual object must not be concurrently written by different threads, even if those threads write different data items in the object. If two threads write to the same object, they should synchronize between their writes. Arrays are treated as collections of objects, and therefore their elements can be written concurrently. Of course, for correctness, the different processes must write to disjoint elements in the arrays. Similarly, concurrent write and read accesses by different threads to the same object are not allowed. The single-writer nature of individual objects is not an inherent part of the design of our system, but we have found that it corresponds to common usage, and is therefore not restrictive. As shown in Section 4, it allows us to use an efficient single-writer protocol for individual objects.

3.2 Memory Model: Release Consistency

In order to combat long network latencies, many distributed shared memory or shared object systems have adopted some form of relaxed consistency. DOSA's object space is release consistent (RC) [Gharachorloo et al. 1990]. In RC, *ordinary* accesses to shared data are distinguished from *synchronization* accesses, with the latter category divided into *acquires* and *releases*. An acquire roughly corresponds to a request for access to data, such as a lock acquire, a wait at a condition variable, or a barrier departure. A release corresponds to the granting of such a request, such as a lock release, a signal on a condition variable, or a barrier arrival. RC requires ordinary shared memory updates by a processor p to become visible to another processor q only when a subsequent release by p becomes visible to q via some chain of synchronization events. Parallel programs that are properly synchronized (i.e., have a release–acquire pair between conflicting accesses to shared data) behave as expected on the conventional sequentially consistent shared memory model [Lamport 1979].

4. IMPLEMENTATION

We focus in this section on the consistency maintenance of individual objects. Synchronization is implemented as in TreadMarks [Amza et al. 1996]. Lock and barrier synchronization are supported. Each lock has a lock manager that keeps track of which processor has most recently requested the lock. The lock manager forwards a lock acquire request to that processor, where it is queued until that processor releases the lock. Each barrier has a barrier manager that implements a barrier by waiting for barrier arrival messages from all other processors, and then sending barrier departure messages to all of them.

We first discuss our implementation of release consistency. Next, we describe the handle table and the supporting data structures. We then turn to the use of the handle table to perform per-object access and modification detection using virtual memory protection. We conclude this section with a discussion of various optimizations.

4.1 Release Consistency Implementation

DOSA uses a single-writer, lazy invalidate protocol to maintain release consistency. The *lazy* implementation delays the propagation of consistency information until the time of an acquire. At that time, the releaser informs the acquiring processor which *objects* have been modified, and these objects are then invalidated. Access to an invalidated object causes a protection fault, which in turn causes an up-to-date version of the object to be brought in.

To implement the lazy invalidate protocol, the execution of each process is divided into *intervals*, defined as the epochs between consecutive synchronization operations on a processor. Each interval is labeled with an interval number, which is simply a count of the number of intervals on a processor. Intervals of different processors are partially ordered [Adve and Hill 1990]: (i) intervals on a single processor are totally ordered by program order, and (ii) an interval on processor p precedes an interval on processor q if the interval of q begins with the acquire corresponding to the release that concluded the interval of p.

The above partial order can be represented concisely by assigning a vector timestamp to each interval. On processor i, the ith element of the vector timestamp for a particular interval is equal to its interval number. The elements $j \neq i$ of the vector timestamp are equal to the interval number of processor j at the time of the last acquire from i to j. The *current vector timestamp* on processor i is the vector timestamp of the current interval on processor i. Vector timestamps are maintained by the consistency protocol. When processor i performs an acquire from processor j, processor j sends its current vector timestamp to processor i. Processor i then computes the pairwise maximum of the elements in its current vector timestamp and in the one it received in the message from j. The result of this computation becomes the current vector timestamp of processor i.

Each object that was modified during an interval is recorded in a *write notice*. Each write notice has an associated processor identifier and vector timestamp, indicating where and when the modification of the object occurred. Arrival of a write notice for an object causes the acquiring processor to *invalidate* its local copy, and to set the *last writer* field in the handle table entry to the processor identifier in the write notice. A processor incurs a page fault on the first access to an invalidated object, and obtains an up-to-date version of that object from the processor indicated in the *last writer* field.

To avoid repeated sending of write notices, a processor i performing an acquire sends its current vector timestamp, and the responding processor sends only those write notices with a vector timestamp between the vector timestamp received from i and its own current vector timestamp.

Creating one write notice per object allows consistency to be maintained on a per-object basis. For applications with a large number of objects, this is inefficient. In Section 4.5, we describe a number of optimizations that largely remove this inefficiency.

4.2 Data Structures

A handle table is present on each processor. The handle table is indexed by a globally unique object identifier (OID). Each entry in the handle table contains the corresponding object's address in local virtual memory. This address may be different from processor to processor (see Figure 3). The object's local state (i.e., invalid, read-only, or read-write) is implicitly reflected in the handle table entry. Depending on the object's state, the handle pointer points to one of three possible VM mappings for the object's local physical address (see Section 4.3). The handle table entry contains a *last writer* field, indicating from which processor to fetch an up-to-date copy of the object on an access miss. Finally, a handle table entry contains a field linking it with the handle table entries of other objects allocated in the same page.

A few auxiliary data structures are maintained as well. An *inverse object table*, implemented as a hash table, is used by the page fault handler to translate a faulting address to an OID. Each processor maintains a per-page linked list of objects allocated in that page. This list is used to implement communication aggregation (see Section 4.5.2). Finally, each processor maintains its vector timestamp and an efficient data structure for sending write notices when responding to an acquire.

To avoid translation between OIDs and local memory addresses on different processors during the exchange of objects, the handle table is located at the same virtual address on all processors, and OIDs are simply assigned as the virtual addresses of the entries in the handle table.

Objects are instantiated by a `new` operation or the equivalent. An OID is generated, and memory is allocated on the local processor to hold the object. In order to minimize synchronization overhead for unique OID generation, each processor is allocated a large chunk of OIDs at once, and this chunk allocation is protected by a global lock. Each processor then independently generates OIDs from its chunk.

4.3 Switching Protection

DOSA relies on hardware page protection to detect accesses to invalid objects and write accesses to read-only objects. We create three nonoverlapping virtual address regions that map to the same physical memory (see Figure 4). An object can thus be viewed through any of the three corresponding addresses from the three mappings. DOSA assigns the access permissions to the three mappings to be invalid, read-only, and read-write, respectively. During program execution, it regulates access to a shared object by adjusting the object's handle to point to one of the three mappings. In addition to providing per-object access control, this approach has the substantial additional benefit that no kernel-based memory protection operations are necessary after the initialization of all mappings.

As a practical matter, the three mappings of a shared object differ only in the two leading bits of their addresses. Therefore, changing protection is a simple bit masking operation.

4.4 Handling Page Faults

When a page fault occurs on access to an invalid object, the up-to-date value of the object is fetched. When a page fault occurs as a result of a write access to a read-only object, a write notice for that object is created. In either case, protection is changed and the faulting instruction is restarted.

Some care has to be taken to properly restart the faulting instruction. Accessing an object through a handle involves two memory dereferences: the first to obtain the object's address and the second to read or write the data field within the object. On most architectures these two dereferences are implemented by two distinct instructions: the first loads the address of the object into a register, and the second uses that register to read or write a field in the object. The page fault occurs during the second instruction. Simply updating the object's address in the handle table does not resolve the page fault, because the (old) address still resides in the register that is used by the second instruction.

Therefore, to resolve the page fault, the page fault handler also needs to decode the register being used by the faulting instruction, and update it as well as the handle in memory.

4.5 Optimizations

The DOSA system as described so far supports sharing at object granularity. We next describe a set of optimizations that reduce the overhead of per-object consistency maintenance for applications with a large number of objects. We also describe an optimization aimed at reducing the overhead of indirection through the handle table for array-based applications.

4.5.1 *Lazy Object Storage Allocation.* The ability to allocate objects at different addresses on different processors suggests that a processor can delay the storage allocation for an object until that object is first accessed by that processor. We call this optimization *lazy object storage allocation*.

N-body simulations (e.g., Barnes-Hut and Water-Spatial [Woo et al. 1995]) illustrate the benefit of this optimization. Each processor typically accesses its own bodies, and a small number of "nearby" bodies on other processors. With global allocation of memory, the remote bodies are scattered in shared memory, causing a large memory footprint, many page faults, messages, and—in the case of TreadMarks—false sharing. In contrast, in DOSA, only the local bodies and the locally accessed remote bodies are allocated in local memory. As a result, the memory footprint is smaller, there are far fewer misses and messages, and false sharing is eliminated through the per-object mappings. Moreover, objects can be locally rearranged in memory, for instance, to improve cache locality or during garbage collection, without affecting the other processors. More generally, this optimization works for all applications that exhibit this kind of locality of access.

4.5.2 *Access Miss Handling and Read Aggregation.* When a processor faults on an object smaller than a page, it uses the list of objects in the same page (see Section 4.2) to find all of the invalid objects residing in that page. It sends out concurrent object fetch messages for all these objects to the processors recorded as the last writers of these objects. We refer to this optimization as *read aggregation*.

By doing so, we aggregate the requests for all invalid objects in the same page. This approach performs better than simply fetching one faulted object at a time. There are two fundamental reasons for this phenomenon.

1. If there is some locality in the objects accessed by a processor, then it is likely that the objects allocated in the same page are going to be accessed closely together in time. Here, again, the lazy object storage allocation works to our advantage. It is true that some unnecessary data may be fetched, but the effect is minimal for the following reason.
2. The destination processors of different request messages process the messages in parallel and also generate the replies in parallel. In addition, a particular destination processor needs to handle only one message per page as opposed to one message per object on that page for which it has the up-to-date value. Thus the latency for updating all objects on the page is not much higher than updating a single object.

If an object is larger than a page, we fall back on a page-based approach. In other words, only the page that is necessary to satisfy the fault is fetched.

