Performance Evaluation of the Orca Shared Object System *

HENRI E. BAL RAOUL BHOEDJANG RUTGER HOFMAN
CERIEL JACOBS KOEN LANGENDOEN TIM RÜHL

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

M. FRANS KAASHOEK

M.I.T. Laboratory for Computer Science
Cambridge, MA

ABSTRACT

Orca is a portable, object-based distributed shared memory system. This paper studies and evaluates the design choices made in the Orca system and compares Orca with other DSMs. The paper gives a quantitative analysis of Orca's coherence protocol (based on write-updates with function shipping), the totally-ordered group communication protocol, the strategy for object placement, and the all-software, user-space architecture. Performance measurements for ten parallel applications illustrate the tradeoffs made in the design of Orca, and also show that essentially the right design decisions have been made. A write-update protocol with function shipping is effective for Orca, especially since it is used in combination with techniques that avoid replicating objects that have a low read/write ratio. The overhead of totally-ordered group communication on application performance is low. The Orca system is able to make near-optimal decisions for object placement and replication.

In addition, the paper compares the performance of Orca with that of a page-based DSM (TreadMarks) and another object-based DSM (CRL). It also analyses the communication overhead of the DSMs for several applications. All performance measurements are done on a 32-node Pentium Pro cluster with Myrinet and Fast Ethernet networks. The results show that the Orca programs send fewer messages and less data than the TreadMarks and CRL programs and also obtain better speedups.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming, Parallel programming*; D.3.2. [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages*; D.3.4 [**Programming Languages**]: Processors—*Compilers; Run-time environments*

General Terms: Languages, Design, Performance

Additional Key Words and Phrases: distributed shared memory, parallel processing, portability

---

* This research was supported in part by the Netherlands organization for scientific research (N.W.O.) under grant 125-30-10.

1. INTRODUCTION

During the past decade, many distributed shared memory (DSM) systems have been designed and implemented. The key idea of DSM is to hide the communication substrate from the programmer and provide a programming model based on shared data rather than message passing. Li laid the foundation for this work by building a system that simulates physical shared memory on top of distributed-memory hardware [30]. Many other, more advanced DSMs have subsequently been built [6, 7, 11, 14, 17, 23, 26, 28, 36, 37, 40, 43, 44, 45]. The DSM system studied in this paper is Orca. Orca's programming model was designed and implemented prior to most other DSM systems [5] and differs from other DSM models by being *object-based* and *language-based*. Orca encapsulates shared data in objects, and allows the programmer to define operations on objects, using abstract data types. In contrast, many other DSMs use physical entities such as pages or memory regions as the unit of sharing. This shared object model is supported by the Orca language, which was designed specifically for parallel programming on distributed-memory systems. This paper focuses on the implementation and performance of Orca's shared object system and compares it to other DSM systems.

Orca's programming model is simple, but implementing shared objects efficiently is challenging. We have designed several implementation techniques for shared objects and used these techniques in a new Orca system that is described in this paper. Unlike an earlier implementation [4], the current system is highly portable. It has been implemented on a variety of multicomputers and workstation clusters and has been used for dozens of parallel applications. The goal of this paper is to provide a detailed performance study of the current Orca system and to give a quantitative performance comparison between Orca and other DSM systems. This analysis is interesting, because during the design of the current Orca system, we have made several decisions that are markedly different from those taken in other DSM systems. Several of these decisions are due to the usage of an object-based (rather than page-based) DSM model. In the paper, we will analyze and evaluate the implementation techniques for shared objects and discuss the advantages and disadvantages of object-based DSM.

To explain how Orca differs from other DSM systems, we briefly summarize the most important aspects of the Orca system (see also Table 1). Like other DSMs, Orca migrates and replicates shared data (objects), but its coherence protocol for shared data differs from that of most other DSMs. First of all, the Orca system uses an *update* protocol for implementing write operations, whereas most DSMs use an invalidation protocol. Objects are updated using *function shipping*: the operation and its parameters are sent to all machines containing a copy of the object and the operation is applied to the local copies. To update replicated objects in a coherent way, the operation is sent using *totally-ordered group communication*, so all updates are executed in the same order at all machines. This primitive is seldom used in DSMs, due to its potentially high cost. The advantage of this primitive is that it greatly simplifies the implementation of the update protocol. We have developed several protocols that make its cost acceptable for the Orca system.

Another interesting aspect of the Orca system is the way in which it determines the location for each object. The system replicates only objects that are expected to have a high read/write ratio, thus reducing the overhead of updates. The compiler computes regular expressions (patterns) that give a high-level description of how shared objects are accessed. The runtime system uses this information as a hint about how objects are going to be used. In addition, it maintains runtime statistics about object usage. Based on all this information, the runtime system decides on which machines to place (copies of) shared objects.

Issue	Design decision
Coherence protocol	Write-update protocol based on function shipping and totally-ordered group communication
Object placement	Compile-time heuristics and runtime statistics
Portability	Layered approach based on a virtual machine
Software architecture	All-software, user-space DSM

Table 1: Key issues in the design of the Orca system.

The structure of the Orca system also differs from that of many other DSMs. The current Orca system was designed from scratch to provide portability and flexibility. To achieve portability, we use a layered approach. The system contains several layers, and the machine-specific parts are isolated in the lowest layer. The compiler and the runtime system, for example, are fully machine-independent. The runtime system is implemented on top of a virtual machine called *Panda*, which provides the communication and multi-tasking primitives needed by the runtime system. Porting the Orca system to a new machine comes down to porting Panda. The Panda layer can be configured statically to match the underlying system. If the operating system or hardware provides certain functionality that is useful to Panda (e.g., reliable communication), Panda can be configured to make use of this.

The Orca system is implemented entirely in software. It requires the operating system (or hardware) to provide only basic communication primitives. In the classification of [23], Orca thus is an all-software DSM. Furthermore, the Orca system runs mostly or entirely in user space, depending on the underlying platform. Other systems rely on virtual memory support [14, 30], which is not uniformly supported on parallel computers and often is expensive to use [34]. The disadvantage of an all-software DSM is a potential performance penalty (e.g., access checks are done in software), but the advantage is increased flexibility. The Orca system exploits this flexibility to implement several important optimizations (described later).

In the rest of the paper, we will give a more detailed description of the Orca system and present a quantitative performance analysis. Based on ten Orca applications, we analyze the performance of several implementation techniques. The analysis shows that (at least for the given applications and target platform) we have essentially made the right decisions during the design of the system. For all ten Orca applications, the write-update protocol is the right choice, despite the common wisdom that such protocols result in a high communication overhead. For Orca applications, the communication overhead of write-updates is relatively low, because the read/write ratio for shared objects often is high and because the system does not replicate objects for which the ratio is low. Another nonintuitive performance result is that the overhead of totally-ordered broadcasting on application performance is small (usually below 1%), because the Orca system uses an efficient broadcast scheme and avoids replicating objects that are written frequently. A key issue thus is to determine which objects to replicate. The performance study shows that the Orca system is able to make good decisions about object replication, without any help from the programmer.

In addition to this quantitative analysis of the Orca system, we compare the performance of the system with that of two other, more recent, DSMs: TreadMarks and CRL. TreadMarks is a page-based DSM that uses several advanced optimizations [14]. CRL is an object-based DSM, but it differs significantly from Orca by

being a library instead of a language and by using a directory-based invalidation protocol [23]. We analyze the speedups and communication behavior of three applications for these systems, showing that the Orca applications obtain better performance than the TreadMarks and CRL applications. The performance comparisons are done on identical hardware: a collection of 32 Pentium Pros connected by Myrinet [9] and Fast Ethernet. This parallel system is substantially larger than those used for other published performance studies of TreadMarks [11, 35], so we provide new results on the scalability of TreadMarks. These results show that on a fast switched network (Myrinet), TreadMarks scales reasonably well (with efficiencies of 50% or higher) for two of the three applications, while the scalability for the most challenging application (Barnes-Hut) is poor. On a hubbed Fast Ethernet, TreadMarks does not scale, due to the high communication volume and contention on the network.

