Under revision for *ACM Trans. on Computer Systems*. An earlier version appears in Proceedings of LCR2000: the 5th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers.

Run-Time Support for Distributed Sharing in Typed Languages

Y. Charlie Hu, Weimin Yu, Alan Cox, Dan Wallach and Willy Zwaenepoel[*]

Department of Computer Science
Rice University
Houston, Texas 77005
{ychu, weimin, alc, dwallach, willy}@cs.rice.edu

Key words: shared memory, DSM, typed programming languages

Abstract

We present a new run-time system for typed programming languages that supports object sharing in a distributed system. The key insight in this system is that the ability to distinguish pointers from data at run-time enables efficient and transparent sharing of data with both fine-grained and coarse-grained access patterns. In contrast, conventional distributed shared memory (DSM) systems that support sharing of an untyped memory region are limited to providing only one granularity with good performance.

This new run-time system, DOSA, provides a shared object space abstraction rather than a shared address space abstraction. Three key aspects of the design are: First, DOSA uses type information, in particular, the ability to unambiguously recognize references, to make fine-grained sharing efficient by supporting object granularity coherence. Second, DOSA aggregates the communication of objects, making coarse-grained sharing efficient. Third, DOSA uses a globally unique "handle" rather than a virtual address to name an object, enabling each machine to allocate storage just for the objects that it accesses, improving spatial locality.

We compare DOSA to TreadMarks, a conventional DSM system that is efficient at handling coarse-grained sharing. Our performance evaluation substantiates the following claims:

1. *The performance of fine-grained applications is considerably (up to 98% for Barnes-Hut and 62% for Water-Spatial) better than in TreadMarks.*
2. *The performance of garbage-collected applications is considerably (up to 65%) better than in TreadMarks.*
3. *The performance of coarse-grained applications is nearly as good as in TreadMarks (within 6%). Since the performance of such applications is already good in TreadMarks, we consider this an acceptable performance penalty.*

1 Introduction

This paper addresses run-time support for sharing objects in a typed language between the different computers within a cluster. Typing must be strong enough that it is possible to determine unambiguously whether a memory location contains an object reference or not. Many modern languages fall under this category, including Java and Modula-3. Direct access through a reference to object data is supported, unlike Java/RMI or Orca [2], where remote object access is restricted to method invocation. Furthermore, in languages with suitable multithreading support, such as Java, distributed execution is transparent: no new API is introduced for distributed sharing. This transparency distinguishes this work from many earlier distributed object sharing systems [2, 6, 11, 9].

The key insight in this paper is that *the ability to distinguish pointers from data at run-time enables efficient and transparent sharing of data with both fine-grained and coarse-grained access patterns*. In contrast, conventional distributed shared memory (DSM) systems that support sharing of an untyped memory region are limited to providing only one granularity with good performance. Indeed, DSM systems have been divided into those offering support for coarse-grained sharing or for fine-grained

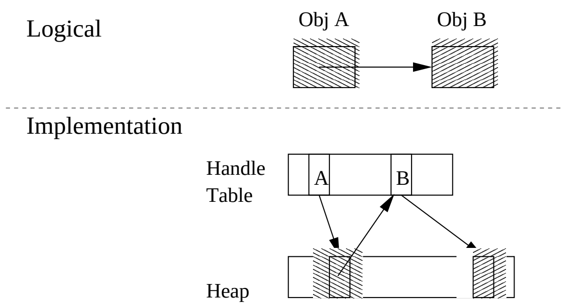

Figure 1. Objects with handles.

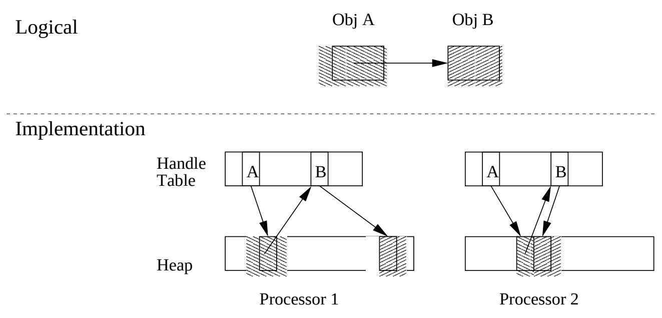

Figure 2. Shared objects identified by unique OIDs.

sharing. Coarse-grain sharing systems are typically page-based, and use the virtual memory hardware for access and modification detection. Although relaxed memory models and multiple-writer protocols relieve the impact of the large page size, fine-grain sharing and false-sharing remain problematic. Throughout this paper, we will use TreadMarks [1] as the representative of such systems, but the results apply to similar systems. Fine-grain sharing systems typically augment the code with instructions to detect reads and writes, 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. In addition, these systems do not benefit from the implicit aggregation effect present in the page-based systems. Fine-grained systems typically require a message per object, while page-based systems bring in all data in a page at once, avoiding additional messages if the application accesses other objects in the same page. Again, in this paper we will use a single system, Shasta [10], to represent this class of systems, but the discussion applies to similar systems.

Consider a (single-processor) implementation of such a strongly-typed language using a *handle table* (see Figure 1). 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. All references to an object refer in fact to its entry in the handle table, which in turn points to the actual object. In such an implementation, it is easy to relocate objects in memory. It suffices to change the corresponding entry in the handle table. No other changes need to be made, since all references are indirected through the handle table.