4.5.3 *Modification Detection and Write Aggregation.* On a write fault, we make a copy (a twin) of the page on which the fault occurred, and we make all read-only objects in the page read-write. At a (release) synchronization point, we compare the modified page with the twin to determine which objects have been changed, and hence for which objects write notices need to be generated.[1] After the (release) synchronization, the twin is deleted and the page is made read-only again.

This approach has better performance than the more straightforward approach, where only one object at a time is made read-write. The latter method generates a substantially larger number of write faults. If there is locality to the write access pattern, the cost of these write faults exceeds the cost of making the twin and performing the comparison. We refer to this optimization as *write aggregation*.

[1]The twin is used here for a different purpose than the twin in TreadMarks. Here it is simply used to generate write notices. In the TreadMarks multiple-writer protocol it is used to generate a diff, a runlength encoding of the changes to the page. Because we are using a single-writer protocol, there is no need for diffs.

4.5.4 *Write Notice Compression.* The write notices are in terms of objects. As a consequence, for applications with a large number of objects, the number of write notices can potentially be very large. DOSA employs a novel compression technique to reduce the number of write notices transmitted. Each time a processor creates a new interval, it traverses in reverse order the intervals that it has created before, searching for one that has a similar set of write notices. If such an approximate "match" is found, the encoding of the differences between the write notices of the new and the old interval are typically much smaller than the write notices themselves. In this case, only the differences are sent. Information about intervals on one processor is always received on another processor in the order of increasing interval numbers. Thus, when a processor receives interval information containing a difference of write notices, it must already have received the interval based on which that difference has been computed. It can then easily reconstruct the write notices of the new interval. To avoid having to save all the old intervals created by all other processors, we set a threshold on how many old intervals to save and thus how far to search back for a close match with an old interval. If no close match is found, the current set of new write notices is sent in the normal way.

4.5.5 *Eliminating Indirect References in Loops.* This optimization eliminates the cost of repeated indirection through the handle table when an array is accessed inside a loop. Assume a one-dimensional array a is accessed in a for loop. Using C-like pseudocode, the resulting memory accesses in DOSA can be expressed as follows.

```
for i
    a->handle[i] = ...;
```

This code sequence leads to twice as many memory accesses as a corresponding loop implemented without a handle table.

Observe also that the following transformation of the DOSA program is legal but not profitable.

```
p = a->handle;
for i
    p[i] = ...;
```

The problem with this transformation occurs when `a->handle` has been invalidated as a result of a previous synchronization. Before the loop, `p` contains an address in the invalid region, which causes a page fault on the first iteration of the loop. DOSA changes `a->handle` to its location in the read-write region, but this change is not reflected in `p`. As a result, the loop page faults on every iteration. We solve this problem by "touching" `a->handle[0]` before assigning it to `p`. In other words,

```
write_touch( a->handle[0] );
p = a->handle[0];
for i
    p[i] = ...;
```

Write-touching `a->handle[0]` outside the loop causes the write fault to occur there, and `a->handle` to point to the read-write mapping. Similarly, if the access is a read (i.e., appearing on the right-hand side of an assignment), we insert a read-touch before the loop to force an update of the handle to point to the read-only mapping.

This optimization is dependent on the lazy implementation of release consistency. Invalidations only arrive at synchronization points, never asynchronously; thus the cached references are never invalidated in a synchronization-free loop.

With suitable compiler support, this optimization can be automated. For example, recent progress on pointer analysis (e.g., Landi and Ryder [1992], Andersen [1994], and Das [2000]) has allowed for fairly accurate computation of the set of memory locations to which a pointer can potentially point. Such analysis can be used to prove that `a` and `a->handle` are invariant across the loop iterations, automating the above transformation.

4.6 Limitations

The triple mapping consumes virtual memory space and causes TLB (translation lookaside buffer) pressure. This is not a significant problem, however, for two reasons: with 64-bit architectures nearby, consumption of virtual memory space is unlikely to remain a problem; and, compared to page-based DSMs, the lazy object allocation in DOSA actually reduces virtual memory use and TLB pressure for applications in which each processor only accesses a subset of the objects. Each processor only needs to allocate memory for those objects that it accesses, thus effectively packing those objects into fewer virtual memory pages, and reducing the virtual memory footprint and resulting TLB pressure. In contrast, in page-based DSMs virtual memory has to be allocated for all objects on all processors, regardless of whether a processor accesses an object. Unfortunately, the hardware counters on the architecture that we are using do not monitor TLB performance [Intel Corporation 2001]. Thus we cannot provide quantitative information on this subject.

Some of the optimizations, in particular lazy object allocation, read aggregation, and write aggregation, seek to take advantage of locality in the application. If no such locality is present, then no benefits can be expected.

5. EVALUATION METHODOLOGY

We evaluate DOSA by comparing its performance to TreadMarks [Amza et al. 1996]. We choose TreadMarks as a basis for comparison because it is the most widely used VM-based distributed shared memory system and because DOSA is derived from the same code base as TreadMarks, thus avoiding performance differences due to coding. Briefly, TreadMarks uses a lazy implementation of the release consistency which reduces the amount of data and the number of messages transmitted compared to eager implementations [Amza et al. 1996]. To address the false sharing problem facing VM-based DSM systems, TreadMarks uses a multiple-writer protocol that allows multiple concurrent writers to modify a page [Carter et al. 1991]. Unfortunately, none of the instrumentation-based

DSM systems aimed at fine-grained sharing is generally available, so a direct comparison to such a system cannot be made.

In general, our performance evaluation seeks to provide evidence for our claim that DOSA provides efficient support for both fine- and coarse-grained sharing, and interacts well with garbage collection. As a result, our performance evaluation considers both fine- and coarse-grained applications as well as garbage-collected applications. In more detail, we seek to substantiate the following claims.

1. The performance of fine-grained applications is considerably better than in TreadMarks.
2. The performance of coarse-grained applications is nearly as good as in TreadMarks. DOSA slightly underperforms TreadMarks because of the overhead of indirection from using handles. Because the performance of such applications is already good in TreadMarks, we consider this an acceptable performance penalty.
3. The performance of garbage-collected applications is considerably better than in TreadMarks.

5.1 Comparison with TreadMarks for Fine- and Coarse-Grained Applications

A difficulty arises in making the comparison with TreadMarks. Ideally, we would like to make these comparisons by simply taking a number of applications in a safe language, and comparing their performance when running on TreadMarks with their performance on DOSA.

For a variety of reasons, the most appealing programming language for this purpose is Java. Unfortunately, commonly available implementations of Java are interpreted and run on slow Java virtual machines. This would render our experiments largely meaningless, because inefficiencies in the Java implementations and virtual machines would dwarf differences between TreadMarks and DOSA. Perhaps more important, we expect efficient compiled versions of Java to become available soon, and we expect that those would be used in preference to the current implementations, quickly making our results obsolete. Finally, the performance of these Java applications would be quite inferior to published results for conventional programming languages.

We have therefore chosen to carry out the following experiments. For the first two comparisons, we have taken existing C applications and rewritten them to follow the model of a handle-based implementation. In other words, a handle table is introduced, and all pointers are redirected through the handle table. This approach represents the results that could be achieved by a language or compilation environment that is compatible with our approach for maintaining consistency, but otherwise exhibits no compilation or execution differences with the conventional TreadMarks execution environment. In other words, these experiments isolate the benefits and the drawbacks of our consistency maintenance methods from other aspects of the compilation and execution process. It also allows us to assess the overhead of the extra indirection on single-processor execution times. The optimizations discussed in Section 4.5.5 have been implemented by hand in both the TreadMarks and the DOSA programs. We report results with and without these optimizations present.

5.2 Comparison with TreadMarks for Garbage-Collected Applications

We have implemented a distributed garbage collector on both TreadMarks and DOSA that is representative of the state of the art. Distributed garbage collectors are naturally divided into an interprocessor algorithm, which tracks crossprocessor references and an intraprocessor algorithm, which performs the traversal on each processor and reclaims the unused memory.

To provide a fair comparison, we use the same interprocessor algorithm for both TreadMarks and DOSA. In particular, we use a *weighted reference counting* [Bevan 1987; Thomas 1981; Watson and Watson 1987]. To implement weighted reference counting transparently, we check incoming and outgoing messages for references. These references are recorded in an import table and an export table, respectively.

We also use the same intraprocessor algorithm in TreadMarks and DOSA, namely, a generational copying collector. The generational copying collectors have two generations. Following the suggestion by Wilson and Moher [1989], objects in the younger generation advance to the older generation in every other garbage collection. Like Tarditi and Diwan's [1996] collector, the old generation is included in a garbage collection if the size of the free space falls below 20% of the total heap size. Due to the existence of handles and the single-writer protocol in DOSA, the implementation of the collectors for TreadMarks and DOSA differs significantly. We next describe the differences in the two implementations.

In TreadMarks, when a processor wants to start a garbage collection, it initiates a barrier-like operation that suspends the DSM system. Only the last writer of a page is allowed to copy objects within that page. After the barrier, the last writer of every page is known to every processor. In the case of multiple writers to a page, an arbitration algorithm in the barrier designates a single processor as the last writer. Each processor starts a depth-first traversal from the "root" references and the exported references. An object is copied and scanned only if the node is its last writer. When an object is copied, a forwarding pointer to its new location is written into its old address. A reference to the moved object is updated only if the processor is also the last writer of the page that contains this reference. After the traversal, imported references still pointing to the old address ranges (the fromspace) are removed from the import table and sent back to the object's creator. The fromspace cannot, however, be immediately reclaimed. A processor instead sets the protection of the fromspace to no-access. When a remote processor follows a stale pointer and accesses a page in the fromspace, it fetches the up-to-date copy of the page, reads the forwarding pointer, and updates the reference that has caused the fault to the new address. The fault handler does not change the protection of the page so that other stale references to moved objects are caught. The fromspace is reclaimed when all stale references have been updated on all nodes, that is, when there are no more export or import references to the fromspace.