The paper thus gives many insights in the implementation techniques for shared objects and evaluates them thoroughly. These contributions will also be useful to other DSM systems. The rest of this paper is organized as follows. In Section 2, we describe the Orca programming model and the Orca system and compare them qualitatively to other models and systems. In Section 3, we give a quantitative analysis of the implementation techniques for shared objects. In Section 4, we compare the performance of Orca with that of TreadMarks and CRL, using three application programs. In Section 5, we look at related work. Finally, in Section 6 we present our conclusions.

2. OVERVIEW OF THE ORCA SHARED OBJECT SYSTEM

This section first briefly describes the Orca programming model and makes a comparison with other DSM models. Next, it describes the implementation of Orca. We refer to the implementation of Orca as the ‘‘Orca system.’’ The system consists of three parts: a compiler, a runtime system, and the Panda portability layer. We motivate the choices we have made during the design of the system, and compare our techniques with alternative designs used in other DSM systems.

2.1. The Orca programming model

In shared memory systems and in page-based (or region-based) DSMs, processes communicate by reading and writing memory words. To synchronize processes, the programmer must use mutual exclusion primitives designed for shared memory, such as locks and semaphores. Orca’s model, on the other hand, is based on high-level operations on shared data structures and on implicit synchronization, which is integrated into the model.

The starting point in the Orca model is to encapsulate shared data in *objects*, which are manipulated through operations of an *abstract data type*. An object may contain any number of internal variables and arbitrarily complex data structures (e.g., lists and graphs). A key idea in Orca’s model is to make each operation on an object *atomic*, without requiring the programmer to use locks. All operations on an object are executed without interfering with each other. Each operation is applied to a single object, but within this object the operation can execute arbitrarily complex code using the object’s data. Objects in Orca are passive entities: they only contain data. Parallel execution is expressed through (dynamically created) processes.

The shared object model resembles the use of monitors. Both shared objects and monitors are based on abstract data types and for both models mutual exclusion synchronization is done by the system instead of the

programmer. For condition synchronization, however, Orca uses a higher-level mechanism [4], based on Dijkstra's guarded commands [15]. This mechanism avoids the use of explicit *wait* and *signal* calls that are used by monitors to suspend and resume processes. Another difference between shared objects and monitors is that monitors are designed for systems with shared memory. Shared objects can be implemented efficiently without shared memory, for example by replicating them, as will be discussed extensively in the paper.

Shared objects thus provide a high-level model for parallel programming. The model is similar to programming with monitors on shared memory, but it uses object-based techniques and it is suitable for systems that do not have shared memory. It integrates sharing of data, mutual exclusion synchronization, and condition synchronization in a clean way.

2.2. Comparison with other DSM models

We discuss Orca's programming model in somewhat more detail here, by comparing it against other DSM models. Although the focus of this paper is on the implementation of Orca rather than its programming model, this discussion is useful to understand several decisions in the Orca implementation, described in the rest of the paper.

The Orca model is characterized by four properties that are discussed below. Other DSMs also have some of these properties, but to the best of our knowledge the combination is unique to Orca.

Property 1: Shared data structures are encapsulated in shared objects, which can only be accessed through (user-defined) operations.

The advantages of this decision are that shared data are easily distinguished from nonshared data and that shared data accesses can easily be identified at compile-time. An operation invocation has a simple syntax, for example:

```
Q$enqueue(5);
```

where *Q* is a variable of an abstract data type (i.e., an object) and *enqueue* is one of the operations defined by this type. Another advantage of this property is that the risk of false sharing is reduced (but not eliminated), because typically only data that logically belong together are stored in the same object. With page-based DSMs, unrelated data may be stored on the same page, resulting in false sharing.

The disadvantage of Property 1 is that, since only objects can be shared, existing shared memory programs require restructuring before they can be expressed in the model. In contrast, flat address space DSMs like Shasta [43] and TreadMarks can run existing shared memory programs with few or no modifications. Orca requires (and enforces) that all shared data be accessed through operations.

In comparison, most DSM systems also make a distinction between shared and nonshared memory. TreadMarks reserves part of the address space of each process for shared memory. Shared memory is allocated with special versions of *sbrk* and *malloc*. Shasta automatically makes all dynamically allocated data shared; static and stack data are not shared. In CRL, shared data must be put in *regions*, which are similar to Orca's objects, except that regions are required to be contiguous and object memory is not. CRL provides library primitives to start and end operations on regions; accesses to shared regions outside an operation are without guarantees on consistency.

Property 2: Orca is language-based.

Although the Orca runtime system can be (and has been) used from a traditional language (e.g., ANSI C), it is primarily designed for programs written in the Orca language. An important advantage of the Orca language is that it is designed especially for parallel programming on distributed-memory systems, allowing new applications to be written cleanly. This claim is supported by an extensive study on the ease-of-use of Orca, which is described elsewhere [48]. Another advantage of using a language is that the implementation can benefit from compiler support, such as the generation of access patterns mentioned earlier and discussed in detail later.

The disadvantage of using a new language is that it is hard to reuse existing code written in other languages. It is possible to invoke foreign-language functions from Orca using stubs, but this will only work correctly for functions that do not use global variables or complicated data types. Another disadvantage is that programmers have to learn a new language. Fortunately, our experience so far indicates that Orca is very easy to learn [48].

Most other DSM systems are based on a library that is linked with programs written in traditional languages. Other systems (e.g., Mentat [19]) are based on extensions to existing languages.

Property 3: Orca integrates synchronization and data accesses.

Both mutual exclusion synchronization and condition synchronization are integrated in the model. An important advantage is that programmers do not have to use explicit synchronization primitives, which significantly eases parallel programming. A disadvantage is that implicit synchronization is somewhat less flexible than explicit locking. For example, it is difficult to write atomic operations on *multiple* objects in Orca, which would be useful for some applications [48]. The restriction to support only operations on single objects was taken to allow an efficient implementation. An extension of the Orca model to support atomic functions on multiple objects is described in [41].

Most other DSMs decouple synchronization and data accesses. TreadMarks, for example, provides lock variables and barriers. DiSOM's programming model is based on shared objects, but it differs from Orca by using explicit synchronization primitives [11]. In CRL, the programmer inserts calls to library primitives to start and end an operation on a shared region; the CRL library then locks and unlocks the region. CRL thus allows synchronization and data accesses to be coupled, but the correct usage is not enforced, since CRL does not use a compiler. As with explicit lock primitives, the programmer can incorrectly access shared data, resulting in race conditions.

Property 4: Orca's shared memory model is sequentially consistent.

Sequential consistency is a simple, easy to understand semantics. The disadvantage of sequential consistency is that it is harder to implement efficiently than weaker forms of consistency. TreadMarks and Midway [7], for example, support lazy release consistency and entry consistency, which allow coherence messages to be buffered until certain synchronization points. If the programmer properly synchronizes access to shared data, these consistency models guarantee the same program behavior as sequential consistency. For incorrectly synchronized programs, however, the behavior need not be sequentially consistent, which makes debugging harder.

The Orca model has resemblances to entry consistency. Entire operations on shared objects are executed sequentially consistently, but the individual reads and writes to memory words within an operation are not

made visible to other processes until the operation completes, since operations are executed atomically. In terms of individual memory accesses, the Orca model thus is similar to entry consistency, but without requiring the programmer to associate lock variables with objects. CRL has a similar coherence model [23].

2.3. Implementation of shared objects

We now turn our attention to the main topic of this paper, which is implementation techniques for Orca's shared objects. We describe four design issues in detail: the coherence protocol for replicated objects, the object placement strategy, the layered approach based on a portability layer, and the all-software, user-space architecture. We will also discuss the advantages and disadvantages of the design choices we have made, and compare our techniques with those of other DSM systems. In Section 3, we will give a quantitative analysis of our techniques.