Extending this simple observation allows an efficient distributed implementation of these languages. Specifically (see Figure 2), 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 both TreadMarks and Shasta.

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. Fine-grain 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. A change in access mode is accomplished by switching between the different mappings *for that object only*. The mappings for the other objects in the same page remain unaffected. Consider the example in Figure 3. A page contains four objects, one of which 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 on the same page.

In addition to avoiding false sharing, this organization has numerous other benefits. First, on a particular processor, memory needs to be allocated only for those objects that are accessed on that processor, resulting in a smaller memory footprint and better cache locality. N-body simulations illustrate this benefit. 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 memory, causing lots of

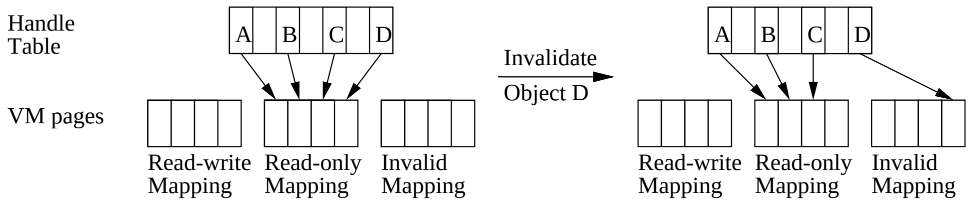

Figure 3. Access detection using the handle pointers.

misses, 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, there are far fewer misses and messages, and false sharing is eliminated through the per-object mappings. Moreover, objects can be locally re-arranged in memory, for instance to improve cache locality or during garbage collection, without affecting the other processors. Finally, the aggregation effect of TreadMarks can be maintained as well. When a fault is detected on an object in a particular page, all invalidated objects in the same page as the faulted object are brought up-to-date. While this approach potentially re-introduces false sharing, its harmful effects are much smaller than in a conventional page-based system, because we are free to co-locate or not to co-locate certain objects in a page on a per-processor basis. Returning to the N-body application, the location of bodies typically changes slowly over time, and a given processor accesses many of the same bodies from one iteration to the next. Thus, bringing in all bodies in the same page on the first access miss to any one of them is beneficial.

While there are many apparent performance benefits, there are some obvious questions about the performance of such a system as well. For instance, the extra indirection is not free, and consistency information now needs to be communicated per-object rather than per-page, potentially leading to a large increase in its size. To evaluate these tradeoffs, we have implemented the system outlined above, and compared its performance to that of TreadMarks. We have derived our implementation from the same code base as TreadMarks, avoiding, to the largest extent possible, performance differences due to unrelated code differences. Our performance evaluation substantiates the following claims:

1. The performance of fine-grained applications is considerably better (up to 98% for Barnes-Hut and 62% for Water-Spatial) than in TreadMarks.

2. The performance of garbage-collected applications is considerably (up to 65%) better than in TreadMarks.

3. The performance of coarse-grained applications is nearly as good as in TreadMarks (within 6%). Since the performance of such applications is already good in TreadMarks, we consider this an acceptable performance penalty.

Unfortunately, there is no similarly available implementation of fine-grained shared memory, so an explicit comparison with such a system could not be made, but we offer some speculations based on published results comparing Cashmere [7], a coarse-grained system, to Shasta.

The outline of the rest of this paper is as follows. Section 2: API and memory model. Section 3: Implementation and comparison with conventional systems. Section 4: Compiler optimizations for coarse-grained applications. Section 5: Experimental methodology. Section 6: Environment. Section 7: Applications. Section 8: Overall results for fine-grained, garbage-collected, and coarse-grained applications. Section 9: Breakdown of optimizations. Section 10: Related work. Section 11: Conclusions.

2 API and Memory Model

2.1 API

The general model is a shared space of objects, in which each reference to an object is typed. The programmer is responsible for creating and destroying threads of control, and for the necessary synchronization to insure orderly access by these threads to the object space. Various synchronization mechanisms may be used, such as semaphores, locks, barriers, monitors, etc. No special API is required in languages with suitable typing and multithreading support, such as Java or Modula-3. Unlike Orca, we do allow references to be used for accessing objects. We do not require a method invocation for each access.

An object is the unit of sharing. In other words, 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.

The single-writer nature of individual objects is not inherent to the design of our system, but we have found that it corresponds to common usage, and is therefore not restrictive. As will be seen in Section 3, it allows us to use an efficient single-writer protocol for individual objects.

2.2 Memory Model: Release Consistency

The object space is release consistent. Release consistency (RC) is a relaxed memory consistency model. 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.

3 Implementation

We focus on the consistency maintenance of individual objects. Synchronization is implemented as in TreadMarks.

3.1 Consistency Protocol

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. This information is carried in the form of write notices.

The protocol maintains a vector timestamp on each processor, the ith element of which records the highest interval number of processor i that has been seen locally. An interval is an epoch between two consecutive synchronization operations. The interval number is simply a count of the number of intervals on a processor. Each write notice has an associated processor identifier and vector timestamp, indicating where and when the modification of the object occurred. To avoid repeated sending of write notices, a processor sends its vector timestamp on an acquire, and the responding processor sends only those write notices with a vector timestamp between the received vector timestamp and its own current vector timestamp.

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.

In DOSA, the write notices are in terms of objects. As a consequence, for very fine-grained applications, the number of write notices can potentially be much larger than in a page-based DSM. To this end, DOSA employs a novel compression technique to reduce the number of write notices transmitted during synchronizations.