The copying collector for DOSA is much simpler. Each processor can start a garbage collection asynchronously because the last writer of an object is always known locally. Like the copying collector for TreadMarks, a processor starts a depth-first traversal from the "root" references and the exported references, and only copies and scans an object if it is the last writer of the object. Forwarding pointers are unnecessary because the only reference that needs to be updated is the address in the object's handle. After the traversal, imported references still pointing to the old address ranges (the fromspace) are removed from the import table and sent back to the object's creator. The fromspace can immediately be reclaimed and reused.

On DOSA, the garbage collector on each processor is also responsible for reclaiming unused handles, that is, OIDs, that it owns (see Section 4.2). The locally owned handles are managed using Baker's [1991] noncopying collector. Two pointer fields are added to each handle: one to chain the free handles in a free list, and one to chain the allocated handles in an allocated list. During garbage collection, live handles are moved from the old allocated list to a new allocated list. This occurs when an object (whose handle is owned by the processor) is copied from the old space to the new space. At the end of garbage collection, any handles remaining in the old allocated list are unused, so the old allocated list is appended to the free list. When changing the membership of handles between free and allocated lists, the handles themselves are not moved and thus the OIDs are not changed.

5.3 Experimental Environment

Our experimental platform is a switched, full-duplex 100 Mbps Ethernet network of thirty-two 300 MHz Pentium II-based computers. Each computer has a 512 Kbyte secondary cache and 256 Mbytes of memory. All of the machines are running FreeBSD 2.2.6 and communicating through UDP sockets. On this platform, the roundtrip latency for a 1 byte message is 126 microseconds. The time to acquire a lock varies from 178 to 272 microseconds. The time for an 32 processor barrier is 1333 microseconds. For TreadMarks, the time to obtain a diff, a runlength encoding of the changes to a page [Amza et al. 1996], varies from 313 to 1544 microseconds, depending on the size of the diff. The time to obtain a full page is 1308 microseconds.

5.4 Applications

Our choice of applications follows immediately from the goals of our performance evaluation. We include a number of fine-grained, coarse-grained, and garbage-collected applications. An application is said to be coarse-grained if a single processor typically accesses a large portion or all of a page, and is said to be fine-grained otherwise. An application may be coarse-grained because its basic objects are large relative to the size of a page, or because it is accessing a number of (small) objects that are contiguous in memory.

We use three fine-grained applications: Barnes-Hut and Water-Spatial from the SPLASH-2 [Woo et al. 1995] benchmark suite, and Gauss distributed with TreadMarks. Barnes-Hut is an N-body simulation, Water-Spatial is a molecular

Table I. Applications, Input Data Sets, and Sequential Execution Time

Application		Problem Size	Time (sec.) Original	Time (sec.) Handle
Fine-grained				
Barnes-Hut	Small	32K bodies, 3 steps	58.68	60.84
	Large	131K bodies, 3 steps	270.34	284.43
Water-Spatial	Small	4K mols, 9 steps	89.63	89.80
	Large	32K mols, 2 steps	158.57	160.39
Gauss	Small	512×512	2.20	2.22
Coarse-grained				
Red-Black SOR	Small	3070×2047, 20 steps	21.12	21.13
	Large	4094×2047, 20 steps	27.57	28.05
Water-N-Squared	Small	1728 mols, 2 steps	71.59	73.83
	Large	2744 mols, 2 steps	190.63	193.50
Gauss	Large	$1k \times 1k$	18.76	18.97

dynamics simulation optimized for spatial locality. Gauss implements Gaussian Elimination with partial pivoting on linear equations stored in a two-dimensional shared array. For Gauss, the computation on the rows is distributed across the processors in a round-robin fashion. Thus, if the size of a row is smaller than the size of a VM page, the access pattern is fine-grained. We denote this case as Gauss/small. Overall, the object sizes for these applications are 104 bytes for Barnes-Hut, 680 bytes for Water-Spatial, and 2 kilobytes for Gauss/small. In all cases, a single processor does not access contiguous objects.

We use three coarse-grained applications: SOR and Gauss/large distributed with TreadMarks and Water-N-Squared from the SPLASH [Singh et al. 1992] benchmark suite. SOR performs red–black successive overrelaxation on a 2-D grid. Gauss/large is identical to Gauss/small except that its row size is the size of a VM page. Water-N-Squared is a molecular dynamics simulation. The object sizes for SOR and Gauss/large are 8 and 4 kilobytes, respectively. For Water-N-Squared it is 680 bytes, but in this case a processor accesses a number of contiguous objects, thus making access coarse-grained.

For each of these applications, Table I lists the problem size and the sequential execution times. We use two problem sizes for each application, denoted large and small. The sequential execution times are obtained by removing all TreadMarks or DOSA calls from the applications, and using the optimization described in Section 4.5.5. Table I also includes execution times with and without handles. These timings show that the overhead of the extra level of dereferencing in the handle-based versions of the applications is never more than 5.2% on one processor for any of these applications. The sequential execution times without handles are used as the basis for computing the speedups reported later in the article.

To exercise the distributed garbage collector, we use three programs that have been modified to perform garbage collection. The first program, OO7, is a modified version of the OO7 object-oriented database benchmark [Carey et al. 1994]. The next program, Game, performs a game-tree search for a game called Othello. The last program, MIP, solves a Mixed Integer Programming problem [Bixby et al. 1999].

The OO7 benchmark is designed to match the characteristics of many CAD/CAM/CASE applications. Its database contains a tree of assembly objects, with leaves pointing to three composite parts chosen randomly from among 500 objects. Each composite part contains a graph of atomic parts linked by connection objects. Each atomic part has three outgoing connections.

Ordinarily, OO7 does not release memory. Thus there would be nothing for a garbage collector to do. Our modified version of OO7 creates garbage by replacing rather than updating objects when the database changes. After the new object with the updated data is installed in the database, the old object becomes eligible for collection.

The OO7 benchmark defines several database traversals [Carey et al. 1994]. For our experiments, we use a mixed sequence of T1, T2a, and T2b traversals. T1 performs a depth-first traversal of the entire object hierarchy. T2a and T2b are identical to T1, except that T2a replaces the root atomic part of each graph, and T2b replaces all the atomic parts.

The Game program for Othello runs for several game steps. In each step, a master processor takes the current game board as the root of the game tree, and expands the tree for a predetermined number of levels. Each node in the tree has a back pointer to its parent node. After the master finishes expanding the tree, it puts the leaf nodes in a task queue. Then each processor repeatedly takes a task from the queue, computes the result, and writes the result into all ancestors of the task node, including the root node. The writes to the ancestor nodes are synchronized by a lock. At the end of each step, the master makes the best move, and the game tree is discarded.

We run the program for 20 steps. The size of the game tree generated in each step is around 256Kbytes. Each processor also allocates a large number of private objects.

MIP solves the Mixed Integer Programming problem, a form of linear programming in which some of the variables are restricted to take on only integer values. It uses branch-and-bound to find the optimal solution to the problem. Nodes in the search space are kept in a doubly linked task queue. Each processor takes a node from this queue, performs its computation, perhaps generating new task nodes, and puts these new nodes back into the queue. For each task node, the computation involves "relaxing" the integer restrictions on the variables and solving the corresponding linear programming problem to determine whether a better solution than the current best solution is possible below that node. This procedure is repeated until the solution is found. This program allocates about 32K objects. All of the objects are shared.

Table II lists the sequential execution times for OO7, Game, and MIP running with the garbage collector on TreadMarks and DOSA. Overall, DOSA underperforms TreadMarks by less than 3.5% due to handle dereference cost and GC overhead. Table II also lists the time spent in the memory allocator and the garbage collector. Compared to TreadMarks, DOSA spends 0.9%, 3.6%, and 10% more time in the garbage collector for OO7, Game, and MIP, respectively. This extra overhead results from the fact that whenever an object is created, deleted, or moved, DOSA has to update the handle table entry.

Table II. Total Execution Time and Allocation and GC Time on 1 Processor for OO7, Game, and MIP with Garbage Collection

Application	Version	Time (sec.) Total	Time (sec.) Alloc. and GC
OO7	TreadMarks	185	10.9
	DOSA	191	11.0
Game	TreadMarks	286	21.7
	DOSA	296	22.5
MIP	TreadMarks	583	0.20
	DOSA	600	0.22

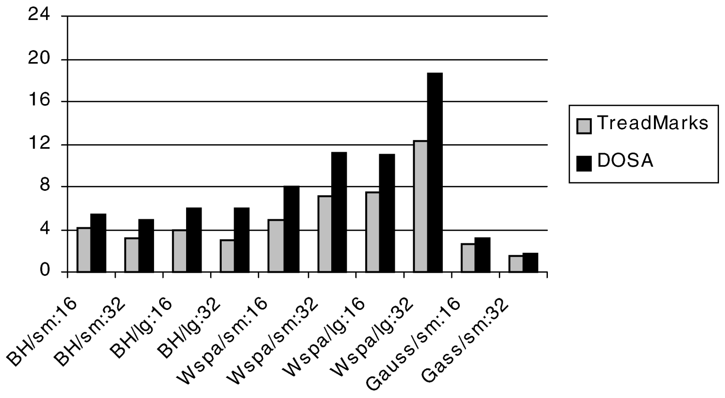

Fig. 5. Speedup comparison between TreadMarks and DOSA for fine-grained applications.

6. OVERALL RESULTS

6.1 Fine-Grained Applications

Figure 5 shows the speedup comparison between TreadMarks and DOSA for Barnes-Hut and Water-Spatial with small and large problem sizes, and Gauss with the small problem size on 16 and 32 processors.