2.3.1. Coherence protocol for replicated objects

The Orca system replicates shared objects that are read frequently. The advantage of replication is that read-only operations (which are recognized by the Orca compiler) can be executed locally, without doing any communication. The problem, however, is how to implement write operations, which modify the object. The most important choice is whether to use a write-invalidate protocol or a write-update protocol.

An invalidation protocol invalidates all copies of the shared data and stores the data only on the machine that issued the write operation. Most page-based DSMs use an invalidation protocol, because pages are often modified by doing many consecutive stores to different memory words of the page. For Orca, objects are not modified by individual store instructions, but by applying user-defined operations to the object. The copies of the object can be updated by transmitting either the new value of the object (data shipping) or the operation and its parameters (function shipping). Either way, the write operation requires only one communication event (a broadcast), no matter how complicated the operation is. Also, write-updates can exploit hardware broadcast, if available. For large objects, write-update has another important advantage: by using function-shipping, Orca avoids sending the entire object over the network. Instead, it only transfers the operation and its parameters, which usually contain far less data than the object. An invalidation protocol for Orca would have to transfer the entire object state over the network whenever a new copy is needed (unless techniques like fine-grain sharing [45] or page-diffing [14] would be applied inside objects, but this would make the implementation much more complicated).

An update protocol also has disadvantages. Multiple consecutive write operations by the same processor are inefficient and consume extra network bandwidth. With an invalidation protocol, only the first write operation causes communication. Also, writing a replicated object causes interrupts on all processors, resulting in CPU overhead to handle the interrupt and execute the operation.

The Orca system uses a write-update protocol and updates the copies of an object using function-shipping. We made this design choice, because for the shared object programming model the disadvantages of write-update are more than compensated by its advantages. Unlike with page-based DSM, multiple consecutive write-operations by the same processor (without intervening accesses from other processors) do not occur often in Orca programs. In many cases, programmers combine such write-operations in a single (logical) operation. Also, as explained later, the Orca system reduces the overhead of handling incoming update

messages by replicating only those objects that have a high read/write ratio. In Section 3, we will look in more detail at the access patterns of Orca applications, and show that a write-update protocol indeed is a better choice than write-invalidate for these applications.

A related design issue is how to keep all copies of a shared object coherent. Simply broadcasting all updates does not suffice, since update messages from different sources may arrive in different orders at the destination processors, resulting in inconsistencies [5]. To solve this problem, the Orca system uses *totally-ordered group communication* [24], which guarantees that all group messages are received by all processors in the same total order. If a write operation is invoked on a replicated object, the invoker broadcasts the operation using totally-ordered group communication and then blocks until the runtime system has processed the broadcast message (and all other messages that occur before it in the total ordering). When a broadcast message with an operation is received, the runtime system calls a procedure that performs the operation and, if the operation was issued by one of its local Orca processes, unblocks the process.

The main reason for using totally-ordered group communication is that it provides a simple mechanism for keeping all copies of an object consistent. In addition, the total ordering makes it relatively easy to implement group communication reliably on unreliable networks [24]. A potential disadvantage is the cost of this communication primitive. To explain this cost, we briefly discuss the group communication protocols Orca uses. The protocols use a centralized sequencer to order all messages. Each broadcast message contains a sequence number, which the receivers use to order messages and to check if they have missed a message. Depending on the underlying system and the size of the message, the Orca system uses one of three different mechanisms for the sequence numbers. The simplest approach is to let the sender ask the sequencer for the next sequence number (using two short messages), after which the sender broadcasts the message and the sequence number. Another approach is to let the sender transmit the entire message to the sequencer; the sequencer then adds the next sequence number and broadcasts the message. For large messages, a third approach is to let the sender broadcast the message (without a sequence number) and let the sequencer issue another (short) broadcast message containing the sequence number. The details of the protocols (including recovery from lost messages) are described elsewhere [24].

Each protocol thus requires one or two extra control messages to implement the total ordering. Another potential weakness of the protocols is that they use a centralized component (the sequencer), which may create contention. The sequencer machine needs to do little extra work, however, so it can handle many broadcast requests per second. Moreover, Orca programs will only generate a large number of broadcast messages if they invoke many write-operations on replicated objects. The Orca runtime system, however, prevents objects that are mainly written from being replicated, so this communication behavior seldom occurs.

Yet another problem with the sequencer-based protocols is that *all* processors that participate in the parallel program receive group messages, so a write operation on a replicated object will interrupt all machines involved in the program. For the large majority of Orca applications, replicated objects have a high degree of sharing, so data broadcast in update messages are subsequently read by most or all machines. In Section 4, we will study an exception to this rule: with the Barnes-Hut application, only part of the processors are interested in certain updates, so replicating the entire object on all machines is disadvantageous. Still, this disadvantage is compensated by the ability to broadcast data instead of having to send multiple point-to-point messages. So, totally-ordered group communication does have a certain cost, but it makes the higher-level coherence

protocols simple and therefore is our method of choice.

To summarize, Orca's coherence protocol for replicated objects is based on a write-update protocol that uses totally-ordered group communication and function shipping to update all copies of an object after a write operation. As stated before, most other DSMs use a different approach. Page-based DSMs typically use a write-invalidate protocol. CRL is an object-based DSM, but its implementation uses a directory-based write-invalidate protocol. TreadMarks uses a directory-based invalidation protocol, extended with a multiple-writer protocol. The DiSOM shared object system uses a write-update protocol, but the coherence protocols of Orca and DiSOM are different. DiSOM transfers the state of an object after it has been modified, whereas Orca uses function-shipping. The state transfers in DiSOM are piggybacked on the control messages that are used to implement distributed locks.

2.3.2. Object placement strategies

With the write-update protocol described above, read-only operations on a replicated object can be executed locally, but write operations are broadcast to all machines. If the read/write ratio of the object is low, replication is inefficient and it is better to store the object on a single machine. In this case, other machines can access the object doing remote object invocations. The Orca system supports both replicated objects and single-copy objects and allows objects to switch dynamically from one representation to another.

An important issue is how to determine whether a given object should be replicated and where to store objects that are not replicated. The programmer could be required to take these decisions, but this approach is undesirable, since the idea of DSM is to hide distribution from the programmer. Also, the optimal choice may be machine-dependent. The Orca system therefore makes the object placement decisions itself, without involvement from the programmer.

The system uses a heuristic algorithm to make the object placement decisions. To explain the algorithm, let us assume (unrealistically) that it is known in advance which operations every processor is going to execute on a given object. Let W_i be the number of write operations that processor *i* is going to execute and R_i the number of read operations. If the object would be replicated, every write operation would result in a broadcast while read operations would not cause communication. The number of broadcast messages can be computed by summing all W_i. If, on the other hand, the object is not replicated, all accesses (reads and writes) would cause communication, except when they are issued from the machine on which the object is stored. To minimize the communication overhead, the object is best stored on the machine that does the largest number of accesses (i.e., has the highest value of $W_i + R_i$). Let this be machine *j*. The number of remote object invocations can then be computed by summing the values of all W_i and R_i, except for i=j.

The system can thus determine how many broadcasts or remote object invocations would be needed in each case. The heuristic rule we use is to select the representation that is expected to give the lowest communication overhead. The overhead is estimated using the above calculations and the cost of the communication primitives. If the system decides not to replicate the object, it also knows the best location to store the object. Clearly, this algorithm is not optimal. For example, the algorithm either replicates an object on all machines that can access the object or it does not replicate the object at all. Also, the algorithm only takes into account the number of messages that will be sent, and not their sizes.

The Orca system tries to estimate the values of W_i and R_i using a combination of compile-time analysis

and runtime statistics. For each type of process defined in an Orca program, the compiler computes a pattern that describes how such processes use shared objects, using interprocedural analysis. The pattern is a regular expression with special symbols denoting sequencing, selection, and repetition. For example, the pattern:

```
A$W; [ B$R | {C$W} ]
```

indicates that the process will perform a write operation on object *A*, followed by either a read-only operation on object *B* or a repetition of write operations on object *C*. The compiler passes a summary of the access pattern to the RTS. This summary contains estimated values of how often the process is going to read and write its shared objects. The values are computed by the compiler using the heuristic that operations inside a repetition (loop) are executed more frequently than operations outside a repetition. Also, operations inside an **if**-statement are executed less frequently than operations outside a selection.

