Run-Time Support for Distributed Sharing in Typed Languages

Y. Charlie Hu, Weimin Yu, Alan L. Cox, Dan S. Wallach and Willy Zwaenepoel

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

Abstract. We present 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 that are not possible in conventional fine-grained or coarse-grained DSM systems.

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, 7, 14, 12].

The key insight in this paper is that *the ability to distinguish pointers from data at run-time enables more efficient fine-grained sharing* than is possible with conventional distributed shared memory (DSM) implementations that do not use type information (e.g., [1, 13]). Conventional VM-based DSM systems have only achieved good performance on relatively coarse-grained applications, because of their reliance on VM pages. Although relaxed memory models [9] and multiple-writer protocols [6] reduce the impact of the large page size, fine-grained sharing and false sharing remain problematic [1]. Fine-grained DSM systems

have been built using code instrumentation, but they have been limited by the cost of instrumentation and lack of communication aggregation [8]. The system presented here, DOSA, uses the ability to distinguish pointers from data at run-time to achieve efficient fine-grained sharing *using VM support and without using instrumentation*. It does so by introducing a level of indirection that allows objects to reside at different virtual memory locations with different protection attributes. Compiler optimization reduces the overhead of this level of indirection where necessary.

We have implemented this system, and compared its performance to that of TreadMarks, a state-of-the-art page-based system [1]. We have derived our implementation from the TreadMarks code base, thereby avoiding performance differences due to irrelevant 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.

No direct comparison with an instrumentation-based DSM system was possible, because no such system is broadly available, but we speculate on the likely outcome of such a comparison in Section 8.

2 Programming Model

No special API is required in languages with suitable typing and multithreading support, such as Java or Modula-3.

The programming model supports a shared space of objects, in which references are distinguishable from data. An object is the unit of sharing. In other words, a single object cannot be written concurrently by different threads, even if those threads modify distinct parts of 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 object space is release consistent [9]. In essence, under release consistency, the propagation of updates from one processor to another may be delayed until the processors synchronize. Parallel programs that are properly synchronized (i.e., synchronize between conflicting accesses to shared data) behave as expected on the conventional sequentially consistent shared memory model.

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

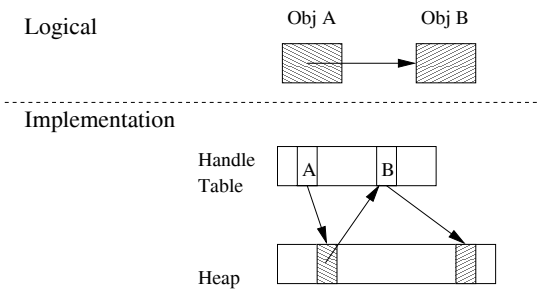

Fig. 1. Objects with handles.

semaphores, locks, barriers, monitors, etc. There is no system-level association between a synchronization variable and a particular object. For instance, a lock may protect a single object, multiple objects, or an array of objects.

3 Implementation

3.1 Single-machine Implementation

Consider a single-processor implementation of a typed language using a *handle table*. Each object is identified by an unique object identifier (OID) that is also the index of the object's handle table entry. Thus, all references to the object, in fact, refer to its handle table entry, which in turn points to the actual object (see Figure 1). In such an implementation, relocating objects in memory is easy. It suffices to change its handle table entry. No other changes need to be made, since all references are indirected through the handle table.

3.2 Distributed Implementation

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 index into the handle tables. As before, each handle table entry contains a pointer to the memory location where the object resides on that processor. The consistency protocol can then be implemented solely in terms of OIDs, because these are the actual 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 reside at the same virtual address on all processors, as is the case in a DSM system.

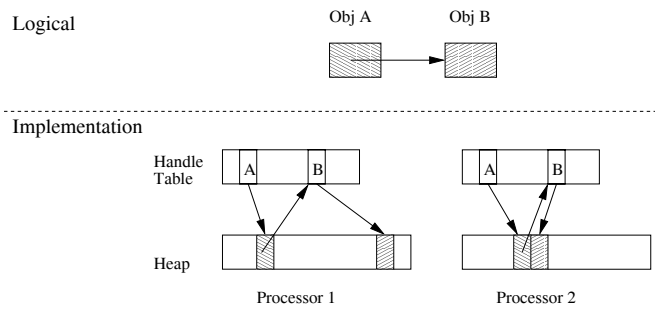

Fig. 2. Shared objects identified by unique OIDs.

