Run-Time Support for Distributed Sharing in Strongly-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

May 24, 1999

Abstract

In this paper, we present a new run-time system for strongly-typed programming languages that supports object sharing in a distributed system. The key insight in this system is that *type information allows 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, SkidMarks, provides a shared object space abstraction rather than a shared address space abstraction. Three key aspects of the design are: First, SkidMarks uses type information, in particular, the ability to unambiguously recognize references, to make fine-grained sharing efficient by supporting object granularity coherence. Second, SkidMarks aggregates the communication of objects, making coarse-grained sharing efficient. Third, SkidMarks 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 SkidMarks 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 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.

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

3. The performance of garbage-collected applications is considerably (up to 150%) better than in TreadMarks.

1 Introduction

This paper addresses support for distributed sharing of objects in strongly-typed programming languages. Typing must be sufficiently strong such that it allows an unambiguous determination of whether a location contains an object reference or not. In addition, in the case of a reference, the type and size of its referent must be known. Many modern languages fall under this category, including Java and Modula-3. Unlike purely object-oriented languages, like e.g., Orca [3, 2], we do not restrict access to occur solely through method invocation. Direct access through a reference to object data is supported.

The key insight in this paper is that *type information allows 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 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 pa-

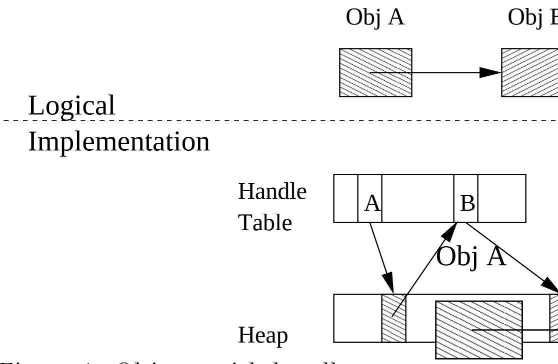

Figure 1: Objects with handles.

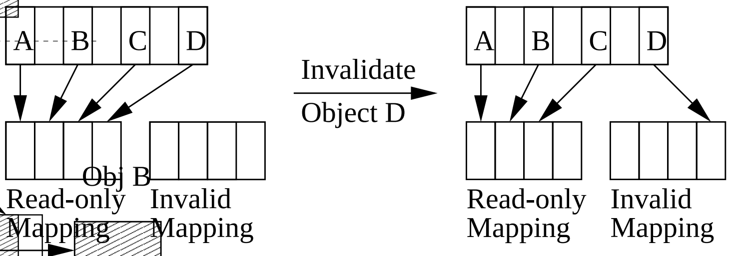

Figure 3: Access detection using the handle pointers.

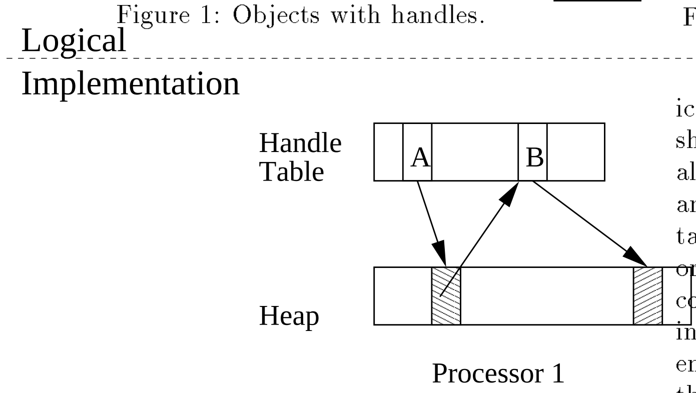

Figure 2: Shared objects identified by unique OIDs.

per, 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 [11], 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 misses, messages, and – in the case of TreadMarks – false sharing. In contrast, in SkidMarks, 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 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.

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

3. The performance of garbage-collected applications is considerably (up to 150%) better than in TreadMarks.

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 [6], 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 coarse-grained, fine-grained, and garbage-collected 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.