We derive the following conclusions from these data. First, even for a small number of processors, the benefits of the handle-based implementation outweigh the cost of the extra indirection. For Barnes-Hut with 32K and 128K bodies, DOSA outperforms TreadMarks by 29% and 52%, respectively, on 16 processors. For Water-Spatial with 4K and 32K molecules, DOSA outperforms TreadMarks by 62% and 47%, respectively, on 16 processors. For Gauss/sm, DOSA outperforms TreadMarks by 23.3%. Second, as the number of processors increases, the benefits of the handle-based implementation grow. For Barnes-Hut with 128K bodies, DOSA outperforms TreadMarks by 52% on 16 processors and 98% on 32 processors. For Water-Spatial with 32K molecules, DOSA outperforms TreadMarks by 47% on 16 processors and 51% on 32 processors. For Gauss/sm, DOSA outperforms TreadMarks by 23.3% on 16 processors and by 25.6% on 32 processors. Third, if the amount of false sharing under TreadMarks

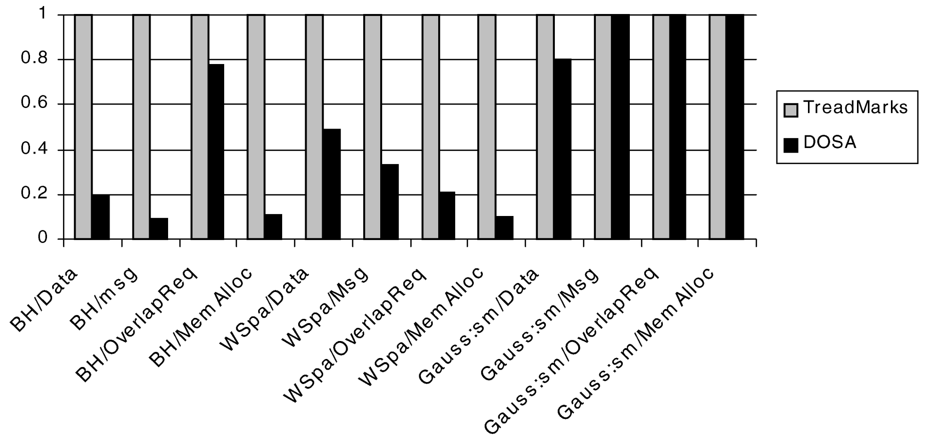

Fig. 6. Statistics for TreadMarks and DOSA on 32 processors for fine-grained applications with large data sizes, normalized to TreadMarks measurements. The statistics include the amount of data (Data), the number of messages (Msg), the number of message rounds where a message round includes all messages sent and received in parallel (OverlapReq), and the average amount of memory allocated per processor (MemAlloc).

Table III. Detailed Statistics for TreadMarks and DOSA on 32 Processors for Fine-Grained Applications, Barnes-Hut, Water-Spatial, and Gauss/sm[a]

	Barnes-Hut/sm		Barnes-Hut/lg	
Application	Tmk	DOSA	Tmk	DOSA
Time	18.07	12.06	89.07	45.98
Data (MB)	315.3	82.6	1307	246
Messages	2549648	307223	10994350	1027932
Overlapped requests	108225	98896	439463	341303
Memory alloc. (MB)	7.36	1.05	29.4	3.35

	Water-Spatial/sm		Water-Spatial/lg		Gauss/sm	
Application	Tmk	DOSA	Tmk	DOSA	Tmk	DOSA
Time	12.52	8.04	12.89	8.52	1.57	1.25
Data (MB)	475.1	262.6	342.1	166.8	44.7	35.8
Messages	617793	188687	330737	109560	94265	95135
Overlapped requests	193692	66937	202170	41491	31248	31683
Memory alloc. (MB)	3.15	0.61	25.2	2.64	1.06	1.06

[a]The statistics include the amount of data (Data), the number of messages (Msg), the number of message rounds where a message round includes all messages sent and received in parallel (OverlapReq), and the average amount of memory allocated per processor (MemAlloc).

decreases as the problem size increases, as in Water-Spatial, then DOSA's advantage over TreadMarks decreases for larger problem sizes. If, on the other hand, the amount of false sharing under TreadMarks increases as the problem size increases, as in Barnes-Hut, then DOSA's advantage over TreadMarks increases for larger problem sizes.

The reasons for DOSA's clear dominance over TreadMarks can be seen in Figure 6, which shows the normalized statistics from the execution of Barnes-Hut/lg, Water-Spatial/lg, and Gauss/sm on 32 processors, and also in Table III, which presents detailed statistics from these executions. The figure and the table show the amount of data communicated, the number of messages

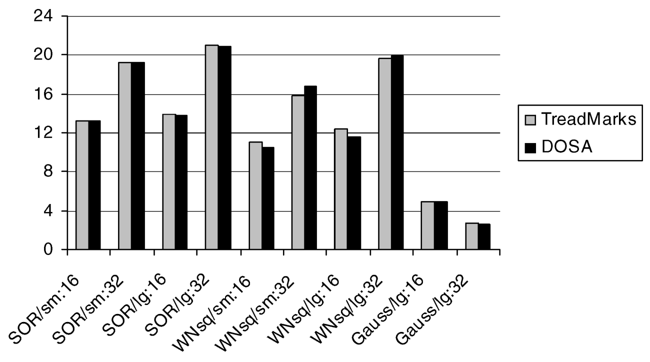

Fig. 7. Speedup comparison between TreadMarks and DOSA for coarse-grained applications.

exchanged, the number of overlapped data requests,[2] and the average amount of shared data allocated on each processor. Specifically, we see a substantial reduction in the amount of data sent for DOSA, as a result of the reduction in false sharing. Furthermore, the number of messages is reduced by a factor of 11 for Barnes-Hut/lg and 3 for Water-Spatial/lg. More importantly, the number of overlapped data requests is reduced by a factor of 1.3 for Barnes-Hut/lg and 4.9 for Water-Spatial/lg. Finally, the benefits of lazy object allocation for these applications are quite clear: the memory footprint of DOSA is considerably smaller than that of TreadMarks. For Gauss/sm, the number of messages, the number of overlapped data requests, and the memory footprint size are the same for DOSA and TreadMarks, as there is no write-write false sharing of shared pages among the processors. The better performance of DOSA over TreadMarks stems solely from a reduction in the amount of data transmitted, as a result of lazy object allocation. In Gauss/sm a row occupies half a page. Lazy object allocation avoids transmitting the other half of the page in which the pivot row is allocated. For a more detailed explanation, see Section 7.1.

6.2 Coarse-Grained Applications

Figure 7 shows the speedup comparison between TreadMarks and DOSA for SOR and Water-N-Squared with small and large problem sizes, and for Gauss with the large problem size, running on 16 and 32 processors. The results show that the performance of these coarse-grained applications in DOSA is within 6% of the performance achieved with TreadMarks.

Figure 8 shows normalized statistics from the execution of these applications on 32 processors for SOR and Water-N-Squared with large problem sizes and Gauss/lg. The detailed statistics are listed in Table IV. For SOR and Gauss/lg, where the data unit (i.e., a row) is a multiple of a page, the detailed statistics for the two systems are almost identical. DOSA performs almost as well as TreadMarks for SOR and is 3.5% slower than TreadMarks for Gauss/lg. These

[2]All concurrent message exchanges for updating a page in TreadMarks or all concurrent message exchanges for updating invalid objects in a page in DOSA are counted as one overlapped data request. These messages are processed and their replies are generated in parallel, so they largely overlap each other.

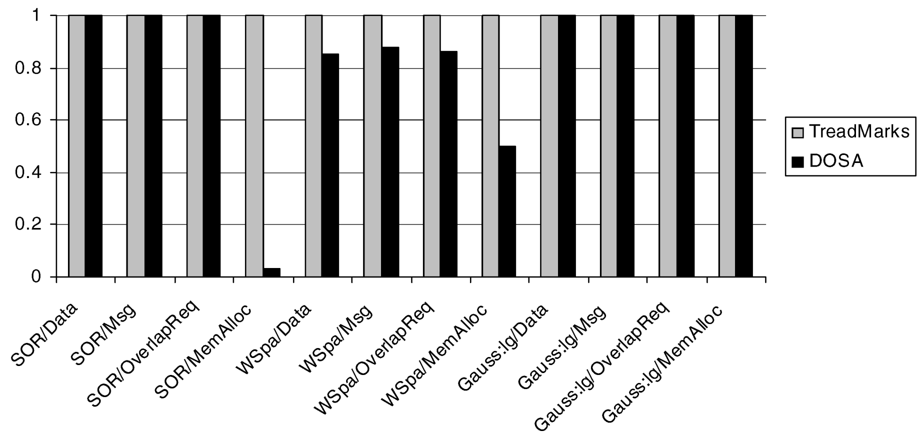

Fig. 8. Statistics for TreadMarks and DOSA on 32 processors for coarse-grained applications with large data sizes, normalized to TreadMarks measurements. The statistics include the amount of data (Data), the number of messages (Msg), the number of message rounds where a message round includes all messages sent and received in parallel (OverlapReq), and the average amount of memory allocated per processor (MemAlloc).