Each time a processor creates a new interval, it traverses in reverse order old intervals that it has created before and looks for the one that consists of similar write notices. If such a “match” is found, the difference between the new interval and the old interval are presumably much smaller than write notices themselves. The processor can then create and later transmit when requested only the write notices that are different from those of the matched old interval, and thus reduce the consistency data. Since intervals are always received and incorporated in the forward order, when a processor receives such an interval containing difference of write notices, it is guaranteed to have already received the old interval based on which the diff of the new interval is made. It can then easily reconstruct the write notices of the new interval.

3.2 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. The object’s local state, i.e., invalid, read-only, or read-write, is also reflected in the handle table entry through different mappings of the object’s local virtual address with the corresponding protection attributes (see Section 3.4). 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 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 3.6). Finally, each processor maintains its vector timestamp and an efficient data structure for sending write notices when responding to an acquire.

As a practical matter, OIDs are currently assigned as the virtual addresses of the entry in the handle table. There-

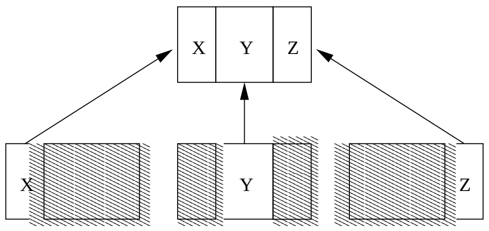

Figure 4. Multiview: one vpage for each object.

fore, the handle table must reside at the same virtual address on all processors. Should this ever become a restriction, it could easily be removed.

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.

3.3 Object Storage Allocation

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

3.4 Switching Protection

DOSA relies on hardware page protection mechanism to detect accesses to invalid objects and write accesses to read-only objects. We create three non-overlapping virtual address regions that map to the same physical memory, from where shared objects are allocated. An object thus can 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 accesses 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 shared memory region differ in two leading bits of their addresses. Therefore, changing protection is a simple bit masking operation.

This approach is superficially similar to the MultiView approach used in Millipede [8], but in fact it is fundamentally different. In MultiView a physical page may be mapped at multiple addresses in the virtual address space, as in DOSA, but the similarity ends there. In MultiView, each object resides in its own *vpage*, which is the size of a VM page. Different vpages are mapped to the same physical memory page, but the objects are offset within the vpage such that they do not overlap in the underlying physical page (see Figure 4). Different protection attributes may be set on different vpages, thereby achieving the same effect as DOSA, namely per-object access and write detection. The MultiView method requires one virtual memory mapping per object, while the DOSA method requires only three mappings per page, resulting in considerably less address space consumption and pressure on the TLB. Also, DOSA does not require any changes in the protection attributes of the mappings after initialization, while MultiView does.

3.5 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 were 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 (see Section 9.3). We refer to this optimization as write-aggregation.

3.6 Access Miss Handling and Read Aggregation

When a processor faults on a particular object, if the object is smaller than a page, it uses the list of objects in the same page (see Section 3.2) 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.

By doing so, we aggregate the requests for all objects in the same page. This approach performs better than simply

[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, an encoding of the changes to the page. Since we are using a single-writer protocol, there is no need for diffs.

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 local object storage allocation works to our advantage. It is true that some unnecessary data may be fetched, but the effect of that is minimal for the following reason.

2. With read aggregation as described above, the messages to fetch the different objects go out in parallel, and therefore their latencies and the latencies of the replies are largely overlapped.

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

3.7 Summary

We summarize with a discussion of the salient differences between DOSA on one hand, and TreadMarks and Shasta on the other hand.

DOSA shares with TreadMarks its use of invalidate-based lazy release consistency, its use of the VM system for access and write detection, and its page-based aggregation. It differs in that it allocates storage for shared data locally, rather than globally, it performs per-object rather than per-page access and write detection, and it uses a single-writer protocol per object rather than a multiple-writer protocol per page.

Shasta uses an invalidate-based eager release consistency. More importantly, it differs from DOSA in that it uses global rather than local memory allocation. It uses instrumentation rather than the VM system for access and write detection. It does access and write detection on a per "cache line" basis, where the cache line is implemented in software and can be varied from program to program. There is no attempt to aggregate data.

4 Compiler Optimizations

The extra indirection creates a potential problem for applications that access large arrays, because it may cause significant overhead, without any gain from better support for fine-grained sharing. This problem can be addressed using type-based alias analysis and loop invariant analysis to eliminate many repeated indirections.

Consider, a C program with a two-dimensional array of scalars, such as `float`, that is implemented in the same fashion as a two-dimensional Java array of scalars, i.e., an array of pointers to an array of a scalar type ("`scalar_type **a;`"). Assume this program performs a regular traversal of the array with a nested for loop.

```
for i
    for j
        ... = a[i][j];
```

In general, a C compiler cannot further optimize this loop nest, because it cannot prove that `a` and `a[i]` do not change during the loop execution. `a`, `a[i]` and `a[i][j]` are, however, of different types, and therefore the compiler for a typed language can easily determine that `a` and `a[i]` do not change, and transform the loop accordingly to

```
for i
    p = a[i];
    for j
        ... = p[j];
```

resulting in a significant speedup. In the DOSA program the original program takes the form of

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

which, in a typed language can be similarly transformed to

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

While offering much improvement, this transformation still leaves the DOSA program at a disadvantage compared to the optimized TreadMarks program, because of the remaining pointer dereferencing in the inner loop. Observe also that the following transformation of the DOSA program is legal but not profitable:

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

The problem with this transformation occurs when `a->handle[i]->handle` has been invalidated as a result of a previous synchronization. Before the j-loop, p contains an address in the invalid region, which causes a page fault on the first iteration of the j-loop. The DSM runtime changes `a->handle[i]->handle` to its location in the read-write region, but this change is not reflected in `p`. As a result, the j-loop page faults on every iteration.