Objects are considered 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 of objects may, however, have multiple concurrent writers. This is in particular true for arrays of scalars. 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

SkidMarks 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 SkidMarks, 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, SkidMarks 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. Therefore, 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

SkidMarks 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. SkidMarks 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 SkidMarks, but the similarity ends there. In MultiView, each object resides in its own *vpage*, which is the size of a VM page. Different

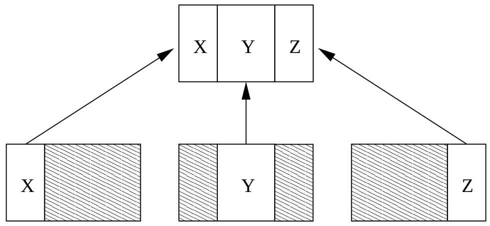

Figure 4: Multiview: one vpage for each object.

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 SkidMarks, namely per-object access and write detection. The MultiView method requires one virtual memory mapping per object, while the SkidMarks method requires only three mappings per page, resulting in considerably less address space consumption and pressure on the TLB. Also, SkidMarks 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.

[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.

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 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 SkidMarks on one hand, and TreadMarks and Shasta on the other hand.

SkidMarks 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 SkidMarks 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 for Coarse-grained Applications

The extra indirection is a source of concern for applications that access large arrays, because the extra indirection for each array access may cause significant overhead, without any gain in return from better support for fine-grained sharing. Here again, however, we can take advantage of the type information in the language, this time by exploiting the type information at compilation time.

Consider, for example, 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. In other words, an array of pointers to an array of the scalar type, i.e.,

```
    scalar_type **a;
```