The RTS uses these compiler-generated values to estimate the accesses each processor will perform. In addition, the RTS counts how many read and write operations are actually issued by each processor on each object. This information is stored on one of the processors containing a copy of the object; this machine is called the *manager* of the object. The information maintained by the manager is shown in Figure 1. The information includes the dynamic counts of the number of read and write operations and the read and write estimates generated by the compiler. These values are summed over all Orca processes on all processors.

The sums of the compiler-generated values are updated whenever a new Orca process is created. The dynamic information is updated whenever the manager executes an operation or services a remote invocation request. The manager, however, is not informed immediately about local read operations that other processors perform on their copy of a replicated object. For replicated objects, each processor keeps track of the number of read operations it executed. This information is piggybacked on the next write operation. The dynamic read counts of the manager may thus lag behind.

```
struct manager {
    int dynamic_writes; /* dynamic count of write operations */
    int dynamic_reads;  /* dynamic count of read operations  */
    int static_writes;  /* sum of compile-time write estimates  */
    int static_reads;   /* sum of compile-time read estimates   */
};
```

Figure 1: The information maintained by the manager of an object.

The RTS combines the compile-time and runtime information to estimate the values of W_i and R_i. The compile-time information is mainly useful to make initial estimates, since no runtime information is available yet. The RTS uses an aging mechanism that decreases the relative contribution of the compile-time values during execution. Likewise, to react adequately to changes in access behavior, the RTS uses another aging mechanism to give less weight to older statistics. To reduce the runtime overhead, the RTS does not apply the algorithm every time the manager's information is updated, but only once every ten updates. Whenever the algorithm is applied, the RTS determines the new best representation and compares it with the current representation for the object. If they differ, the RTS can start replicating an object, stop replicating an object, or migrate an object. We do not describe the implementation details of these three mechanisms here, since they are based on fairly standard techniques. In general, we use totally-ordered group communication to inform

all processors of the new status of the object. Note that replicating and migrating an object involve transferring the entire object state over the network, which may be expensive. The RTS therefore only changes the representation of an object if that is expected to result in much less communication.

In Section 3, we will analyze the object placement strategies experimentally, by running a large set of Orca applications with all possible combinations of compile-time and runtime heuristics. In particular, we will show that the Orca system is able to make good placement decisions and has little overhead compared to user-defined placement decisions.

Most other DSM systems are implemented entirely by a library or RTS, so they can only predict memory usage patterns based on information about the past. For DSMs that use a write-invalidate protocol, the replication strategy is determined implicitly by the access pattern (replicas are deleted on a write and reinstalled on a read). In some cases (e.g., producer-consumer behavior) this strategy may be inefficient. Some recent work has been done on using optimizing compilers for specific languages in combination with a DSM system, with the goal of reducing communication overhead [13, 16, 33]. In DiSOM, the programmer has to specify which machines get a copy of each shared object.

2.3.3. Portability

Portability is an important goal of Orca. Orca applications generally are fully independent of the underlying platform, and can run unmodified on a range of architectures. In this section, we discuss how we have also made the Orca system portable. The main design problem is how to make the system both portable and efficient. In particular, it is hard to use machine-specific properties in a portable system. Our approach is based on a virtual machine, called *Panda*, that can be configured statically to match the underlying architecture. For example, Panda requires only unreliable point-to-point communication from the underlying system, but on systems that support reliable communication, Panda can be configured to make use of that. (This approach is somewhat similar to the x-kernel [39].) Also, unlike some DSM systems, Panda does not require virtual memory support. Panda provides a fixed interface to the RTS, making the RTS machine-independent.

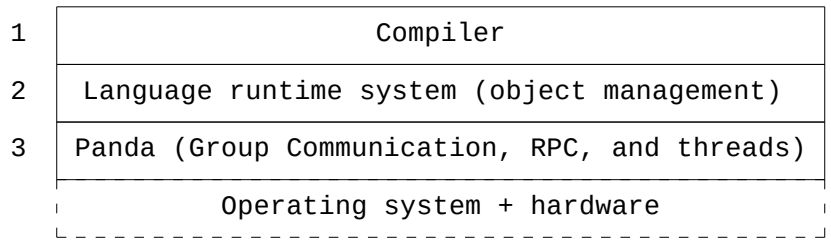

Figure 2: Layers of the portable Orca system.

Figure 2 depicts the layers of the Orca system. The first layer is the compiler, which translates Orca programs to ANSI C augmented with calls to the RTS. The compiler also generates information about which operations are read-only and how processes access shared objects, as described earlier.

The runtime system is the second layer. It is responsible for managing Orca processes and objects. It decides how to represent objects, it has mechanisms for remote object invocations and object migration, and it implements the write-update protocol for replicated objects.

Panda is the third layer in the Orca system. It provides the system facilities needed to implement the RTS. The Panda layer provides threads, Remote Procedure Calls (RPC), and totally-ordered group communication. Threads are used by the RTS to implement Orca processes; RPC is used to implement remote object invocations; and group communication is used by the write-update protocol.

The Panda thread interface is implemented either on top of existing kernel threads or by a user-level threads package. On most platforms Panda uses OpenThreads, which is a portable, efficient, and flexible user-level threads package [21]. For RPC and group communication, Panda has multiple implementations, depending on whether or not the underlying system supports reliable communication, messages of arbitrary size, and totally-ordered group communication.

Panda internally uses a separate interface to encapsulate the details of the underlying system (e.g., the number and types of the parameters of the communication primitives). The communication protocols are implemented on top of this internal interface, rather than on top of the underlying system. Porting Panda comes down to implementing this internal interface and configuring some modules that implement the communication protocols. The details of Panda and descriptions of various Panda ports are given in a separate paper [2]. The latter paper also shows that the performance overhead of the layered approach is small and that the Panda-based Orca system performs at least as good as the original, nonportable Orca system.

Several other DSMs have certain properties that make them somewhat less portable. Page-based DSMs (e.g., TreadMarks) usually need virtual memory support to detect page misses. Fine-grain DSMs (e.g., Shasta) often rewrite binaries to insert coherency checks, which makes them harder to port. Shrimp [8] uses special hardware support to implement an efficient DSM. Like Orca, CRL and DiSOM are implemented entirely in software, without requiring any special hardware or operating system support.

2.3.4. All-software, user-space architecture

The Orca system is implemented in software that runs entirely or mostly in user space. We look at the RTS and Panda in turn below and discuss the consequences of the all-software, user-space architecture.

The Orca RTS manages objects and processes and is implemented entirely with user-space software. Consequently, all access checks for shared objects are done in software. To implement an operation invocation, the RTS has to examine the current state of the object and determine whether the operation must be executed using a local invocation, a remote invocation, or a broadcast. For operations that result in communication, the relative overhead of this check is low compared to the communication time. For local operations, however, the relative overhead can be substantial (especially since Orca is multi-threaded, so these checks have to use locks). The overhead of the access checks is quantified in Section 3, showing that for the large majority of applications the overhead is negligible.

The Panda layer can be implemented using either kernel-space or user-space protocols. If the operating system kernel provides the right communication primitives for Panda (e.g., RPC), Panda can be configured to use these. Alternatively, Panda can be implemented on top of the kernel's datagram service. With this configuration, the network device is managed by the kernel but most of the communication protocols run in user space. Yet another possibility is to also map the network device into user space and to run the entire protocol stack (including the network device driver) outside the operating system. Most high-performance

communication layers on fast networks use the latter approach, because it avoids the relatively high costs of system calls. We use this approach on several platforms.