We solve this problem by touching `a->handle[i]->handle[0]` before assigning it to `p`. In other words,

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

Touching `a->handle[i]->handle[0]` outside the j-loop causes the fault to occur there, and `a->handle[i]->handle` to be changed to the read-write location. The same optimization can be applied to the outer loop as well.

These optimizations are dependent on the lazy implementation of release consistency. Invalidations can only arrive at synchronization points, never asynchronously, thus the cached references cannot be invalidated in a synchronization-free loop.

5 Evaluation Methodology

Our performance evaluation seeks to substantiate the following claims:

1. The performance of fine-grained applications is considerably better than in TreadMarks.

2. The performance of garbage-collected applications is considerably better than in TreadMarks.

3. The performance of coarse-grained applications is nearly as good as in TreadMarks. Since the performance of such applications is already good in TreadMarks, we consider this an acceptable performance penalty.

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 typed language, and running them, on one hand, on TreadMarks, simply using shared memory as an untyped region of memory, and, on the other hand, running them on top of DOSA, using a shared object space.

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 implementation and virtual machine would dwarf differences between TreadMarks and DOSA. Perhaps more importantly, we expect efficient compiled versions of Java to become available soon, and we would expect that those be used in preference over the current implementations, quickly obsoleting our results. Finally, the performance of these Java applications would be much inferior to published results for conventional programming languages.

We have therefore chosen to carry out the following experiments. For comparisons 1 and 3, we have taken existing C applications, and we have re-written them to follow the model of a handle-based implementation. In other words, a handle table is introduced, and all pointers are indirected 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 compiler optimizations discussed in Section 4 have been implemented by hand in both the TreadMarks and the DOSA programs. We report results with and without these optimizations present.

For comparison 2, 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 two parts: the inter-processor algorithm, which tracks cross-processor references; and the intra-processor algorithm, which performs the traversal on each processor and reclaims the unused memory. Our distributed garbage collector uses a *weighted reference counting* algorithm for the inter-processor part [3, 13, 14] and a generational, copying algorithm for the intra-processor part. 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.

6 Experimental Environment

Our experimental platform is a switched, full-duplex 100Mbps Ethernet network of thirty-two 300 MHz Pentium II-based computers. Each computer has a 512K byte secondary cache and 256M bytes of memory. All of the computers were running FreeBSD 2.2.6 and communicating through UDP sockets. On this platform, the round-trip 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 1,333 microseconds. The time to obtain a diff varies from 313 to 1,544 microseconds, depending on the size of the diff. The time to obtain a full page is 1,308 microseconds.

7 Applications

Our choice of applications follows immediately from the goals of our performance evaluation. First, we use two

Application	Small Problem Size	Time (sec.)		Large Problem Size	Time (sec.)	
		Original	Handle		Original	Handle
Red-Black SOR	3070x2047, 20 steps	21.13	21.12	4094x2047, 20 steps	27.57	28.05
Water-N-Squared	1728 mols, 2 steps	71.59	73.83	2744 mols, 2 steps	190.63	193.50
Barnes-Hut	32K bodies, 3 steps	58.68	60.84	131K bodies, 3 steps	270.34	284.43
Water-Spatial	4K mols, 9 steps	89.63	89.80	32K mols, 2 steps	158.57	160.39

Table 1. Applications, input data sets, and sequential execution time.

fine-grained applications for which we hope to see significant benefits over a page-based system. These applications are Barnes-Hut and Water-Spatial from the SPLASH-2 [15] benchmark suite. Barnes-Hut is an N-body simulation, and Water-Spatial is a molecular dynamics simulation optimized for spatial locality.

Second, we use two coarse-grained applications to assess the potential performance loss in such applications, compared to a system that is geared towards such coarse-grained applications. These two applications are SOR and Water-N-Squared. SOR performs red-black successive over-relaxation on a 2-D grid, and Water-N-Squared is a molecular dynamics simulation from the SPLASH [12] benchmark suite.

For each of these applications, Table 1 lists the problem size and the sequential execution times. The sequential execution times were obtained by removing all TreadMarks or DOSA calls from the applications and for DOSA using the compile-time optimizations described in Section 4. The optimizations were applied by hand. 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 the four non-synthetic applications. The sequential execution times without handles were used as the basis for computing the speedups reported later in the paper.

Third, to exercise the distributed garbage collector, we use a modified version of the OO7 object-oriented database benchmark [5]. This benchmark is designed to match the characteristics of many CAD/CAM/CASE applications. The OO7 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 3 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 updating objects when the database changes. After the new object, containing the updated data, is in place in the database, the old object becomes eligible for collection.

The OO7 benchmark defines several database traversals [5]. For our experiments, we use a mixed sequence of T1, T2a, and T2b traversals. T1 performs a depth-first traversal of the entire composite part graph. T2a and T2b are identical to T1 except that T2a modifies the root atomic part of the graph, while T2b modifies all the atomic parts.