This program performs a regular traversal of the array with a nested for loop. First, consider the TreadMarks program. Here, the accesses to the two-dimensional array take the form of

```
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[2]. In a strongly typed-language, however, `a`, `a[i]` and `a[i][j]` are of different types, and therefore the compiler 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 SkidMarks program the original program takes the form of

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

which, in a strongly typed language can be similarly transformed to

[2] Some C compilers support a `#pragma` or command line option that enables the programmer to make assertions about pointer aliasing that the compiler cannot determine automatically.

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

While offering much improvement, this transformation still leaves the SkidMarks 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 SkidMarks 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.

One alternative is to dynamically re-write the value of `p` during execution. Another alternative, which is the one we use, is to perform a slightly different transformation, 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, resulting in

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

This optimization is essentially loop invariant analysis, and can be carried out with well-understood compiler technology. The only twist is that a "cached" handle, such as `p` or `q` above, cannot be reused across a synchronization point.

5 Methodology

Our performance evaluation seeks to substantiate the following claims:

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

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 strongly-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 SkidMarks, 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 SkidMarks. 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 2, 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 SkidMarks programs. We report results with and without these optimizations present.

For comparison 3, we have implemented two methods of distributed garbage collection on both TreadMarks and SkidMarks that are representative of those in common use. The first method is based on mark-and-sweep and the second method is based on copying. In the remainder of this section, we describe these methods and their implementations on SkidMarks and TreadMarks.

5.1 Distributed Garbage Collection Methods

Plainfosse and Shapiro [10] survey various solutions to the distributed garbage collection problem. The fundamental factor that distinguishes *distributed* garbage collection from single-machine garbage collection is that an object may reside on one node while all references to it are within objects residing on other nodes. To address this situation, all of our garbage collectors maintain a table of imported references and of exported references. When a message containing an object is sent or received, it is checked for references. On the sending side, any references to locally created objects are entered into the export table; and on the receiving side, all references are entered into the import table. In other words, if a message contains object A and object A references object B that was created by the sender, object B is entered into the sender's export table and the receiver's import table. When a node later determines that any objects containing an imported reference are dead, the node notifies the object's creator that the reference is no longer used and removes it from its import table.

In general, the object creator's task of determining whether any references to the object still exist on some node is complicated by the fact that a node can send a reference to the object to another node without the creator's knowledge. For example, consider a system consisting of three nodes: node N_1 creates an object and exports a reference to the object to nodes N_2 and N_3. When node N_2 no longer possesses a reference to the object, it notifies node N_1. Then node N_3 passes a reference to the object to node N_2 and removes its own references to the object. Even though node N_1 has received notifications from both nodes N_2 and N_3, it must recognize that there is still a valid reference to the object. We use a technique called *weighted reference counting* [13, 14, 4] to solve this problem. It works as follows:

- A reference is assigned a predetermined weight when it is first exported by its creator.
- Whenever a reference is duplicated across a node boundary, the weight of the reference is equally divided between the local reference and the new remote reference, so that the sum of the weights remains constant.
- When a reference is no longer used and is sent back to its creator, its weight is also returned. When the sum of the returned weights equals the original weight assigned at the reference's creation, the creator node is sure that no one needs this reference.