The main advantage of an all-software, user-space DSM (besides portability) is flexibility, which allows optimizations that would be hard or impossible to implement with kernel-space software. We illustrate this flexibility by describing one important optimization, concerning the handling of incoming messages. Messages can be handled using interrupts, but these often have a high cost. Alternatively, poll statements can be used to avoid interrupts, but getting the polling frequency right is often difficult. The idea of the optimization is to use interrupts whenever the processor is active, and switch to polling when the processor becomes idle. If, for example, a machine is doing a remote object invocation and no other threads are runnable, it will poll the network until the reply message comes back, so we save one interrupt in this important case. This approach is effective for the majority of Orca applications [27]. It clearly works better than always taking an interrupt; on systems (e.g., Myrinet) where polls are relatively expensive, it also works better than a pure polling-based approach. Implementing this strategy is nontrivial, however, since it requires a careful integration of the thread scheduler and the communication protocols. With user-space protocols, these components are easy to change, so we were able to implement this strategy in Panda [27].

Many other DSM systems have recognized the advantages of user-space protocols. For example, Shasta, CRL, TreadMarks, Midway, and DiSOM are all implemented in user-space. These systems perform access checks either in hardware or software. TreadMarks uses hardware memory management support to check whether a given page is available locally, so it can be classified as a mostly-software DSM [23]. Fine-grain DSMs (e.g., Shasta) let the binary rewriter insert (software) access checks. CRL uses a similar approach as Orca and does all access checks in user-space software.

2.4. Ports of the Orca system

The Orca system has been ported to several operating systems (including Solaris, BSD/OS, Linux, Amoeba, and Parix), parallel machines (the CM-5, SP-2, Parsytec GCel, Parsytec PowerXplorer, Meiko CS-2), and networks (Ethernet, Fast Ethernet, Myrinet, ATM). Although Panda was initially designed to support a portable implementation of Orca, it also has been used successfully for several other programming systems (e.g., PVM, SR, and MPI-Linda) [10, 42].

Below, we describe the Fast Ethernet and Myrinet ports of the Orca system in more detail, since they are used in Section 4 for the performance comparison with TreadMarks and CRL. The Myrinet port will also be used for the quantitative analysis of the Orca system in Section 3. Both ports use OpenThreads [21] to implement threads.

The Fast Ethernet port of the Orca system uses the kernel UDP/IP protocol (including the IP multicast extensions). The Panda system is configured (as described in Section 2.3.4) to use unreliable point-to-point and multicast primitives. Panda uses the algorithms described in Section 2.3.1 to implement totally-ordered group communication on top of unreliable multicast. The implementation of the protocols differs from our earlier work [24] by adding congestion control. Small messages are broadcast by forwarding them to the sequencer. Large messages are sent by first requesting a range of sequence numbers for all fragments of the message; the fragments are broadcast by the originator of the message. When a machine detects it has missed a fragment, it requests its retransmission from the sequencer or from the originator. This protocol performs

congestion control by postponing the reward of requested sequence numbers until sufficient acknowledgements have arrived at the sequencer. The acknowledgements are obtained in batches by a rotating token.

Another port of the Orca system uses Myrinet instead of Fast Ethernet. This port runs the entire protocol stack (including the device driver) in user space, to avoid the relatively high costs of system calls. This implementation is based on the Illinois Fast Messages (Version 1.1) software [38], which provides efficient reliable message passing. We have extended FM with reliable broadcast, using a spanning-tree forwarding protocol that runs entirely on the network interface processors of Myrinet (the LANai) [3]. The protocol uses a credit-based flow control scheme, which is also implemented on the network interfaces. The host processors are not involved in forwarding broadcast messages, which saves expensive interactions (e.g., interrupts) between the network interface and the host. The protocol therefore achieves a low latency. We have also added a primitive to the LANai software to retrieve a sequence number from a centralized sequencer machine, which is used by Panda to make the group communication totally ordered in an efficient way. The Panda system is configured to use reliable point-to-point and multicast communication for implementing RPC and group communication.

3. QUANTITATIVE ANALYSIS OF THE ORCA SYSTEM

In this section we provide a quantitative analysis of the design choices discussed above. Based on a collection of Orca applications, we will study the performance of the techniques and some of their alternatives. We first describe the applications and then discuss the coherence protocol for replicated objects, the object placement strategies, and the cost of a modular user-space software architecture.

The platform used for the performance measurements is a homogeneous cluster of Pentium Pro processor boards, running the BSD/OS (Version 3.0) operating system from BSDI. Each node contains a 200 MHz Pentium Pro, 64 MByte of EDO-RAM, and a 2.5 Gbyte IDE disk. The Pentium Pro processors have an 8 KByte, 4-way set-associative L1 instruction cache and an 8 KByte 2-way set-associative data cache. The L2 cache is 256 KBytes large and is organized as a 4-way set-associative, unified instruction/data cache.

All boards are connected by two different networks: Myrinet and Fast Ethernet (100 Mbit/sec Ethernet). We use Myrinet for most experiments (including the quantitative analysis described in this section). We use both Myrinet and Fast Ethernet for the comparison between Orca and TreadMarks (see Section 4).

Myrinet is a 1.28 Gbit/sec network. It uses the SAN (System Area Network) technology, consisting of LANai-4.1 interfaces (with 1 MByte SRAM memory) connected by 8-port Myrinet switches. The Myrinet switches are connected using a 2-dimensional torus topology (i.e., a grid with ‘‘wrap-around’’) . The Fast Ethernet network uses SMC EtherPower 10/100 adaptors, KTI Networks KF1016TX hubs, and a single NuSwitch FE-600 switch. The Fast Ethernet network uses segments with 12 or 14 processors each, connected by the switch. The switch has a delay of approximately 12 microseconds for small packets. The entire system is packaged in a single cabinet and was built by Parsytec (Germany).

We have measured the latencies of Orca’s basic communication primitives on the above hardware, using the Fast Ethernet and Myrinet ports of the Orca system. The two primitives are an object invocation on a remote (nonreplicated) object and a write operation on a replicated object. The results of these benchmarks, called ROI (Remote Object Invocation) and GOI (Group Object Invocation), are shown in Table 2, using empty messages (i.e., operations without parameters). For GOI, we use two versions: the object is either replicated on 8 or 32 machines. The latency is measured by having two processors in turn perform an increment

operation on the replicated object. The table gives the average latency measured over all possible pairs of processors. On Fast Ethernet, the fairly large difference in latency between these two versions is caused by the scheme we use for reliability and congestion control. On Myrinet, the difference is smaller, due to the efficient multicast protocol, which does message forwarding and flow control on the network interface processors.

Benchmark	Fast Ethernet	Myrinet
Remote object invocation	242	40.6
Group object invocation (8 nodes)	244	64.7
Group object invocation (32 nodes)	385	84.7

Table 2: Latencies (in microseconds) for Orca operations on the Pentium Pro cluster.

3.1. Applications

We have selected ten applications for the performance study. Together, they are representative for the kinds of applications for which Orca is typically used. They include numerical as well as symbolic applications. Some applications mainly use replicated objects, some use nonreplicated objects, and some use both. Below, we briefly describe the applications and the problem sizes we use.

- *ASP* (all-pairs shortest paths problem) finds the shortest path between any pair of nodes in a given 2,000-node graph.
- *Barnes* is the Barnes-Hut application from the SPLASH benchmark suite [46]. We have rewritten this program in Orca and use it to simulate the behavior of a system with 16,384 bodies.
- *IDA** is a combinatorial search algorithm that performs repeated depth-first searches. We use it to solve an instance of the 15-puzzle.
- *Linsolver* is an iterative linear equation solver. It is given a set of 800 equations as input.
- *RA* (Retrograde Analysis) computes a 13-piece end-game database for the Awari board game. It starts at the terminal positions with 13 or fewer pieces on the board and reasons backwards to compute the game-theoretical value of all positions in this database.
- *Skyline* solves a matrix equation for skyline matrices. It uses an input set with 3,500 equations.
- *SOR* (Successive overrelaxation) is a well-known iterative method for solving discretized Laplace equations on a grid. We use a 964×4,000 grid as input.
- *TSP* solves a 19-city traveling salesman problem using a branch-and-bound algorithm.
- *Water* is based on the ‘‘n-squared’’ Water application from the SPLASH benchmark suite. The program has been rewritten in Orca and simulates the behavior of a system with 1,728 water molecules, during 5 time steps (iterations).
- *Povray* is a public-domain ray tracer written in C. It was parallelized by adding calls to the Orca runtime system. This application thus is not written in the Orca language, but it does use the Orca RTS.