Table 2 lists the sequential execution times for OO7 running with the garbage collector on TreadMarks and DOSA. It also lists the time spent in the memory allocator/garbage collector. DOSA incurs 2% overhead to the copying collector because of extra overhead in handle management; it has to update the handle table entry whenever an object is created, deleted, or moved. Overall, DOSA underperforms TreadMarks by 3% due to handle dereference cost.

Tree	Tmk	DOSA
Overall time (in sec.)	184.8	190.8
Alloc and GC time (in sec.)	10.86	11.04

Table 2. Statistics for TreadMarks and DOSA on 1 processor for OO7 with garbage collection.

8 Overall Results

8.1 Fine-grained Applications

Figure 5 shows the speedup comparison between TreadMarks and DOSA for Barnes-Hut and Water-Spatial on 16 and 32 processors for small and large problem sizes. Figure 6 shows normalized statistics from the execution of these applications on 32 processors for both problem sizes. The detailed statistics are listed in Table 3.

We derive the following conclusions from this data. First, from Table 1, the overhead of the extra indirection in the sequential code for these applications is less than 5.2% for Barnes-Hut and 1.1% for Water-Spatial. Second, even for a small number of processors, the benefits of the handle-based implementation are larger than 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. Third, as the number of processors increases, the benefits of the handle-based

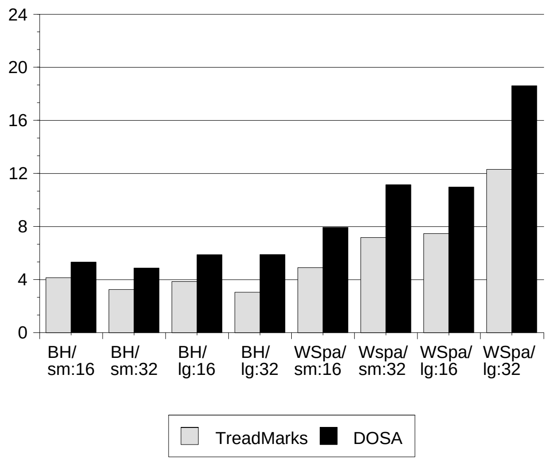

Figure 5. Speedup comparison between TreadMarks and DOSA for fine-grained applications.

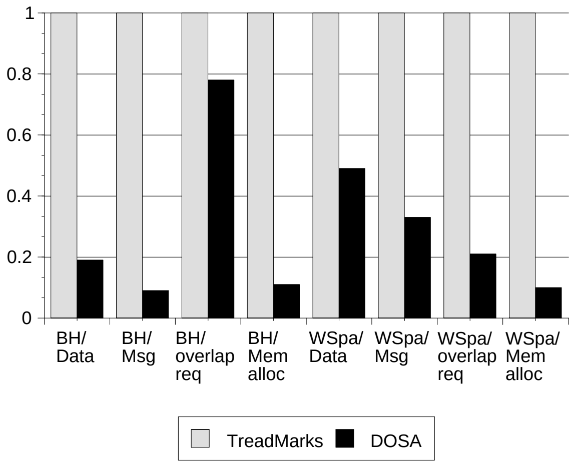

Figure 6. Statistics for TreadMarks and DOSA on 32 processors for fine-grained applications with large data sizes, normalized to TreadMarks measurements.

Application	Barnes-Hut/sm		Barnes-Hut/lg		Water-Spatial/sm		Water-Spatial/lg	
	Tmk	DOSA	Tmk	DOSA	Tmk	DOSA	Tmk	DOSA
Time	18.07	12.06	89.07	45.98	12.52	8.04	12.89	8.52
Data (MB)	315.3	82.6	1307	246	475.1	262.6	342.1	166.8
Messages	2549648	307223	10994350	1027932	617793	188687	330737	109560
Overlapped data requests	108225	98896	439463	341303	193692	66937	202170	41491
Object memory alloc. (MB)	7.36	1.05	29.4	3.35	3.15	0.61	25.2	2.64

Table 3. Detailed statistics for TreadMarks and DOSA on 32 processors for fine-grained applications, Barnes-Hut and Water-Spatial. In TreadMarks, a call to diff request which may involve parallel messages to different processors is counted as one overlapped request. In DOSA, a call to object request which may involve parallel messages to different processors to update other objects in the same page is counted as one overlapped request.

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. Fourth, if the amount of false sharing under TreadMarks decreases as the problem size increases, as in Water-Spatial, then DOSA's advantage over TreadMarks decreases. If, on the other hand, the amount of false sharing under TreadMarks doesn't change, as in Barnes-Hut, then DOSA's advantage over TreadMarks is maintained. In fact, for Barnes-Hut, the advantage grows due to slower growth in the amount of communication by DOSA, resulting from improved locality due to lazy object allocation.

The reasons for DOSA's clear dominance over TreadMarks can be seen in Figure 6. This figure shows the number of messages exchanged, the number of overlapped data requests [2], the amount of data communicated, 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.

8.2 Garbage Collected Applications

Figures 7 and 8 show the execution statistics on 16 processors for the OO7 benchmark running on TreadMarks and DOSA using the generational, copying collector. The detailed statistics are listed in Table 4. We do not present results 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 on DOSA outperforms OO7 on TreadMarks by almost 65%. Figure 7 shows that 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 actually 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 8.