3.3 Fine-grained VM access detection

The ability to locate the same object at different virtual memory addresses on different machines allows us to provide fine-grained access detection using VM techniques 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 map 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.

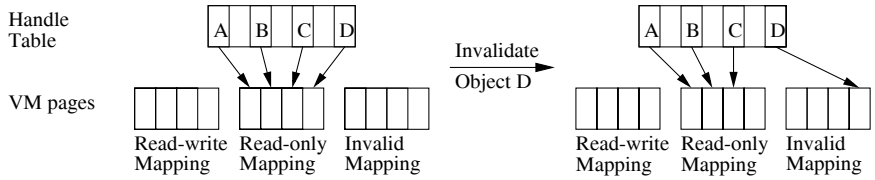

Fig. 3. Access detection using the handle pointers.

3.4 Object Storage Allocation

The ability to allocate objects at different addresses on different processors also 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*. For some programs, it reduces the memory footprint

and produces 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 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 of the smaller memory footprint, there are far fewer access 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.

3.5 Consistency Protocol

DOSA, like TreadMarks, uses a lazy invalidate protocol to implement release consistency [1]. Consistency is, however, maintained in terms of objects rather than pages. In other words, consistency messages specify object identifiers instead of page numbers. For individual objects, a single writer protocol is used [11]. For arrays of objects, whether of a scalar type or a reference type, a multiple writer protocol is used [1]. This permits the use of a single OID for the entire array, while still allowing concurrent modifications to distinct objects within the array.

The *lazy* implementation delays the propagation of consistency information until a processor acquires a lock or departs from a barrier. At that time, the last releaser of the lock or the barrier manager processor informs the processor which *objects* have been modified. In particular, invalidations never arrive asynchronously; they only arrive at the time of a synchronization.

An *inverse object table*, implemented as a hash table, is used by the page fault handler to translate a faulting address to an OID.

3.6 Read Aggregation

When a processor faults on a particular object, if the object is smaller than a page, it uses a list of objects in the same page 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 invalid objects in the same page. This approach performs better than simply fetching one faulted object at a time. 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, in particular given lazy object storage allocation. Some unnecessary objects may be fetched, but the messages to fetch those objects go out in parallel, and therefore their latencies and the latencies of the replies are largely overlapped.

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

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 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. We have taken existing C applications, and 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 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, 16, 17] 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 Environment and Applications

Our experimental platform is a switched, full-duplex 100Mbps Ethernet network of thirty-two 300 MHz Pentium II-based computers. Each computer has 256M bytes of memory, and runs FreeBSD 2.2.6.

We demonstrate the performance improvements of DOSA over TreadMarks for fine-grained applications, by using Barnes-Hut and Water-Spatial, both from the SPLASH-2 benchmarks [18]. SOR and Water-Nsquared from the SPLASH benchmarks [15] demonstrate only minimal performance loss for coarse-grained applications.

For each of these applications, Table 1 lists each of the problem sizes and its corresponding sequential execution time. The sequential execution times were obtained by removing all TreadMarks or DOSA calls from the applications. They also include the compile-time optimizations described in Section 4.

The sequential 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 four applications. The sequential execution times without handles were used as the basis for computing the speedups reported later in the paper.

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.

Table 1. Applications, problem sizes, and sequential execution time.

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

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.

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

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

7 Results

7.1 Fine-grained Applications

Figure 4 shows the speedup comparison between TreadMarks and DOSA for Barnes-Hut and Water-Spatial on 16 and 32 processors for small and large prob-

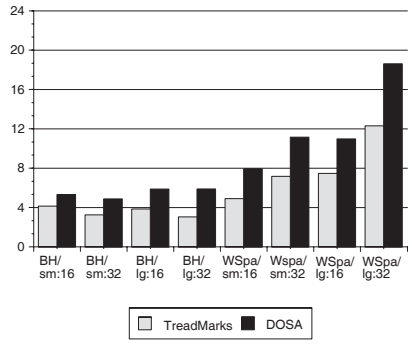

Fig. 4. Speedup comparison between TreadMarks and DOSA for fine-grained applications.