Figure 3 gives the speedups for the ten Orca applications on the Pentium Pro cluster, using the Myrinet port of the system. The speedups are computed relative to the parallel program on one CPU. For most applications, the performance of the Orca program on one processor is close to that of a sequential C program. (For

the three applications described in Section 4.4, one is even faster in Orca than in C, whereas the two others are 2% and 17% slower.)

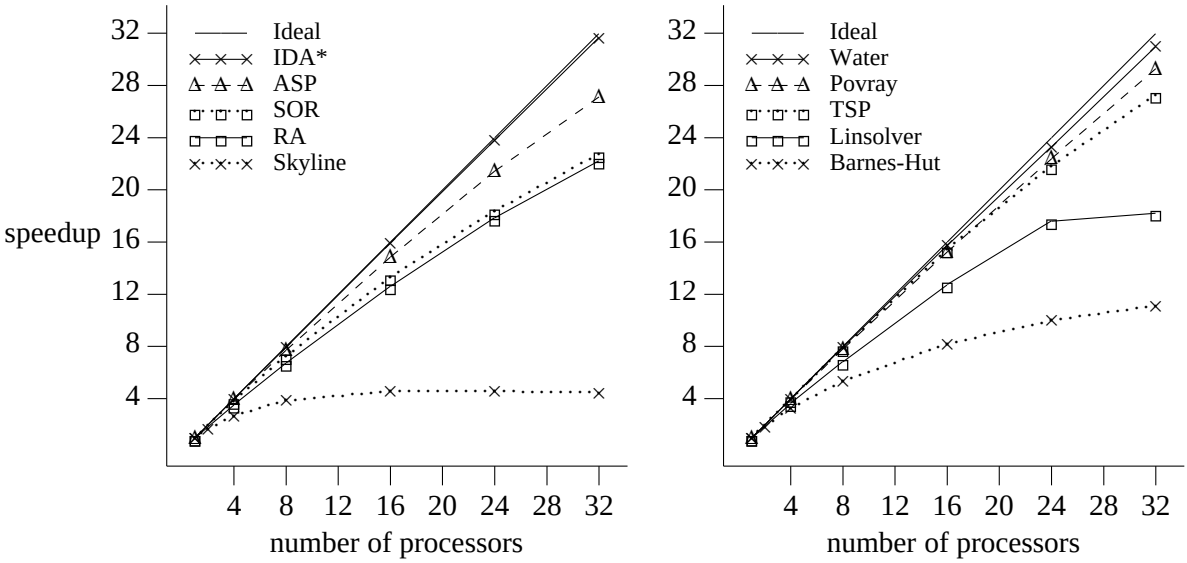

Figure 3: Speedups for ten Orca applications on the Pentium Pro/Myrinet cluster.

3.2. Coherence protocol for replicated objects

The effectiveness of Orca's write-update protocol depends on the behavior of the application. A write-update protocol is ineffective if the read/write ratio is low. In particular, the protocol performs poorly if one processor does many consecutive write operations on the same object, without intervening read operations from other processors. Each write operation will then be broadcast, but no processor will read the new data. An invalidation protocol, on the other hand, would only cause communication (an invalidation request) for the first write operation. We expect such behavior to occur infrequently for Orca applications, because the Orca system tries to replicate only objects with a high read/write ratio, and because consecutive writes to the same object are often expressed using a single logical operation (see Section 2.3.1). To analyze the behavior of applications, we have used the tracing facility of the Orca RTS. (For performance debugging purposes, the RTS can generate trace files that contain one record for every event that occurred during program execution, such as the start and end of an operation, the receipt of a message, and so on.)

The results confirm that the read/write ratio is high for replicated objects and that multiple consecutive writes without intervening reads occur infrequently for Orca programs. The lowest read/write ratio and the largest number of consecutive writes were observed for objects in two applications (the linear equation solver and Barnes-Hut) that implement an ''all-to-all exchange,'' in which every processor sends data to all other processors. Each process invokes two operations that implement a barrier-like behavior (i.e., they synchronize the callers) and also exchange data. The last process to arrive at the barrier often also is the first process to extract its data, so it performs two consecutive write operations. Also, the number of read and write operations is about the same. Still, a write-update protocol is far more efficient than write-invalidate, since the object is large (it contains data from all processors). A write-invalidate protocol would send the entire object over the

network many times.

Except for these cases, all replicated objects that are accessed frequently have a read/write ratio that is either close to the number of processors or significantly higher. This indicates that each processor reads the object one or more times after it has been updated. Also, multiple consecutive writes to the same object occur very infrequently. The only object for which a write-invalidate protocol might have been more efficient is used in IDA*. This object contains the total number of solutions found for the 15-puzzle. The object is written 71 times, and 32 of these write operations were issued without intervening reads. The impact on program performance of this object is negligible, however, given the very low number of operations. We have also determined how often objects are written consecutively by *different* processors, without intervening reads. In this case, a write-invalidate protocol would effectively migrate the object, which may be more efficient than updating it. Except for programs using an all-to-all exchange, this behavior also occurred seldom. In conclusion, we believe that a write-update strategy is indeed the best choice for Orca applications.

A second important property of Orca's coherence protocol is the usage of totally-ordered group communication, which makes it easy to update all copies of an object in a coherent way. This approach has several potential performance problems. First, the group communication protocols have a certain overhead (e.g., communication with the sequencer) for ordering the messages. Second, the protocols send group messages to all processors. We study the first problem below. The second problem is studied in Section 4.5, using an application (Barnes-Hut) that suffers from this problem (although it still obtains better speedups in Orca than with TreadMarks and CRL).

We study the overhead of totally-ordered group communication using the Myrinet port of the Orca system. With this port, the sender of a broadcast message first gets a sequence number from a centralized sequencer and then broadcasts the message tagged with the sequence number. The sequencer runs on one of the Myrinet network interface processors rather than on a host [3] (see also Section 2.4). For all ten applications, we have measured the average time needed to retrieve the next sequence number, which is implemented using a roundtrip message (*GetSeqno*) to the sequencer. This information can be used to compute how much time the application spends in retrieving sequence numbers, which is an estimate for the overhead of total ordering.

Application	Average GetSeqno (microsecs)	Total number of broadcasts	Total GetSeqno time (secs)	Execution time (secs)	Overhead per CPU (%)
ASP	30	2108	0.06	16.27	0.01
Barnes	72	812	0.06	5.83	0.03
IDA*	92	3294	0.30	31.96	0.03
Linsolver	135	182,354	24.62	18.46	4.17
Povray	0	0	0	15.36	0
RA	149	1459	0.22	21.76	0.03
Skyline	28	28316	0.79	13.85	0.18
SOR	44	1283	0.06	33.46	0.01
TSP	103	618	0.06	3.99	0.05
Water	0	0	0	14.46	0

Table 3: Overhead of getting sequence numbers for the totally-ordered group communication protocol.