Table IV. Detailed Statistics for TreadMarks and DOSA on 32 Processors for Coarse-Grained Applications SOR, Water-N-Squared, and Gauss/lg[a]

	SOR/sm		SOR/lg	
Application	Tmk	DOSA	Tmk	DOSA
Time	1.10	1.10	1.31	1.32
Data (MB)	23.6	23.6	23.6	23.6
Messages	12564	12564	12440	12440
Overlapped requests	4962	4962	4962	4962
Memory alloc. (MB)	50.3	1.64	67.1	2.16

	Water-N-Squared/sm		Water-N-Squared/lg		Gauss/lg	
Application	Tmk	DOSA	Tmk	DOSA	Tmk	DOSA
Time	4.52	4.27	9.74	9.58	6.77	7.01
Data (MB)	134.1	114.0	212.4	181.4	134.3	136.7
Messages	77075	63742	114322	101098	189500	190368
Overlapped requests	33033	28032	51816	44758	62992	63427
Memory alloc. (MB)	1.58	0.66	2.10	1.04	4.20	4.20

[a]The statistics include the amount of data (Data), the number of messages (Msg), the number of message rounds where a message round includes all messages sent and received in parallel (OverlapReq), and the average amount of memory allocated per processor (MemAlloc).

differences are due to the indirection through the handle table. For Water-N-Squared/lg, DOSA underperforms TreadMarks by 6.7% on 16 processors, but surprisingly outperforms TreadMarks by 1.7% on 32 processors. This is due to the difference between the multiple-writer protocol used in TreadMarks and the single-writer protocol used in DOSA. The migratory data access in Water-N-Squared/lg results in diff accumulation [Lu et al. 1997] in a multiple-writer protocol implementation, which causes TreadMarks to send more and more accumulated diffs as we increase the number of processors. On 16 processors, TreadMarks sends 15% less data than DOSA, and as a result outperforms DOSA by 6.7%. On 32 processors, as diff accumulations become more severe,

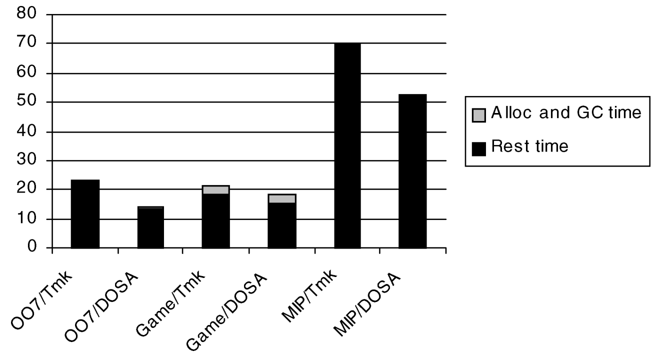

Fig. 9. Time breakdown (in seconds) for the garbage-collected applications, OO7 on 16 processors, and Game and MIP on 32 processors, on TreadMarks and DOSA.

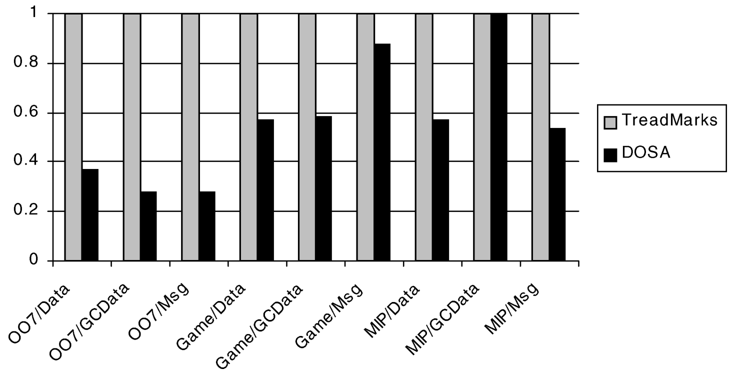

Fig. 10. Statistics for garbage-collected applications, OO7 on 16 processors, and Game and MIP on 32 processors, on TreadMarks and DOSA, normalized to TreadMarks measurements. The statistics include the total amount of data transmitted (Data), the amount of GC data transmitted (GC Data), and the number of messages (Msg). GC data consist of the weights transferred and the nack data, which list the references that have been removed from a processor and their weights.

TreadMarks sends 17% more data than DOSA and as a result slightly underperforms DOSA. Finally, because in Water-N-Squared each processor only accesses half of the molecules due to the use of symmetry, the memory footprint in DOSA is about half of that in TreadMarks.

6.3 Garbage-Collected Applications

Figures 9 and 10 show the execution statistics for OO7 running on 16 processors and Game and MIP running on 32 processors on TreadMarks and DOSA using the generational copying collector. The detailed statistics are listed in Table V. We do not present results for OO7 on 32 processors because the total data size, which increases linearly with the number of processors, is so large that it causes paging on 32 processors.

On 16 processors, OO7 performs almost 65% better on DOSA than on TreadMarks. On 32 processors, Game and MIP perform 19% and 33% better running on DOSA than running on TreadMarks. Table V shows that for all three

Table V. Detailed Statistics for TreadMarks and DOSA for the Garbage-Collected Applications[a]

	OO7		Game		MIP	
Tree	Tmk	DOSA	Tmk	DOSA	Tmk	DOSA
Time	23.4	14.2	21.5	18.0	70.3	52.7
Alloc and GC time	0.70	0.70	3.04	3.04	0.20	0.21
Data (MB)	48.4	17.9	60.5	34.5	203	116
GC Data (MB)	9.8	2.7	9.45	5.51	2.26	2.26
Messages	427.8K	117.4K	79.7K	69.8K	228K	122K

[a]OO7 is run on 16 processors, and Game and MIP are run on 32 processors. The statistics include the total amount of data transmitted (Data), the amount of GC data transmitted (GC Data), and the number of messages (Msg). GC data consist of the weights transferred and the nack data, which list the references that have been removed from a processor and their weights.

programs the time spent in the memory management code performing allocation and garbage collection is almost the same for TreadMarks and DOSA. The effects of the interaction between the garbage collector and DOSA or TreadMarks appear during the execution of the application code. The main cause for the large performance improvement in DOSA is reduced communication, as shown in Figure 10.

The extra communication on TreadMarks is primarily a side effect of garbage collection. On TreadMarks, when a processor copies an object during garbage collection, this operation is indistinguishable from an ordinary write by the application. Consequently, when another processor accesses the object after garbage collection, the object is transmitted, even though its content has not been changed by the garbage collector's copy. In fact, the requesting processor may have an up-to-date copy of the object in its memory, but at a new virtual address. In contrast, on DOSA, when the garbage collector copies an object, it simply updates its handle table entry, which is local information that never propagates to other processors.

7. EFFECTS OF THE VARIOUS OPTIMIZATIONS

To achieve the results described in the previous section, various optimizations are used in DOSA. These optimizations include lazy object allocation (Section 4.5.1), read aggregation (Section 4.5.2), write aggregation (Section 4.5.3), write notices reduction (Section 4.5.4), and removing indirect references from loops (Section 4.5.5). To see what effect each optimization has individually, we perform the following experiments. For each of the optimizations, we compare the performance of DOSA without that optimization to the fully optimized system.

Not all optimizations benefit all applications. The first four optimizations have no effect on SOR and Gauss/lg. The objects in these two applications are larger than a page. Read aggregation, write aggregation, and write notice reduction are performed on a per-page basis, and therefore produce no improvement, and lazy object allocation does not reduce false sharing. The fifth optimization produces noticeable improvements only for SOR and Gauss, because these two applications perform many indirect accesses that can be eliminated. For

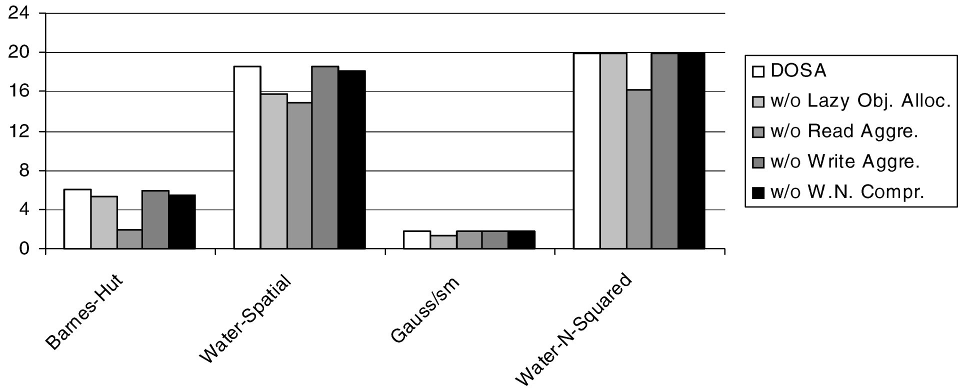

Fig. 11. Speedup comparison on 32 processors between DOSA and DOSA without lazy object allocation, read aggregation, write aggregation, and write notice reduction for Barnes-Hut/lg, Water-Spatial/lg, Gauss/sm, and Water-N-squared/lg. No results are shown for SOR and Gauss/lg, since these applications are not affected by these optimizations.

the other applications, floating point computations dominate the overhead of indirection.

For a particular optimization and a particular application, we present results only if the optimization benefits the application. Figure 11 shows the speedups for each of the experiments for the first four optimizations for Barnes-Hut, Water-Spatial, Gauss/sm, and Water-N-Squared. Table VI provides further detail on the effects of the optimizations for these four applications, comparing the number of messages, the number of overlapped data requests, the amount of data, the memory footprint, the number of write notices, and the number of write page faults, all with and without the optimizations. Table VII provides execution times for SOR/lg and Gauss/lg on 1 and 32 processors for DOSA and TreadMarks, with and without the optimization that removes indirect references from loops.

7.1 Lazy Object Allocation

Table VI shows that lazy object allocation produces improvements in execution time of 13% for Barnes-Hut, 18% for Water-Spatial, and 30% for Gauss/sm, compared to a version of DOSA without lazy object allocation. Lazy object allocation shows no improvement for Water-N-Squared.