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 is indistinguishable from ordinary writes. Consequently, when another processor accesses the object after garbage collection, the object is communicated to it, even though the object's contents have not been changed by the copy. In fact, the processor may have an up-to-date copy of the object in its memory, just at the wrong virtual address. In contrast, on DOSA, when a processor copies an object during garbage collection, it simply updates its handle table entry, which is local information that never propagates to other processors.

The lazy storage allocation in DOSA also contributes to the reduction in communication. In OO7, live objects and garbage may coexist in the same page. In TreadMarks, if a processor requests a page, it may get both live objects and garbage. In DOSA, however, only live objects will be communicated, reducing the amount of data communicated. This also explains why the memory footprint in DOSA is smaller than in TreadMarks.

8.3 Coarse-grained Applications

Figure 9 shows the speedup comparison between TreadMarks and DOSA for SOR and Water-Nsquared on 16 and 32 processors for small and large problem sizes. Figure 10 shows normalized statistics from the execution of these applications on 32 processors for both problem sizes. The detailed statistics are listed in Table 5. The results show that the performance of these coarse-grained applications in DOSA is within 6% as in TreadMarks.

9 Effects of the Various Optimizations

To achieve the results described in the previous section, various optimizations were used in DOSA. These optimizations include lazy object allocation (Section 3.3), read aggregation (Section 3.6), write aggregation (Section 3.5), and compile-time optimization (Section 4). To see what effect each optimization has individually, we performed the following experiments: For each of the optimizations, we compare the performance of DOSA without that optimization to the fully-optimized system. Figure 11 shows the speedups for each of the experiments, except for compile-time optimization, for Barnes-Hut, Water-Spatial, and Water-N-Squared. The compile-time optimization is omitted because it only effects SOR. SOR is omitted because the only optimization that has any effect is the compile-time optimization. Table 6 provides further detail beyond speedups on the effects of the optimizations.

[2] The concurrent messages for updating a page in TreadMarks or updating all invalid objects in a page in DOSA are counted as one overlapped data request. Since these messages go out and replies come back in parallel, their latencies are largely overlapped.

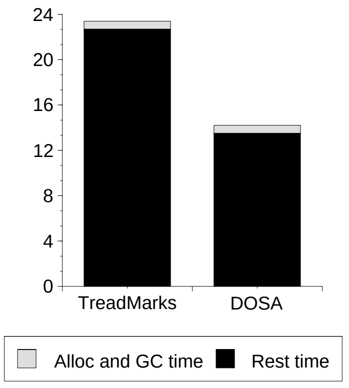

Figure 7. Time breakdown (in seconds) for the OO7 benchmark on TreadMarks and DOSA on 16 processors.

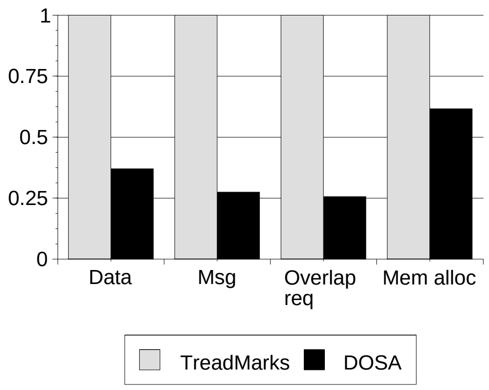

Figure 8. Statistics for OO7 on TreadMarks and DOSA on 16 processors, normalized to TreadMarks measurements.

Tree	OO7	
	Tmk	DOSA
Time	23.4	14.2
Alloc and GC time	0.70	0.70
Data (MB)	48.4	17.9
Messages	427811	117403
Overlapped data requests	171024	43747

Table 4. Detailed statistics for TreadMarks and DOSA on 16 processors for OO7.

9.1 Lazy Object Allocation

Table 6 shows that without lazy object allocation, DOSA sends 57% and 68% more messages and runs 13% and 18% slower than DOSA with lazy object allocation, for Barnes-Hut and Water-Spatial, respectively.

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 segment consisting of half of the array elements in a fixed increasing order.

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 the introduction, for such applications, lazy object allocation will only allocate space for objects on a processor that are accessed by that processor. Therefore, a physical page will contain only "useful" objects. With read aggregation, these objects will all be updated in a single round of parallel 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 the computation. In other words, objects accessed by a specific processor may be scattered in many more pages than in the scenario with lazy object allocation. When accessing these objects, this processor has to fault many more times and send many more rounds of messages in order to update them.

9.2 Read Aggregation

The single optimization that affects performance most is read aggregation. Table 6 shows that without read aggregation, DOSA sends 2.2, 4.4, and 5.2 times more messages, and 3.3, 5.8, and 5.8 times more data message rounds for Barnes-Hut, Water-Spatial, and Water-N-Squared, respectively. As a consequence, DOSA without read aggregation is 310%, 24%, and 22% slower than DOSA for these three applications.

Intuitively, the potential problem with read aggregation is that DOSA may fetch more objects than necessary. DOSA without read aggregation, however, only fetches accessed or necessary objects. Thus, by looking at the differ-

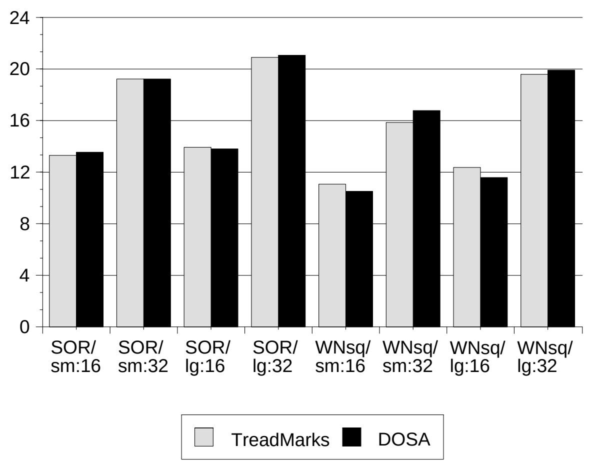

Figure 9. Speedup comparison between TreadMarks and DOSA for coarse-grained applications.