Table 3 shows the results for 32 processors. The table gives the average latency for the *GetSeqno* primitive, the number of calls to this primitive issued by all processors together (i.e., the number of broadcasts in the program), the total time of these calls, the execution time of the program on 32 processors, and the percentage of overhead on execution time per processor. The overhead is calculated by dividing the total *GetSeqno* times (i.e., the fourth column) by the number of processors (32) and by the execution time of the program (i.e., the fifth column). We thus assume that all processors send the same number of broadcast messages, which is indeed the case for most applications. The minimum latency for the *GetSeqno* primitive is 28 to 30 microseconds, depending on the number of Myrinet switches between the sender and the sequencer. This latency is obtained for ASP and Skyline, in which only one processor broadcasts a message at the same time. Other applications send multiple broadcasts at the same time, for example when all processors access a replicated barrier object. In this case, the latencies increase (see the second column of Table 3). The impact on application performance of the *GetSeqno* calls is small, however. The worst case is Linsolver, which does many simultaneous broadcasts to implement an all-to-all exchange, resulting in about 4% overhead. For all other applications, the overhead is negligible.

This experiment shows that, for bursty broadcast traffic, the requests for sequence numbers may be slowed down, up to a factor of five. This slowdown need not be attributed only to the centralized sequencer, but may also be caused by contention on the network. The impact on application performance, however, is small, since the Orca system avoids replicating objects that are written very frequently and because we use an efficient sequencer implementation. If the sequencer is run on a host processor instead of on a network interface, the *GetSeqno* latencies increase (mainly due to interrupt processing overhead), resulting in 8% overhead for Linsolver and negligible overhead for the other applications.

3.3. Object placement strategies

We now study whether the Orca system makes the right decisions for object placement and how important the compile-time information and runtime statistics are. For this purpose, we have added a special “strategy” call to the RTS, with which the programmer can tell the RTS whether to replicate an object and where to store non-replicated objects. For each application, we manually inserted these strategy calls, using the best strategy for each object. We compare four different versions of the RTS:

Manual	Uses the strategy calls and ignores all other information.
Static	Only uses the compiler-generated information.
Dynamic	Only uses the runtime statistics.
Combined	Uses both the compile-time and runtime information, as described in Section 2.3.2.

The *Combined* version is normally used by Orca users. All but the first version ignore the strategy calls.

Table 4 gives the elapsed time of all applications on 32 processors, using the four different runtime systems. One important conclusion from this table is that the Orca system is able to make good placement decisions for the applications. For most applications, the *Combined* version of the RTS (which is the default) obtains almost the same performance as the *Manual* version, in which the programmer has to make all placement decisions. (In some cases the *Combined* version even is slightly faster than the *Manual* version, but this is probably caused by caching effects.)

Application	Manual (secs)	Combined (secs)	Static (secs)	Dynamic (secs)
ASP	16.32	16.27	16.26	16.25
Barnes	5.84	5.83	5.86	5.89
IDA*	31.99	31.96	35.56	31.95
Linsolver	18.62	18.46	18.81	18.54
Povray	15.33	15.36	-	15.29
RA	20.81	21.76	-	21.90
Skyline	12.73	13.85	12.72	12.82
SOR	34.07	33.46	33.66	34.53
TSP	4.00	3.99	4.31	3.99
Water	14.39	14.46	511.88	14.43

Table 4: Execution times for ten Orca applications using different placement heuristics.

For Skyline, the speedup for the *Manual* version is 8.7% higher than for the *Combined* version. In the latter version, the initial placement decision for certain objects is wrong; the decision is changed later on (based on runtime statistics), but the overhead of migrating the objects has a relatively high impact on speedup, given the short elapsed time of the program. For RA, the speedup obtained for the *Manual* version is 4.6% higher than that of the *Combined* version. For all other applications, the difference is less than 1%.

The table also shows that the runtime statistics are more important than the compiler heuristics. Only for SOR does the version without compiler support (i.e., the *Dynamic* version) obtain a slightly lower performance than the default version (the speedups are 22.2 and 22.9, respectively). If, on the other hand, the runtime statistics are omitted, performance sometimes degrades substantially. For Water, the *Static* version makes the wrong placement decision, resulting in very poor performance; for IDA*, the performance also drops significantly. For RA, the *Static* version runs out of memory, due to wrong placement decisions. For Povray, there is no static version, since this program is written in C.

It is also interesting to see how often the RTS changes the representation of an object, since these changes often require the current state of the object to be transferred over the network. For each of the ten applications, we have studied how often its objects migrate or change between being replicated and nonreplicated; we call such changes *state transitions*. We have determined the number of state transitions for each object in each application, using the *Combined* version of the RTS. Most objects are created by the first process (before other processes are forked), so they are stored initially on the first processor. The large majority of these objects subsequently go through one transition, after other processes have been created. Since this transition occurs early during the program, the object usually contains little data, so the state transition is inexpensive. Some objects suffer from a start-up effect and are, for example, migrated first and then replicated, resulting in one unnecessary state transition. We have identified only a few objects that go through several transitions. IDA* uses one job queue object per processor. Each object should be stored on its own processor, but due to the random (job stealing) communication pattern, the RTS migrates each object a few times (four times, in the worst case) before it ends up on the right processor. TSP uses a centralized jobqueue that should be stored on CPU 0 (which generates the jobs). Due to imprecise compiler analysis, however, the object goes through two unnecessary state transitions (from nonreplicated to replicated and back). In all these cases, the number of state transitions is low. Only for the Skyline program do the state transitions have an overhead that

is not negligible (as described above).

Finally, it is important to know what the runtime overhead of the dynamic statistics is. To measure this overhead, we have made another version of the RTS that does not maintain these statistics. In worst case (for Barnes-Hut) the overhead was 1%, but for all other applications the overhead was below 0.3%.

3.4. All-software, user-space architecture

The Orca system is implemented entirely in software. The disadvantage of an all-software DSM is that all access checks have to be done in software. For Orca, the RTS determines for every object invocation how it is to be executed. The RTS first checks if the object is replicated. If not, it checks if the object is stored locally or remotely. The access check is protected by a lock, since Orca is multi-threaded. We have measured that the runtime overhead of a single check is 0.4 microseconds.

In addition, we have determined how many operation invocations per second each of the applications executes on shared objects. From this information we can compute the overhead of the software access checks on application performance. The overhead is significant only for RA, which spends 13.8% of its time in access checks. For all other applications, the overhead is far less than 1%, because they execute much fewer operation invocations per second. An important reason for this low access-check overhead is that a single logical operation on a shared object can do many memory accesses to the object's data, so the costs of an access check are amortized over these memory accesses.

3.5. Summary

Quantitative analysis supports the claim that a write-update protocol implemented using totally-ordered broadcasting is effective for Orca. The decision of which objects to replicate and where to store nonreplicated objects can be made by the Orca runtime system, without involvement from the programmer. The statistics maintained by the RTS are essential for this purpose (and even more important than the compiler-generated information). The runtime statistics have a negligible overhead. Finally, except for one application, the overhead of implementing Orca in an all-software, user-space system is very small.

4. PERFORMANCE COMPARISON OF ORCA, TREADMARKS, and CRL

In this section we compare the performance of the Orca system with two other distributed shared memory systems: TreadMarks and CRL. TreadMarks is a modern page-based DSM that uses a multiple-writer protocol and lazy release consistency to reduce communication overhead. CRL, like Orca, is an object-based DSM system. CRL's implementation is very different from Orca's, since it uses a directory-based invalidation protocol. Also, CRL is a library, whereas Orca is a language.

Below, we first discuss the methodology used for the performance comparison. Next, we outline how TreadMarks and CRL were implemented on the Pentium Pro cluster and we describe the performance of their low-level operations. We emphasize that these performance figures should not be compared against each other, because the low-level primitives in the three DSMs are very different. Finally, we use several parallel applications to compare Orca with both TreadMarks and CRL.

4.1. Methodology

An important goal in our comparison is to use identical hardware for Orca and the two other DSMs, which is the Pentium Pro cluster described in Section 3. Ideally, we would like to run the TreadMarks and CRL systems ‘‘out-of-the-box’’ on this platform. TreadMarks is available on a variety of platforms, but not on Myrinet networks. We were able to run the Unix (UDP/IP-based) implementation of TreadMarks on our cluster, using Fast Ethernet for interprocessor communication. This allowed us to compare Orca with an (almost) ‘‘out-of-the-box’’ version of TreadMarks. Unfortunately, TreadMarks often performs poorly on this platform, because its high communication volume causes too much contention on the (hubbed) Fast Ethernet, resulting in many lost messages. To allow a fair comparison between Orca and TreadMarks, we therefore also ported TreadMarks to Myrinet. This port required several changes to TreadMarks (described later).