Lazy object allocation significantly benefits irregular applications that exhibit spatial locality of reference in their *physical domain*. For example, even though the bodies in Barnes-Hut and the molecules in Water-Spatial are input or generated in random order, in the parallel algorithms each processor only updates bodies or molecules corresponding to a contiguous physical subdomain. Furthermore, inter-subdomain data references only happen on the boundary of each subdomain. As described in Section 4.5.1, for such applications, lazy object allocation only allocates memory for objects on a processor that are accessed by that processor. Therefore, a physical page contains mostly "useful" objects. With read aggregation, these objects are updated in a single round of overlapped

Table VI. Statistics from the Execution on 32 Processors for DOSA and DOSA Without Lazy Object Allocation, Read Aggregation, Write Aggregation, and Write Notice Reduction for Barnes-Hut/lg, Water-Spatial/lg, Gauss/sm, and Water-N-Squared/lg[a]

Application		DOSA	w/o Lazy Obj. Alloc.	w/o Read Aggre.	w/o Write Aggre.	w/o W.N. Reduc.
Barnes-Hut (lg)	Time (sec.)	45.07	50.90	139.88	46.05	49.56
	Data (MB)	245.8	251.7	124.7	246.2	277.4
	Write notices (M)	6.42	6.42	6.42	6.42	35.4
	Messages	1027903	1612276	2254374	1027944	1027991
	Overlapped reqs.	341303	428537	1123734	341303	341303
	Mem. alloc. (MB)	3.35	23.2	3.35	3.35	3.35
	Write faults	27586	163865	28248	582533	27728
Water-Spatial (lg)	Time (sec.)	8.52	10.07	10.54	8.54	8.75
	Data (MB)	166.8	167.2	166.0	166.7	169.4
	Write notices (M)	0.74	0.74	0.74	0.74	3.30
	Messages	109560	183965	478674	109560	109558
	Overlapped reqs.	41486	72664	238341	41491	41490
	Mem. alloc. (MB)	2.64	22.5	2.64	2.64	2.64
	Write faults	20989	35277	20777	110864	21062
Gauss (sm)	Time (sec.)	1.25	1.62	1.25	1.25	1.25
	Data (MB)	35.8	53.8	35.8	35.8	35.8
	Write notices (M)	0.10	0.10	0.10	0.10	0.10
	Messages	95135	95135	95135	95135	95135
	Overlapped reqs.	31683	31683	31683	31683	31683
	Mem. alloc.(MB)	1.06	1.06	1.06	1.06	1.06
	Write faults	768	768	768	768	768
Water-N-Squared (lg)	Time (sec.)	9.58	9.58	11.74	9.58	9.58
	Data (MB)	181.3	181.8	183.2	181.4	181.6
	Write notices (M)	0.81	0.81	0.81	0.81	0.97
	Messages	101098	101228	530783	101116	101108
	Overlapped reqs.	44758	44874	261669	44770	44757
	Mem. alloc. (MB)	1.04	1.89	1.04	1.04	1.04
	Write faults	16953	16978	87498	96589	16968

[a]The statistics include the execution time (Time), the amount of data (Data), the number of messages (Msg), the number of message rounds where a message round includes all messages sent and received in parallel (OverlapReq), and the average amount of memory allocated per processor (MemAlloc). No results are shown for SOR and Gauss/lg, since these applications are not affected by these optimizations.

Table VII. Execution Time (sec.) for SOR and Gauss/lg With and Without Removing Indirect Accesses on 1 and 32 Processors

Application	No. Procs.	Tmk/Opt.	Tmk/No-Opt.	DOSA/Opt.	DOSA/No-Opt.
SOR/lg	1	27.57	33.19	28.05	47.47
	32	1.32	1.54	1.31	1.84
Gauss/lg	1	18.76	23.64	18.97	33.58
	32	6.77	8.07	7.01	10.02

messages when faulting on the first object. In contrast, without lazy object aggregation, objects are allocated on all processors in the same order and at the same virtual address. Thus the order of the objects in memory reflects the access pattern of the initialization which may differ from that of the computation. In other words, objects accessed by a specific processor may be scattered in many more pages than with lazy object allocation. As a result, the memory footprint is much larger, many more faults occur, many more rounds of messages are required to make all the pages consistent, and more data are sent because all modified objects in the same page are sent. The statistics in Table VI quantify all these improvements.

As already explained in Section 6.1, Gauss/sm benefits from lazy object allocation because it reduces the extra data sent as a result of false sharing in a page containing a pivot row. For Gauss/sm, two rows fit in one VM page. With lazy object allocation, when the first row in a page becomes the pivot row, only that row is allocated and fetched. When the second row in that page becomes the pivot row, only that second row is allocated and updated, inasmuch as the first row is still valid. Therefore, remote pivot rows are always fetched once. Without lazy object allocation, when updating the first row on a page to become the pivot row, read aggregation causes both rows in that page to be updated, because the second row has been modified by the owning processor during some previous iterations. The update of the second row at this point is useless, as it needs to be updated again when it becomes the pivot row. This effect is clearly visible in the statistics for Gauss/sm in Table VI: the amount of data transmitted increases by 50%.

Lazy object allocation has no impact on Water-N-Squared because molecules are allocated in a 1-D array, and each processor always accesses the same contiguous segment, consisting of half of the array elements, in fixed increasing order. Table VI shows that the memory footprint is halved as a result of lazy object allocation, but all the other statistics remain the same.

7.2 Read Aggregation

Read aggregation produces the biggest performance improvements. Table VI shows execution time improvements of 310% for Barnes-Hut, 24% for Water-Spatial, and 22% for Water-N-squared. Read aggregation has no effect on Gauss/sm for the same reasons as explained in Section 7.1.

Read aggregation brings in all modified objects in a page at once, on the first fault on an object on that page. The potential gains come from fewer faults, fewer messages, and fewer rounds of overlapped requests. The potential problem with read aggregation is that DOSA may fetch more objects than necessary.

For Water-N-squared each processor accesses the same set of elements on each iteration. For Water-Spatial the set of accessed elements is almost the same on each iteration. In either case, spatial locality is good throughout the execution. Consequently, objects prefetched by read aggregation are typically used. In Barnes-Hut, the set of bodies accessed by a processor changes over time. In effect, when a body migrates from its old processor to its new one, it leaves behind a “hole” in the page that it used to occupy. When the old processor accesses any of the remaining objects in that page, read aggregation still updates the hole.

Table VI shows that without read aggregation, DOSA sends 2.2, 4.4, and 5.3 times more messages, and requires 3.3, 5.7, and 5.8 more overlapped requests for Barnes-Hut, Water-Spatial, and Water-N-Squared, respectively. For Water-Spatial and Water-N-Squared, the amount of data remains the same, with or without read aggregation. For Barnes-Hut, twice as much data are sent with read aggregation. Overall, the benefits outweigh the drawback for all three applications.

7.3 Write Aggregation

Improvements as a result of write aggregation are minor. Table VI shows a 2.2% improvement for Barnes-Hut and no noticeable improvement for the other applications. Table VI further shows that write aggregation reduces the number of page faults by factors of 21, 5.3, and 5.7 for Barnes-Hut, Water-Spatial, and Water-N-Squared, respectively. The impact on Water-Spatial and Water-N-Squared is marginal, as page faults constitute only a very small fraction of the execution time for these two applications. Write aggregation has no effect on Gauss/sm.

7.4 Write Notice Reduction

Write notice reduction is effective for Barnes-Hut and Water-Spatial. Table VI shows an improvement in execution time of 10% for Barnes-Hut and 3% for Water-Spatial. For Barnes-Hut, it reduces the amount of write notice data by a factor of 5.5, and for Water-Spatial by a factor of 4.5. This optimization has little effect on Water-N-Squared and Gauss/sm as they have relatively few write notices.

7.5 Removing Indirect Accesses in Loops

Table VII shows the execution times of SOR/lg and Gauss/lg, on 1 and 32 processors, for TreadMarks and DOSA, with and without the optimization for removing indirect accesses in loops. For these two applications, this optimization significantly improves performance. On a single processor, it improves the performance of the original array-based version of SOR/lg and Gauss/lg by 20% and 26%, respectively, and the handle-based version by 69% and 77%, respectively. On 32 processors, the improvements are 17% and 40% for the two versions of SOR, and 19% and 43% for the two versions of Gauss, respectively. This optimization has a similar effect on Gauss/sm as on Gauss/lg. It has little impact on Barnes-Hut, Water-Spatial, and Water-N-Squared, as these applications are dominated by floating-point computations.

8. RELATED WORK

The article already contains an extensive description of the coarse-grained TreadMarks system and a detailed comparison of DOSA and TreadMarks. In this section we discuss additional DSM systems that offer support for fine-grained sharing. We also compare DOSA with related work in the broader field of parallel and distributed object systems, persistent programming languages, and distributed persistent storage systems. Finally, we compare the garbage collection algorithm in DOSA to previous GC algorithms on DSMs.

8.1 Fine-Grained Distributed Shared Memory

8.1.1 *Using Instrumentation.* Instrumentation in support of fine-grained sharing has been used in Blizzard-S [Schoinas et al. 1994], Shasta [Scales et al. 1996], and Midway [Bershad et al. 1993]. Aggressive optimizations are required to reduce the potentially high overhead of run-time checking. In addition, these systems do not aggregate updates to shared blocks or take advantage of data locality to reduce memory usage and communication as DOSA does.

Dwarkadas et al. [1999] compared Cashmere, a coarse-grained system somewhat like TreadMarks, and Shasta, an instrumentation-based system, running on an identical platform—a cluster of four 4-way AlphaServers connected by a Memory Channel network. In general, Cashmere outperformed Shasta on coarse-grained applications (e.g., Water-N-Squared), and Shasta outperformed Cashmere on fine-grained applications (e.g., Barnes-Hut). Surprisingly, Cashmere's performance on the fine-grained application Water-Spatial equaled that of Shasta. They attributed this result to the run-time overhead of the in-line access checks in Shasta. In contrast, DOSA outperforms TreadMarks by 62% on the same application. We attribute this to lazy object allocation, which is not possible in Shasta, and read aggregation.

Jackal [Veldema et al. 2001b] is a fine-grained DSM system for Java that implements the Java memory model, which resembles but is subtly different from release consistency. Like Shasta, Jackal uses software inline checks for access detection to Java objects or arrays. Aggressive compiler and run-time optimizations are reported to be necessary to achieve reasonable performance [Veldema et al. 2001a]. Unfortunately, the system is not generally available, and therefore a direct comparison cannot be made.