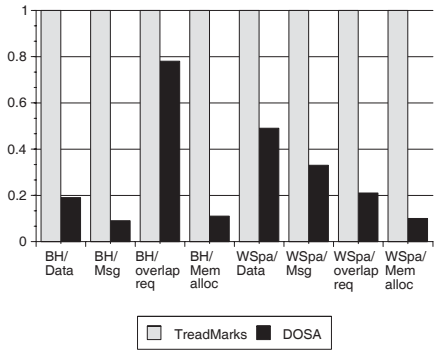

Fig. 5. Statistics for TreadMarks and DOSA on 32 processors for fine-grained applications with large data sizes, normalized to TreadMarks measurements.

lem sizes. Figure 5 shows various statistics from the execution of these applications on 32 processors for both problem sizes.

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 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 5. This figure shows the number of messages exchanged, the number of overlapped data requests [1], the amount of data communicated, and the average

[1] 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.

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.

7.2 Garbage Collected Applications

Figure 6 shows the execution statistics on 16 processors for the OO7 benchmark running on TreadMarks and DOSA using the generational, copying collector. 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 6 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 7.

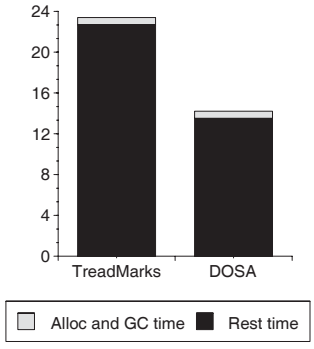

Fig. 6. Time breakdown (in seconds) for the OO7 benchmark on TreadMarks and DOSA on 16 processors.

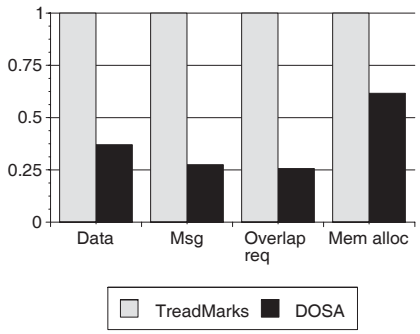

Fig. 7. Statistics for OO7 on TreadMarks and DOSA on 16 processors, normalized to TreadMarks measurements.

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

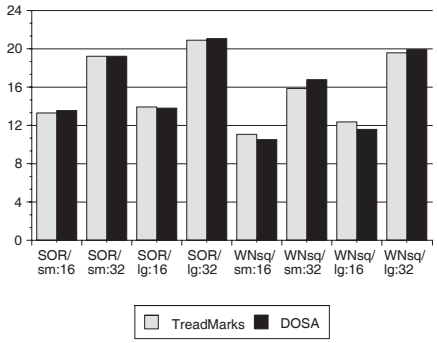

Fig. 8. Speedup comparison between TreadMarks and DOSA for coarse-grained applications.

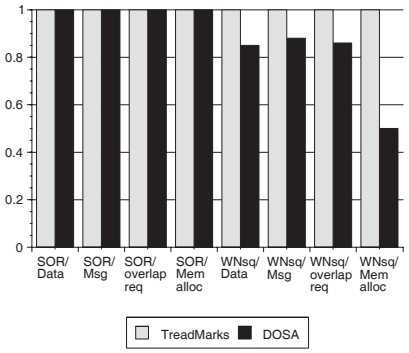

Fig. 9. Statistics for TreadMarks and DOSA on 32 processors for coarse-grained applications with large data sizes, normalized to TreadMarks measurements.

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.

7.3 Coarse-grained Applications

Figure 8 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 9 shows various statistics from the execution of these applications on 32 processors for both problem sizes.

8 Related Work

Two other systems have used VM mechanisms for fine-grain DSM: Millipede [10] 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 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 [12], COOL [7], and SAM [14] 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. [8] 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.

9 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-Nsquared), 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).

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. J. Carter, J. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related information in distributed shared memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, Aug. 1995.
7. R. Chandra, A. Gupta, and J. Hennessy. Cool: An object-based language for parallel programming. *IEEE Computer*, 27(8):14–26, Aug. 1994.
8. 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.
9. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.
10. 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.

11. P. Keleher. The relative importance of concurrent writers and weak consistency models. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 91–98, May 1996.
12. 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.
13. 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.
14. 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.
15. J. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):2–12, Mar. 1992.
16. R. Thomas. A dataflow computer with improved asymptotic performance. Technical Report TR-265, MIT Laboratory for Computer Science, 1981.
17. 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.
18. 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.