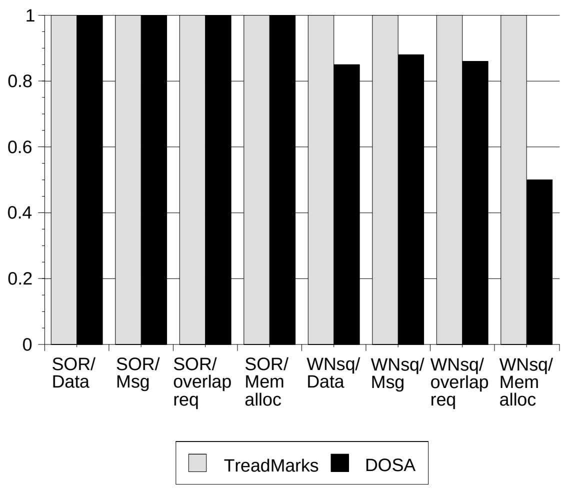

Figure 10. Statistics for TreadMarks and DOSA on 32 processors for coarse-grained applications with large data sizes, normalized to TreadMarks measurements.

Application	SOR/sm		SOR/lg		Water-Nsquare/sm		Water-Nsquare/lg	
	Tmk	DOSA	Tmk	DOSA	Tmk	DOSA	Tmk	DOSA
Time	1.10	1.10	1.32	1.31	4.52	4.27	9.74	9.58
Data (MB)	23.6	23.6	23.6	23.6	134.1	114.0	212.4	181.4
Messages	12564	12564	12440	12440	77075	63742	114322	101098
Overlapped data requests	4962	4962	4962	4962	33033	28032	51816	44758
Object memory alloc. (MB)	1.64	1.64	1.64	2.18	1.58	0.66	2.10	1.04

Table 5. Detailed statistics for TreadMarks and DOSA on 32 processors for coarse-grained applications SOR and Water-N-Squared.

ence between DOSA with and without read aggregation, we can determine the amount of unnecessary data communicated. Table 6 shows that DOSA without read aggregation sends almost the same amount of data as fully-optimized DOSA for Water-Spatial and Water-N-Squared, but half as much data for Barnes-Hut. The data totals for Water-Spatial and Water-N-Squared are nearly identical because lazy object allocation improves the initial spatial locality of the data on each processor. Since the set of molecules accessed by each processor remains static, spatial locality is good throughout the execution. Consequently, objects prefetched by read aggregation are typically used. In Barnes-Hut, however, 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 will still update the hole.

9.3 Write Aggregation

Table 6 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. As a result, DOSA is one second or 2.2% faster for Barnes-Hut than DOSA without write aggregation. The impact on Water-Spatial and Water-N-Square is marginal.

9.4 Write Notice Reduction

Table 6 shows that our write notice reduction optimization is highly effective for Barnes-Hut and Water-Spatial. For Barnes-Hut, it reduces the amount of write notice data by a factor of 5.5, resulting in a 10% performance improvement; and for Water-Spatial, it reduces the amount of write notice data by a factor of 4.5, resulting in a 3% performance improvement. This optimization has little effect on the other applications.

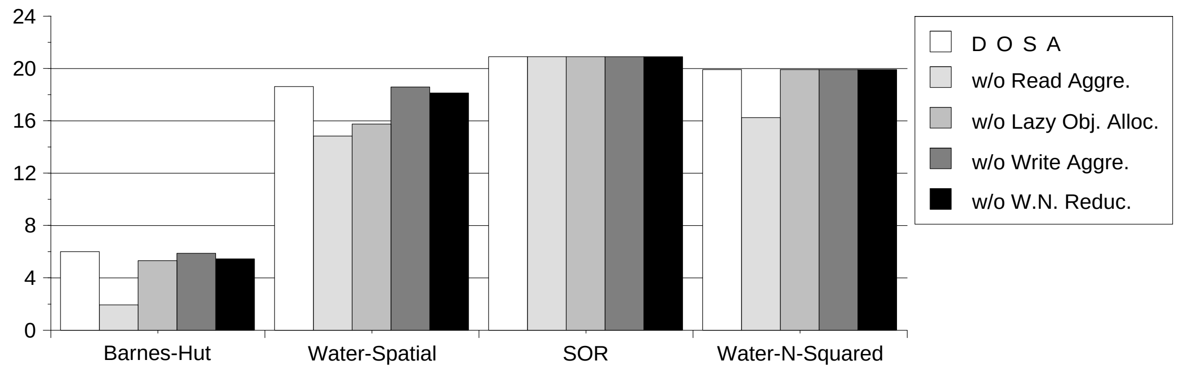

Figure 11. Speedup comparison between DOSA and DOSA without each of the optimizations on 32 processors.

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

Table 6. Statistics for DOSA and DOSA without each of the optimizations on 32 processors for Barnes-Hut, Water-Spatial, and Water-N-Squared.