8.1.2 *Using VM Protection.* Millipede [Itzkovitz and Schuster 1999] does not take advantage of language safety or of a handle-based programming model. Instead, it attempts to provide transparent support for fine-grained sharing using VM protection. This effectively requires that every object reside in a separate virtual page. Different virtual pages are mapped to a single physical page by offsetting the objects in their virtual pages such that they do not overlap in physical memory. Different protection attributes may be set on different virtual pages that are mapped to the same physical page, thereby achieving the same effect as DOSA, namely, per-object access and write detection. The Millipede approach is more costly than the DOSA approach in a variety of ways. It requires costly OS system calls (e.g., `mprotect`) to change page protections each time a page's protection needs to be changed. DOSA implements protection changes by user-level pointer switching in the handle table. Millipede requires one virtual memory mapping per object, whereas the DOSA method requires only three mappings per page, resulting in considerably less address space consumption and pressure on the TLB. Finally, it does not allow any aggregation optimizations, because each object must reside in a separate page.

The Region Trapping Library (RTL) [Brecht and Sandhu 1999] requires the programmer to define regions and identify all pointers into each region. It then allocates three different regions of memory with different protection attributes to perform per-object access and modification detection. All pointers declared to point into a region are changed when the region's protection changes. By virtue of using a handle-based programming model or by virtue of using a safe language in which the compiler can redirect all accesses through a handle table, DOSA effectively allows only a single pointer into an object, thereby avoiding the complication of having to declare all pointers into shared memory as belonging to a particular region. Furthermore, in the RTL implementation, the read memory region and the read-write memory region are backed by *different* physical memory regions. As result, modifications made in the read-write region must be copied to the read region, every time protection changes from read-write to read.

8.1.3 *Using Handles.* Freeh and Andrews [1996] use handles to reduce false sharing in a page-based DSM system. If false sharing occurs within a page, their system moves the objects causing the false sharing to another page. However, their system still maintains coherence at the page granularity. The handles are only used to facilitate the data movement, not to detect memory accesses at the object granularity.

8.1.4 *Shared Addresses Versus Shared Objects.* In all of the above systems, objects need to appear at the same virtual address on all processors. As a result, none can support lazy object allocation.

8.2 Parallel and Distributed Object-Oriented Languages and Systems

There has been a host of concurrent object-oriented languages and systems that aim to provide distributed or distributed shared objects.

Orca [Tanenbaum et al. 1992], Jade [Rinard and Lam 1998], COOL [Chandra et al. 1994], and SAM [Scales and Lam 1994] are parallel or distributed object-oriented languages. All of these systems differ from ours in that they present a new language or API to the programmer to express distributed sharing, whereas DOSA does not. DOSA aims to provide transparent object sharing for existing safe languages, such as Java. Furthermore, Orca, Jade, COOL, and SAM do not use VM-based mechanisms for object sharing.

Many recent systems add concurrency extensions to Java to support distributed shared objects. Examples include JavaParty [Philipsen and Zenger 1997], Javelin [Christiansen et al. 1997], Kan [James and Singh 2000], Aleph [Herlihy 1999], Shareholder [Harris and Sarkar 1998], and Parallel Java [Kale et al. 1997]. Again, these systems differ from DOSA in that they add either new language extensions or APIs to Java.

8.3 Persistent Languages and Persistent Object Systems

The need to deal with object identifiers as opposed to memory addresses has long faced implementors of persistent programming languages and persistent storage systems. In a persistent language, the persistent objects can outlive a single program execution. In a persistent storage system, the object space can potentially be larger than the virtual memory address space. In either case, persistent objects need to be stored using object identifiers.

The two classic ways of implementing persistent objects, *swizzling on discovery* [White and DeWitt 1992] and *address translation at page-fault time* [Wilson and Moher 1992], mirror the two classic ways of access detection in software DSMs, software inline checks and the VM protection mechanism. The tradeoffs between these two approaches mirror the trade-offs between fine-grained and coarse-grained DSMs: the faulting cost for the VM scheme favors large objects that contain many object references.

Similar trade-offs carry over to distributed persistent storage systems, in which servers provide persistent storage for information accessed by applications running at clients (see, e.g., Kachler and Krasner [1990], White and DeWitt [1992], Lamb et al. [1991], White and DeWitt [1994], and Castro et al. [1997]). These systems cache recently used information at client machines to provide low access latency and good scalability. Clients can use page caching [Lamb et al. 1991; White and DeWitt 1994], potentially leading to a mismatch between the cache unit size and the object size, or object caching [Kachler and Krasner 1990; White and DeWitt 1992], potentially leading to significant overhead in checking and maintaining the cache. An interesting in-between solution is hybrid adaptive caching (HAC) [Castro et al. 1997]. In this approach, the client dynamically partitions the cache between pages and objects based on the application behavior. Pages with good locality are cached as whole pages, whereas for pages with poor locality, only hot objects are retained and cold objects are evicted. To enable the hybrid adaptive caching, the object identifier is translated to a pointer to an entry in an indirection table, and the entry in turn points to the target object in memory.

8.4 Garbage Collection on DSM Systems

Le Sergent et al. [Le Sergent and Berthomieu 1992; Matthews and Le Sergent 1995] extend an incremental copying collector originally designed for a multiprocessor to a DSM system. The garbage collector requires a consistent image of the entire object graph, and therefore is very expensive. They do not report any performance measurements. Kordale et al. [1993] describe a garbage collector for DSM based on a mark-sweep technique. Like Le Sergent's collector, this algorithm also requires a consistent image of the entire object graph, and is therefore very expensive.

Ferreira and Shapiro [1994] are the first to point out that a garbage collector can be designed to tolerate memory inconsistency. Their algorithm allows the processors to collect independently. Their design depends on the entry consistency model [Bershad et al. 1993], which presents a single-writer interface. Address changes are propagated asynchronously and piggybacked in messages sent out by the application program. They evaluate the scalability of their design, but do not study the impact of the garbage collector on overall program performance. It is not straightforward to adapt their algorithms to DSM systems using other relaxed consistency protocols.

Yu and Cox [1996] discuss a conservative mark-sweep garbage collector for DSM systems. They show that the garbage collector can be very efficient, incurring low overheads during garbage collection. However, the poor spatial locality as a side effect of the mark-sweep collector can result in high communication cost, and is detrimental to the overall performance of some programs.

The shared object space abstraction in DOSA decouples an object's naming from its address in memory, making the intraprocessor garbage collector orthogonal to the DSM operations. Therefore, a processor is free to use any garbage collection algorithm without it having any negative effect on the performance of other processors. The shared object space abstraction not only eliminates the negative effect of the intraprocessor garbage collector and improves the overall program performance, it also simplifies the design of the intraprocessor garbage collectors.

9. CONCLUSIONS

We have presented a new run-time system, DOSA, that efficiently supports both fine- and coarse-grained object sharing in a distributed system. A handle table supports efficient per-object access and modification detection using VM protection mechanisms. In a safe language, a compiler can generate a handle-based implementation of a program, thereby making it transparent to the application programmer. The same benefits can also be obtained nontransparently if the programmer accesses all objects through a handle and refrains from pointer arithmetic on the handles. Like earlier systems designed for fine-grained sharing, DOSA improves the performance of fine-grained applications by eliminating false sharing. In contrast to these earlier systems, DOSA's VM-based approach and read aggregation enable it to match the performance of a page-based system for coarse-grained applications. Furthermore, its architecture permits optimizations, such as lazy object allocation, that are not possible in conventional fine- or coarse-grained DSM systems. Lazy object allocation transparently improves the locality of reference in many applications, improving their performance.

Our performance evaluation on a cluster of 32 Pentium II processors connected with a 100 Mbps Ethernet demonstrates that the new system performs comparably to TreadMarks for coarse-grained applications (SOR, Water-N-Squared, and Gauss/lg), and significantly outperforms TreadMarks for fine-grained applications (up to 98% for Barnes-Hut, 62% for Water-Spatial, and 25.6% for Gauss/sm) and garbage-collected applications (65% for OO7, 19% for Game, and 33% for MIP).

We have also presented a complete breakdown of the performance results. The optimizations of lazy object allocation and read aggregation are particularly significant to DOSA's performance. Without lazy object allocation, on 32 processors, Barnes-Hut runs 13% slower, Water-Spatial 18%, and Gauss/small 30%. Without read aggregation, on 32 processors, Barnes-Hut runs 310% slower, Water-Spatial 24%, and Water-N-Squared 22%.

ACKNOWLEDGMENTS

We thank the anonymous TOCS reviewers, whose comments have helped to improve the presentation and the content of this article. We also thank Gregory

Andrews, Rudi Eigenmann, Matthias Felleisen, Ralph Grishman, Allan Gottlieb, Rajiv Gupta, Antony Hosking, Liviu Iftode, Assaf Kfoury, Shriram Krishnamurthi, Stuart Kurtz, Srini Parthasarathy, Benjamin Pierce, Rick Snodgrass, and Stan Zdonik for their helpful comments and discussions on this work.

REFERENCES

Adl-Tabatabai, A., Cierniak, M., Lueh, G., Parakh, V. M., and Stichnoth, J. M. 1998. Fast effective code generation in a just-in-time Java compiler. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 280–290.

Adve, S. and Hill, M. 1990. Weak ordering: A new definition. In *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture*, 2–14.

Amza, C., Cox, A., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., and Zwaenepoel, W. 1996. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer 29*, 2 (Feb.), 18–28.

Amza, C., Cox, A., Rajamani, K., and Zwaenepoel, W. 1997. Trade-offs between false sharing and aggregation in software distributed shared memory. In *Proceedings of the Sixth Symposium on the Principles and Practice of Parallel Programming*, 90–99.

Andersen, L. 1994. Program analysis and specialization for the C programming language. PhD Thesis, DIKU University of Copenhagen.