The primary limitation of this technique is that it cannot trivially reclaim cycles that span nodes. This does not, however, become a factor in our evaluation. (See Plainfosse and Shapiro [10] for a discussion of how to handle this situation.)

On SkidMarks, both garbage collectors are responsible for reclaiming unused handles in addition to the underlying storage. Handles are managed in the same way by both garbage collectors: There is a list of free handles and a list of allocated handles. During garbage collection, live handles are moved from the old allocated list to a new allocated list. In the mark-and-sweep collector, this occurs when an object is marked; and in the copying collector, this occurs when an object is copied from the old space to the new space. At the end of garbage collection, any handles remaining in the old allocated list are unused, so the old allocated list is appended to the free list.

The mark-and-sweep collectors for TreadMarks and SkidMarks are identical apart from the handle management required in the SkidMarks version. The set of live objects is determined during the mark phase and the storage associated with dead objects is reclaimed during the sweep phase. The mark phase performs a depth-first traversal of the directed graph formed by the objects (vertices) and references (edges), marking each encountered object. The traversal starts at the set of "root" references and the exported references. During this traversal, memory coherence is disabled, eliminating a vast amount of potential communication. Roughly speaking, the only ill consequence of using an out of date copy of an object is that it may delay the reclamation of storage. (See Ferreira and Shapiro [7] for a detailed explanation.) The sweep phase has two parts. First, each locally created object is examined, and the storage used by the unmarked ones is returned to the free pool. Second, each imported reference is examined. If the object that it references was not marked, the object's creator is notified that the imported reference is no longer used[3].

The copying collectors for TreadMarks and SkidMarks differ somewhat, owing to the existence of handles and the single-writer protocol in SkidMarks. In TreadMarks, only the last writer of a page is allowed to copy objects within the page. When a processor wants to start a garbage collection, it initiates a barrier-like operation that suspends the DSM system. After the barrier, the last writer of every page is known to every processor. In the case of multiple writers to a page, an arbitration algorithm in the barrier will designate a single processor as the last writer and bring the page on the designated node up to date. After the barrier, each node starts a depth-first traversal from the "root" references and the exported references. An object is copied and scanned only if the node is its last writer. When an object is moved, a forwarding pointer to its new location is written into its old address. References to the moved object are updated only if the node is also the last writer of the pages that contain them. After the traversal, imported references still pointing to the old address ranges (the fromspace) are removed from the import table and sent back to the object's creator. The fromspace cannot, however, be immediately reclaimed. A node simply sets the protection of the fromspace to no-access. When a remote node follows a stale pointer and accesses a page in the fromspace, it will fetch the up to date copy of the page, find the forwarding pointer, and update the reference which has caused the fault. The fault handler does not change the protection of the page so that other stale references to the moved objects will be caught when they are followed. The fromspace is reclaimed when all stale references have been updated on all nodes.

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

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 have chosen 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.

Second, we use two 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.

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 SkidMarks calls from the applications

[3] The fact that an object is marked on some node does not necessarily imply that a copy of the object exists on that node, only that a reference to the object exists.

and for SkidsMarks 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, we use Tree, a synthetic application, to exercise the mark-and-sweep and copying collectors. Tree creates and then updates a binary search tree. Each internal node of the tree contains a key as well as the pointers to the node's two children. A leaf node contains a key and a list of data items. The data items are of variable size.