9.5 Compile-time Optimization

Table 7 shows that for SOR, the compile-time optimization can significantly improve the performance. On a single processor, the compile-time optimization improves the performance of the original array-based version of SOR/lg by 20%, and the handle-based version by 69%. On 32 nodes, the improvements are 17% and 40% for the two versions, respectively.

10 Related Work

Two other systems have used VM mechanisms for fine-grain DSM: Millipede [8] and the Region Trapping Library [4]. The fundamental difference between DOSA and these systems is that *DOSA takes advantage of a typed language to distinguish a pointer from data at run-time and these other systems do not*. This allows DOSA to implement a number of optimizations that are not possible in these other systems.

Specifically, in Millipede a physical page may be mapped at multiple addresses in the virtual address space, as in DOSA, but the similarity ends there. In Millipede, each object resides in its own *vpage*, which is the size of a VM page. Different vpages are mapped to the same physical memory page, but the objects are offset within the vpage such that they do not overlap in the underlying physical page. Different protection attributes may be set on different vpages, thereby achieving the same effect as DOSA, namely per-object access and write detection. The Millipede method requires one virtual memory mapping per object, while the DOSA method requires only three mappings per page, resulting in considerably less address space consumption and pressure on the TLB. Also, DOSA does not require any costly OS system calls (e.g., `mprotect`) to change page protections after initialization, while Millipede does.

The Region Trapping Library is similar to DOSA in that it allocates three different regions of memory with different protection attributes. Unlike DOSA, it doesn't use the regions in any way that is transparent to the programmer. Instead, it provides a special API. Furthermore, in the implementation, the read memory region and the read-write memory region are backed by *different* physical memory regions. This decision has the unfortunate side effect of forcing modifications made in the read-write region to be copied to the read region, every time protection changes from read-write to read.

Orca [2], Jade [9], COOL [6], and SAM [11] 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, while DOSA does not. DOSA aims to provide transparent object sharing for existing typed languages, such as Java. Furthermore, none of Orca, Jade, COOL, or SAM use VM-based mechanisms for object sharing.

Dwarkadas et al. [7] 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). The only surprise was that Shasta equaled Cashmere on the fine-grained application Water-Spatial. They attributed this result to the run-time overhead of the inline 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.

11 Conclusions

In this paper, we have presented a new run-time system, DOSA, that efficiently implements a shared object space abstraction underneath a typed programming language. The key insight behind DOSA is that *the ability to unambiguously distinguish pointers from data at run-time enables efficient fine-grained sharing using VM support*. 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 a page-based system on coarse-grained applications. Furthermore, its architecture permits optimizations, such as lazy object allocation, which are not possible in conventional fine-grained 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 100Mbps Ethernet demonstrates that the new system performs comparably to TreadMarks for coarse-grained applications (SOR and Water-Nsquare), and significantly outperforms TreadMarks for fine-grained applications (up to 98% for Barnes-Hut and 62% for Water-Spatial) and a garbage-collected application (65% for OO7).

We have also presented a complete breakdown of the performance results, in particular, the contributions of lazy object allocation and read aggregation to DOSA's performance. Their effects are significant: Without lazy object allocation, on 32 processors, Barnes-Hut runs 13% slower and Water-Spatial runs 18% slower; and without read aggregation, on 32 processors, Barnes-Hut runs 68% slower, Water-Spatial runs 24% slower, and Water-N-Squared runs 22% slower.

Application	Tmk/pref.		Tmk/nopref.		DOSA/pref.		DOSA/nopref.	
	1-proc	32-proc	1-proc	32-proc	1-proc	32-proc	1-proc	32-proc
SOR (lg)	27.57	1.32	33.19	1.54	28.05	1.31	47.47	1.84

Table 7. Running time (sec.) comparison between SOR with and without the compile-time optimization.

References

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

[2] H. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Ruhl, and M. Kaashoek. Performance evaluation of the Orca shared object system. *ACM Transactions on Computer Systems*, 16(1), Feb. 1998.

[3] D. I. Bevan. Distributed garbage collection using reference counting. In *Parallel Arch. and Lang. Europe*, pages 117–187, Eindhoven, The Netherlands, June 1987. Spring-Verlag Lecture Notes in Computer Science 259.

[4] T. Brecht and H. Sandhu. The region trap library: Handling traps on application-defined regions of memory. In *Proceedings of the 1999 USENIX Annual Tech. Conf.*, June 1999.

[5] M. Carey, D. DeWitt, and J. Naughton. The OO7 benchmark. Technical report, University of Wisconsin-Madison, July 1994.

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

[7] S. Dwarkadas, K. Gharachorloo, L. Kontothanassis, D. J. Scales, M. L. Scott, and R. Stets. 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*, pages 260–269, Jan. 1999.

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

[9] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, 20(3):483–545, May 1998.

[10] D. Scales, K. Gharachorloo, and C. Thekkath. Shasta: A low overhead software-only approach for supporting fine-grain shared memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[11] D. J. Scales and M. S. Lam. 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*, pages 101–114, Nov. 1994.

[12] J. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):2–12, Mar. 1992.

[13] R. Thomas. A dataflow computer with improved asymptotic performance. Technical Report TR-265, MIT Laboratory for Computer Science, 1981.

[14] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In *PARLE'87—Parallel Architectures and Languages Europe*, number 259 in Lecture Notes in Computer Science, Eindhoven (the Netherlands), June 1987. Springer-Verlag.

[15] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.