Baker, H. 1991. The TreadMill: Realtime garbage collection without motion sickness. In *Proceedings of the OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems*.

Bershad, B., Zekauskas, M., and Sawdon, W. 1993. The Midway distributed shared memory system. In *Proceedings of the '93 CompCon Conference*, 528–537.

Bevan, D. I. 1987. Distributed garbage collection using reference counting. In *Parallel Architecture and Languages, Europe*. Lecture Notes in Computer Science, vol. 259, Springer-Verlag, Eindhoven, The Netherlands, 117–187.

Bixby, R., Cook, W., Cox, A., and Lee, E. 1999. Computational experience with parallel mixed integer programming in a distributed environment. *Ann. Oper. Res. 90*, 19–43.

Brecht, T. and Sandhu, H. 1999. The region trap library: Handling traps on application-defined regions of memory. In *Proceedings of the 1999 USENIX Annual Technical Conference.*

Budimlic, Z. and Kennedy, K. 1997. Optimizing Java: Theory and practice. *Concurr. Pract. Exper. 9*, 6, 445–463.

Budimlic, Z. and Kennedy, K. 1999. Prospects for scientific computing in polymorphic, object-oriented style. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*.

Burke, M. G., Choi, J.-D., Fink, S., Grove, D., Hind, M., Sarkar, V., Serrano, M. J., Sreedhar, V. C., Srinivasan, H., and Whaley, J. 1999. The Jalapeno dynamic optimizing compiler for Java. In *Proceedings of the ACM 1999 Java Grande Conference*, 129–141.

Carey, M., DeWitt, D., and Naughton, J. 1994. The OO7 benchmark. Tech. Rep., University of Wisconsin-Madison, July.

Carter, J., Bennett, J., and Zwaenepoel, W. 1991. Implementation and performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, 152–164.

Carter, J., Bennett, J., and Zwaenepoel, W. 1995. Techniques for reducing consistency-related information in distributed shared memory systems. *ACM Trans. Comput. Syst. 13*, 3 (Aug.), 205–243.

Casanova, H., Dongarra, J., and Doolin, D. 1997. Java access to numerical libraries. *Concurr. Pract. Exper. 9*, 11.

Castro, M., Adya, A., Liskov, B., and Myer, A. C. 1997. Hac: Hybrid adaptive caching for distributed storage systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*.

Chandra, R., Gupta, A., and Hennessy, J. 1994. Cool: An object-based language for parallel programming. *IEEE Computer 27*, 8 (Aug.), 14–26.

Christiansen, B., Cappello, P., Ionescu, M. F., Neary, M. O., Schauser, K. E., and Wu, D. 1997. Javelin: Internet-based parallel computing using Java. *Concurr. Pract. Exper. 9*, 11 (Nov.), 1139–1160.

Das, M. 2000. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*.

Deutsch, L. P. and Schiffman, A. M. 1984. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, 297–302.

Dwarkadas, S., Gharachorloo, K., Kontothanassis, L., Scales, D. J., Scott, M. L., and Stets, R. 1999. Comparative evaluation of fine- and coarse-grain approaches for software distributed shared memory. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, 260–269.

Ferreira, P. and Shapiro, M. 1994. Garbage collection and DSM consistency. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*.

Fox, G. and Furmanski, W. 1996. Towards Web/Java based high performance distributed computing—an evolving virtual machine. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*.

Freeh, V. and Andrews, G. 1996. Dynamically controlling false sharing in distributed shared memory. In *Proceedings of the Fifth Symposium on High-Performance Distributed Computing*.

Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture*, 15–26.

Goldberg, A. and Robson, D. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, Mass.

Harris, J. and Sarkar, V. 1998. Lightweight object-oriented shared variables for distributed applications on the Internet. In *OOPSLA '98 Conference Proceedings*, 296–309.

Herlihy, M. 1999. The aleph toolkit: Support for scalable distributed shared objects. In *CANPC '99.* Lecture Notes in Computer Science, vol. 1602, Springer-Verlag, New York.

Intel Corporation. 2001. IA-32 *Intel Architecture Software Developer's Manual, Volume* 3: *System Programming Guide*.

Itzkovitz, A. and Schuster, A. 1999. Multiview and millipage—fine-grain sharing in page-based DSMs. In *Proceedings of the Third USENIX Symposium on Operating System Design and Implementation*.

James, J. and Singh, A. K. 2000. Design of the Kan distributed object system. *Concurr. Pract. Exper. 12*, 8, 755–797.

JavaGrande. Javagrande. Available at http://www.javagrande.org/.

Kachler, T. and Krasner, G. 1990. *LOOM—Large Object-Oriented Memory for Smalltalk-80 Systems*. Morgan Kaufmann, San Francisco, 298–307.

Kale, L., Bhandarkar, M., and Wilmarth, T. 1997. *Design and Implementation of Parallel Java with Global Object Sace*. Morgan Kaufmann, San Francisco, 235–244.

Keleher, P., Cox, A. L., and Zwaenepoel, W. 1992. Lazy release consistency for software distributed shared memory. In *Proceedings of the Nineteenth Annual International Symposium on Computer Architecture*, 13–21.

Kordale, R., Ahamad, M., and Shilling, J. 1993. Distributed/concurrent garbage collection in distributed shared memory systems. In *Proceedings of the International Workshop on Object Orientation and Operating Systems*.

Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. 1991. The objectstore database system. *CACM 34*, 10 (Oct.), 50–63.

Lamport, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. C-28,* 9 (Sept.), 690–691.

Landi, W. and Ryder, B. 1992. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN 92 Conference on Programming Language Design and Implementation*.

LE SERGENT, T. AND BERTHOMIEU, B. 1992. Incremental multi-threaded garbage collection on virtually shared memory architectures. In *Proceedings of the International Workshop on Memory Management*.

LI, K. AND HUDAK, P. 1989. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst. 7*, 4 (Nov.), 321–359.

LU, H., DWARKADAS, S., COX, A. L., AND ZWAENEPOEL, W. 1997. Quantifying the performance differences between PVM and TreadMarks. *J. Parallel Distrib. Comput. 43*, 2 (June), 56–78.

MATTHEWS, D. C. J. AND LE SERGENT, T. 1995. Lemma: A distributed shared memory with global and local garbage collections. In *Proceedings of the International Workshop on Memory Management*.

PHILIPSEN, M. AND ZENGER, M. 1997. Javaparty—transparent remote objects in Java. *Concur. Pract. Exper. 9*, 11 (Nov.), 1225–1242.

RINARD, M. C. AND LAM, M. S. 1998. The design, implementation, and evaluation of Jade. *ACM Trans. Program. Lang. Syst. 20*, 3 (May), 483–545.

SCALES, D., GHARACHORLOO, K., AND THEKKATH, C. 1996. Shasta: A low overhead software-only approach for supporting fine-grain shared memory. In *Proceedings of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems*.

SCALES, D. J. AND LAM, M. S. 1994. The design and evaluation of a shared object system for distributed memory machines. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, 101–114.

SCHOINAS, I., FALSAFI, B., LEBECK, A. R., REINHARDT, S. K., LARUS, J. R., AND WOOD, D. A. 1994. Fine-grain access control for distributed shared memory. In *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems*, 297–306.

SINGH, J., WEBER, W.-D., AND GUPTA, A. 1992. SPLASH: Stanford parallel applications for shared-memory. *Comput. Arch. News 20*, 1 (Mar.), 2–12.

TANENBAUM, A., KAASHOEK, M., AND BAL, H. 1992. Parallel programming using shared objects and broadcasting. *IEEE Computer 25*, 8 (Aug.), 10–20.

TARDITI, D. AND DIWAN, A. 1996. Measuring the cost of storage management. *Lisp Symbol. Comput. 9*, 4 (Dec.).

THOMAS, R. 1981. A dataflow computer with improved asymptotic performance. Tech. Rep. TR-265, MIT Laboratory for Computer Science.

TIMBER. Timber: A Spar/Java compiler. Available at http://www.pds.twi.tudelft.nl/timber/.

TOWER TECHNOLOGIES. Towerj 3.0: A new generation native Java compiler and runtime environment. Available at http://www.towerj.com/.

VELDEMA, R., HOFMAN, R., BHOEDJANG, R., AND BAL, H. E. 2001a. Runtime optimizations for a Java DSM implementation. In *Proceedings of the Joint ACM Java Grande—ISCOPE 2001 Conference*.

VELDEMA, R., HOFMAN, R., BHOEDJANG, R., JACOBS, C., AND BAL, H. E. 2001b. Jackal: A compiler-supported distributed shared memory implementation of Java. In *Proceedings of the Eighth Symposium on the Principles and Practice of Parallel Programming*.

WATSON, P. AND WATSON, I. 1987. An efficient garbage collection scheme for parallel computer architectures. In *PARLE'87—Parallel Architectures and Languages Europe*. Lecture Notes in Computer Science, vol. 259, Springer-Verlag, Eindhoven, the Netherlands.

WHITE, S. AND DEWITT, D. 1992. A performance study of alternative object faulting and pointer swizzling strategies. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*.

WHITE, S. AND DEWITT, D. 1994. Quickstore: A high performance mapped object store. In *Proceedings of the 1994 ACM SIGMOD Conference*.

WILKINSON, T. 1996. Kaffe: A virtual machine to run Java code. Available at http://www.kaffe.org/.

WILSON, P. R. AND MOHER, T. G. 1989. Design of the opportunistic garbage collector. In *Proceedings of the Fourth ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 23–35.

WILSON, P. R. AND MOHER, T. G. 1992. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, 364–377.

Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the Twentysecond Annual International Symposium on Computer Architecture*, 24–36.

Yu, W. and Cox, A. L. 1996. Conservative garbage collection on DSM systems. In *Proceedings of the Sixteenth International Conference on Distributed Computing Systems*.

Received October 2000; revised April 2002; accepted June 2002