The program works as follows. In the initialization phase, the master node takes a list of <item, index> pairs and builds the tree. The main loop of the program consists of two phases: the update phase, in which the master node may insert new data items to the tree or delete existing data items from the tree; and the search phase, in which each node is assigned a list of indices to search. Once a data item is found in the search phase, some processing is done which involves the allocation of some temporary objects.

Table 2 lists the sequential execution times for Tree running under the mark-and-sweep and copying collectors on TreadMarks and SkidMarks. It also lists the time spent in the memory allocator/garbage collector. In our experiments, the mark-and-sweep collectors use a single 8M byte heap, while the copying collectors use two 4M byte semispaces. The tree on average has 24K nodes totaling 1M bytes. The master node allocates 30M bytes of memory during the execution.

SkidMarks slightly underperforms TreadMarks on 1 processor, mainly because of the extra overheads in memory management. SkidMarks incurs extra overheads because it has to update the handle table entry whenever an object is created, deleted, or moved. On 1 processor, SkidMarks adds 2% overhead to the mark-and-sweep collector and 7% overhead to the copying collector.

8 Overall Results

8.1 Fine-grained Applications

Table 3 details statistics from the execution of Barnes-Hut and Water-Spatial on 32 processors for a small and a large data set.

We derive the following conclusions from the results. 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 at a small number of processors and for a small data set, the benefits of the handle-based implementation are larger than the cost of the extra indirection. For example, for Barnes-Hut with 32K bodies, SkidMarks outperforms TreadMarks by 22% and 29% on 8 and 16 processors, respectively. For Water-Spatial with 4K molecules, SkidMarks outperforms TreadMarks by 25% and 62% on 8 and 16 processors, respectively. Third, for larger numbers of processors and larger data sets, the benefits of the handle-based implementation grow considerably larger. For example, for Barnes-Hut with 128K bodies, SkidMarks outperforms TreadMarks by 98% on 32 processors. For Water-Spatial with 32K molecules, SkidMarks outperforms TreadMarks by 51% on 32 processors. The reason that this gap between the two systems is smaller than with 4K molecules on 16 processors is from somewhat reduced false sharing in TreadMarks when scaling up the number of molecules in Water-Spatial.

The reasons behind these improvements can be seen in Table 3. This table shows for both implementations, the number of messages exchanged, the number of message "rounds", the amount of data, and the average amount of shared data allocated on one processor. The benefits of lazy object allocation for these applications are quite clear: the memory footprint of SkidMarks is considerably smaller than that of TreadMarks, and, for a given data set, gets smaller as the number of processors is increased.

Table 3 also shows a substantial reduction in the amount of data sent for SkidMarks, as a result of the reduction in false sharing. 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 factors of 1.3 and 4.9 for the two applications, respectively.

8.2 Coarse-grained Applications

Table 4 details statistics from the execution of SOR and Water-N-Squared on 32 processors for a small and a large data set.

For these two coarse-grained applications, SkidMarks performs comparably to TreadMarks. In fact, SkidMarks performs almost identically as TreadMarks for SOR; both send the same amount of data and number of messages, and allocate the same amount of physical memory on each processor.

Water-N-Squared contains migratory data, and diff accumulation occurs in TreadMarks. On the other hand, diffs are only one-fifth to one-fourth the size of

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.

Tree	Mark–Sweep		Copying	
	Tmk	Skid	Tmk	Skid
Time	18.99	19.04	17.42	17.50
Alloc and GC time	1.00	1.02	0.57	0.61

Table 2: Detailed statistics for TreadMarks and SkidMarks on 1 processor for Tree.

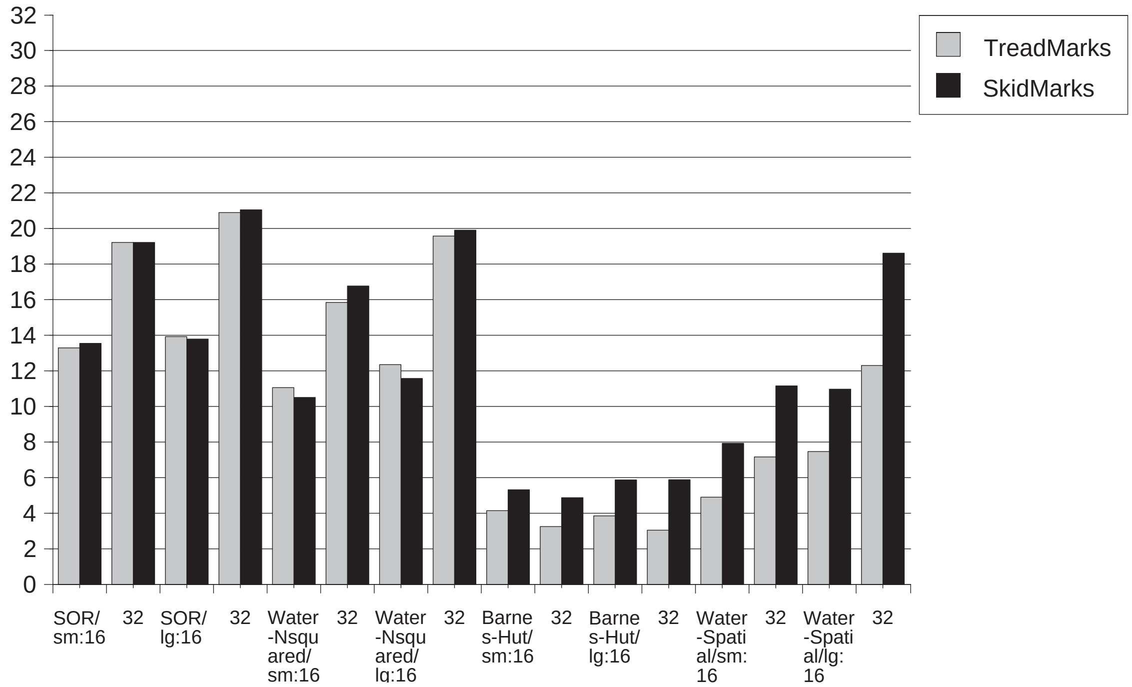

Figure 5: Speedup comparison between TreadMarks and SkidMarks.

Application	Barnes-Hut/sm		Barnes-Hut/lg		Water-Spatial/sm		Water-Spatial/lg	
	Tmk	Skid	Tmk	Skid	Tmk	Skid	Tmk	Skid
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 SkidMarks 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 SkidMarks, 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.

a page, thus it takes a long migration path for diff accumulation to lose to sending whole objects. On 16 processors, diff accumulation does not outweigh sending whole objects, and SkidMarks is up to 7% slower than TreadMarks from sending 14% more data. On 32 processors, SkidMarks sends 14% less data than TreadMarks and is up to 6% faster than TreadMarks.

8.3 Garbage Collected Applications

Table 5 details the statistics for the execution of Tree using both the mark-and-sweep collector and the copying collector on 32 processors.

On 32 processors, the memory management cost is higher for both collectors on either system. For TreadMarks, the reason for the increased cost is that the object movement or the memory allocation may trigger protection faults. For SkidMarks, it is because some data structures that are not used in the sequential execution, e.g., the per page object list, must be maintained. The difference in memory management between the two mark-and-sweep collectors remain below 2%. However, SkidMarks' copying collector outperforms its TreadMarks counterpart by more than 25%.

On DSM systems, the effect of the garbage collectors extends beyond the memory management cost and affects the total execution time of the programs. On 32 processors, SkidMarks plus copying GC outperforms its TreadMarks counterpart by almost 75%, and SkidMarks outperforms TreadMarks by 150% when the mark-and-sweep collector is used. The cause for the large performance difference is the spatial locality and false sharing.

In the test program, with the frequent changes to the tree, the spatial locality of the tree decreases over time, and the tree may span across many pages. Since each page requires a separate message to communicate in TreadMarks, the decreased spatial locality increases the number of messages. Furthermore, the tree nodes that have been cut off may be collected and reused for the short-lived temporary objects, causing false sharing. With the mark-and-sweep collector, TreadMarks sends 200% more data and 300% more messages than SkidMarks.

The copying collector in TreadMarks improves the spatial locality and largely solves the above problems. However, the object copying and the reference updates made by one node will be propagated to remote nodes by TreadMarks just like the normal writes by the program. This increases the amount of data communicated between the nodes. From the table we can see that, compared with mark-and-sweep, the copying collector on TreadMarks reduces the number of messages by more than 60%, but fails to reduce the total message size.

Compared with TreadMarks, SkidMarks is much less sensitive to the spatial locality because space is only allocated for an object when it is accessed. Therefore, two objects that are in separate pages on one processor may be put in the same page by another processor. In addition, SkidMarks does not suffer from false sharing. The measurement shows that even with the mark-and-sweep collector, SkidMarks still sends fewer messages and less data than TreadMarks with either collector. The copying collector further reduces the communication cost in SkidMarks, although its benefit is not significant.

9 Effects of the Various Optimizations

To achieve the results described in the previous section, various optimizations were used in SkidMarks. 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 Skidmarks without that optimization to the fully-optimized system. Figure 6 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.

9.1 Lazy Object Allocation

Table 6 shows that without lazy object allocation, SkidMarks sends 57% and 68% more messages and runs 13% and 18% slower than SkidMarks 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-

Application	SOR/sm		SOR/lg		Water-Nsquare/sm		Water-Nsquare/lg	
	Tmk	Skid	Tmk	Skid	Tmk	Skid	Tmk	Skid
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 4: Detailed statistics for TreadMarks and SkidMarks on 32 processors for coarse-grained applications SOR and Water-N-Squared.

Tree	Mark-&-Sweep		Copying	
	Tmk	Skid	Tmk	Skid
Time	9.99	3.95	6.68	3.82
Alloc and GC time	1.11	1.13	0.95	0.74
Data (MB)	42.7	13.9	44.9	13.9
Overlapped data requests	239000	57400	86500	56600

Table 5: Detailed statistics for TreadMarks and SkidMarks on 32 processors for Tree.

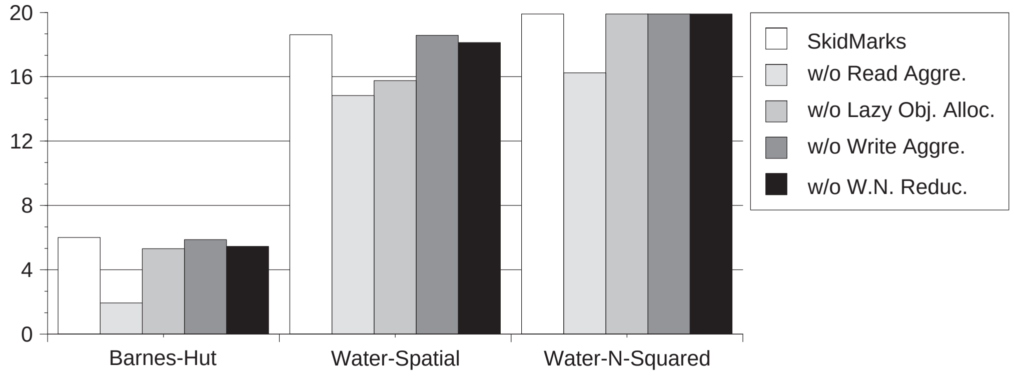

Figure 6: Speedup comparison between SkidMarks and SkidMarks without each of the optimizations on 32 processors.

Application		Skid	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 SkidMarks and SkidMarks without each of the optimizations on 32 processors for Barnes-Hut, Water-Spatial, and Water-N-Squared.

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, SkidMarks 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, SkidMarks without read aggregation is 310%, 24%, and 22% slower than SkidMarks for these three applications.

Intuitively, the potential problem with read aggregation is that SkidMarks may fetch more objects than necessary. SkidMarks without read aggregation, however, only fetches accessed or necessary objects. Thus, by looking at the difference between SkidMarks with and without read aggregation, we can determine the amount of unnecessary data communicated. Table 6 shows that SkidMarks without read aggregation sends almost the same amount of data as fully-optimized SkidMarks 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, SkidMarks is one second or 2.2% faster for Barnes-Hut than SkidMarks 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.

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

We have already compared our work extensively to TreadMarks [1] and Shasta [11], using them as examples of coarse-grained and fine-grained DSM systems. Qualitatively similar comparisons can be made with other DSM systems [9, 5, 16]. We have also compared our work to the MultiView approach used in Millipede [8].

Dwarkadas et al. [6] compare Cashmere, a coarse-grained system, and Shasta running on an identical platform – a cluster of four four-way AlphaServers connected by a Memory Channel network. Both systems are designed to leverage the Memory Channel network and take advantage of the hardware shared memory within each SMP node. In general, Cashmere outperforms Shasta on coarse-grained applications and Shasta outperforms Cashmere on fine-grained applications.

All of the applications used in this paper, except for the synthetic GC application, were also used in their paper. Shasta only outperformed Cashmere for one of these four applications, the fine-grained Barnes-Hut/lg by 350%. Cashmere outperformed Shasta by 50% and 10% for the coarse-grained applications, SOR/sm and Water-N-Squared/4K, respectively. The only surprise is that Cashmere equals Shasta on the fine-grained application Water-Spatial. (The difference is less than 3% in favor of Cashmere.) They attribute this result to the run-time overhead of the cache line validity checks in Shasta. In contrast, SkidMarks outperforms TreadMarks by 62% on Water-Spatial. We attribute this result to lazy object allocation, which is not possible in Shasta, and read aggregation.

11 Conclusions

In this paper, we have presented a new runtime system, SkidMarks, for supporting distributed sharing of objects in strongly-typed programming languages. The key insight is that *type information allows efficient and transparent sharing of data with both fine-grained and coarse-grained access patterns.* Using the runtime type information, SkidMarks efficiently implements a shared space of objects on distributed systems. It uses lazy release consistency and a single-writer protocol. Like previous fine-grain systems, SkidMarks improves performance of fine-grained applications by eliminating false sharing. Furthermore, it incorporates a number of novel techniques that further improve the performance. These include lazy object allocation to improve data locality, read aggregation to reduce the number of message rounds for updating shared objects, write aggregation to reduce the number of write faults, and finally, a novel write notice reduction technique which dramatically reduces consistency data.

Our performance evaluation on a cluster of 32 Pentium-II processors connected with a 100Mbps Ethernet substantiates the following claims:

1. 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.

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

Application	Tmk/pref.		Tmk/nopref.		Skid/pref.		Skid/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.

3. The performance of garbage-collected applications is considerably (up to 150%) better than in TreadMarks.

We are currently pursuing several future directions. The handle-based SkidMarks implementation introduces an extra level of indirection. With compile-time optimization, the overhead seems insignificant for scientific applications. It may, however, become larger for a different class of applications. We are working on a handle-less implementation that will eliminate the indirection overhead.

The decoupling of shared object space from the underlying address space enabled lazy object allocation, which has been shown to significantly improve performance of irregular applications. It further enables objects to be relocated independently and dynamically on different processors. As shown by Barnes-Hut, for applications where objects migrate, the migrated objects can create holes in a physical page, which in turn can cause read aggregation to fetch unnecessary objects. We are exploring ways of dynamically relocating objects to improve data locality for applications with migratory objects.

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] H. Bal, M. Kaashoek, and A. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, Mar. 1992.

[4] 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.

[5] 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.

[6] 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.

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

[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] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov. 1989.

[10] D. Plainfosse and M. Shapiro. A survey of distributed garbage collection techniques. In *International Workshop on Memory Management*, pages 211–249, Sept. 1995.

[11] 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.

[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.

[16] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation*, pages 75–88, nov 1996.