CRL also has been implemented on a variety of platforms, including the CM-5, Alewife, and Unix clusters. The implementation on Unix is based on PVM and TCP/IP and is not well tuned. To allow a fair comparison between CRL and Orca, we ported the CM-5 implementation of CRL to Myrinet. We thus use our Myrinet ports of TreadMarks and CRL for the performance comparison. In addition, we compare Orca with the original TreadMarks system, using Fast Ethernet. Our discussion of the results will focus on the Myrinet ports.

We use three applications for the performance comparison: Water, Barnes-Hut, and the Traveling Salesman Problem (TSP). The first two applications are from the SPLASH suite. Barnes-Hut is a ‘‘challenging’’ application that does much communication. An important issue is which programs to use for each DSM. We decided to use mainly the programs developed by the designers of the three systems, and only made changes to the original programs when this was required to allow a fair comparison (as described later). The advantage of this decision is that each program is written in a style that best fits the specific DSM model. A disadvantage is that the three programs for each application differ in important ways, because the programming styles differ. The CRL designers stayed closest to the shared-memory style. For the two SPLASH applications, for example, they made few changes to the original shared-memory codes, whereas the Orca and TreadMarks programs made several optimizations to reduce communication overhead. We thus do an ‘‘end-to-end’’ comparison: our conclusions reflect differences in both the programming systems and their programming styles.

We compare the different systems by presenting the speedups of the application programs relative to a sequential C program, so we use the same base program for all three systems. The sequential C programs we used are the CRL programs with all calls to the CRL library removed (as in [23]). The hardware configuration used for the measurements is described in Section 3.

4.2. Implementation of TreadMarks

We ported TreadMarks (Version 1.0.0) to BSD/OS, using UDP/IP (over Fast Ethernet) for communication. We had to make only a few minor modifications to TreadMarks (mainly to get around a problem with socket buffer space in BSD/OS). We had to slightly extend BSD/OS’s signal handler facility (because the segmentation-violation signal handler did not provide the address at which a violation occurs).

In addition, we ported the TreadMarks system to Myrinet, using a modified version of Illinois Fast Messages 1.1 for communication. (The modifications to FM allow the generation of a Unix I/O signal on message arrival and give the user control over message-buffer deallocation.) As with the Orca (and CRL) system on

Myrinet, the network device is mapped in user space. The Myrinet port required several important modifications to TreadMarks. We reimplemented the send/receive primitives used by TreadMarks, using the FM interface, which is based on an upcall model. Since FM on Myrinet provides reliable communication, the timeout mechanism that TreadMarks uses for UDP/IP was disabled. Incoming messages are serviced through a Unix signal handler, using FM's ''extract'' primitive (which extracts messages from the network and performs the upcalls). However, for flow-control reasons, FM's ''send'' primitive also extracts incoming messages. So, if a signal handler is invoked while the send primitive is extracting messages, a race condition may occur. We solve this problem using a global flag, which informs the signal handler to delay handling incoming requests. Finally, if a processor is idle, it receives incoming messages through polling, to avoid the cost of an interrupt.

We have run some of the low-level TreadMarks benchmarks described in [35]. The results are shown in Table 5, for Fast Ethernet and Myrinet. For Fast Ethernet, the latencies on the Pentium Pro cluster are already lower than those reported for the SP-2 in [35]; for Myrinet, the latencies are even lower.

Benchmark	Latency on Fast Ethernet	Latency on Myrinet
8-processor barrier	780	130
32-processor barrier	3380	570
2-processor lock	300	80
3-processor lock	430	100
Empty diff page fault	560	330
Full page diff page fault	1460	940

Table 5: TreadMarks latencies (in microseconds) for the Pentium Pro cluster.

4.3. Implementation of CRL

We ported the CM-5 active messages implementation of CRL (Version 1.0) to Myrinet. We used our extended version of Illinois Fast Messages (see Section 2.4) as communication substrate, so CRL and Orca use exactly the same low-level communication software. We have replaced all CM-5 calls with FM calls as described below, but we have made no changes to CRL's coherence protocol code.

CRL's control messages are transmitted using *FM_send_4*, which carries four words of CRL control information. Since FM uses fixed size packets (of 256 bytes on our system), this primitive transmits 256 bytes. Messages carrying region data are transmitted using *FM_send_buf*, which packetizes the data buffer into 256-byte network packets. CRL's collective communication messages are implemented using *FM_send_4* and *FM_broadcast*. As in the CM-5 implementation, we use interrupts (by default) to deliver messages. Interrupts are handled using Unix signal handlers, which are relatively expensive. Handling a message via a signal handler instead of via polling adds approximately 24 microseconds of overhead. Not all messages generate an interrupt, however. CRL disables interrupts when waiting for a (reply) message and then polls the network until the message has arrived. While polling, other protocol messages that arrive before the reply are also processed.

We have increased the size of CRL's unmapped region cache. Its default value, 1K entries, turned out to be too small for the Barnes-Hut program with the given input size, resulting in many capacity misses. The cache size we use is 16K entries, which improved the performance on 32 nodes by more than 25%. (Increasing

the cache size further did not improve performance.)

We have run some of the benchmarks used in [23] on our platform, using CRL on FM (see Table 6). As expected, the latencies are substantially better than those for the CM-5 reported in [23], mainly because the Pentium Pros are much faster than the CM-5's SPARC processors.

Benchmark	Latency on Myrinet
Start read (hit)	0.35
End read	0.30
Start read (miss, no invalidations)	32.68
Start write (miss, 1 invalidation)	32.81
Start write (miss, 8 invalidation)	123.32
Start write (miss, 32 invalidations)	345.89

Table 6: CRL latencies (in microseconds) for 16-byte regions on the Pentium Pro cluster.

4.4. Performance comparison between Orca and TreadMarks

The original TreadMarks programs are described in [35]. Barnes-Hut and Water are based on the SPLASH-1 codes, but the TreadMarks designers optimized the codes to get better performance on their system [35]. We have made no significant changes to the Water program. We changed the Barnes-Hut program to use the data structures of the SPLASH-2 version of this application, since the Orca and CRL programs are also based on the SPLASH-2 version. We had to rewrite the TreadMarks TSP program to let it use the same algorithm and data structures as the Orca and CRL programs. The original TreadMarks TSP program used a prioritized job queue. The new program uses a simple, centralized job queue. A master process generates all jobs consisting of the first 3 hops. For each job, a lower bound is computed for the length of the rest of the path. The jobs are searched recursively, with cutoffs when the length of the path plus the lower bound of the rest of the path is larger than the current minimum. Our changes to the original TreadMarks codes for Barnes-Hut and TSP had only a small impact on the performance behavior of the applications.

Figure 4 gives the speedup (relative to the sequential C program) for the three Orca, TreadMarks, and CRL applications on Myrinet, using up to 32 Pentium Pro processors. Table 7 contains data on the communication behavior of the programs on Myrinet, using 8 and 32 processors; the table gives the total number of messages and the total amount of user data sent by all processors together. For Orca, each remote object invocation counts as two messages (a request and a reply); a write operation on a replicated object counts as one broadcast message. Except for the execution times, the data were collected with separate runs of the programs, in which instrumentation code was enabled. Figure 5 gives the speedups of the applications on Fast Ethernet. Below, we discuss the performance of the three applications in turn. Except where indicated otherwise, this discussion concerns the results for Myrinet.

Water. On a single processor, the Orca Water program is somewhat slower than the TreadMarks program (442 versus 394 seconds). Both programs are faster than the sequential C program (which takes 557 seconds). We believe that most of this difference is due to caching effects. The Orca program scales well and obtains superlinear speedup relative to the (slow) sequential C program. The TreadMarks program scales

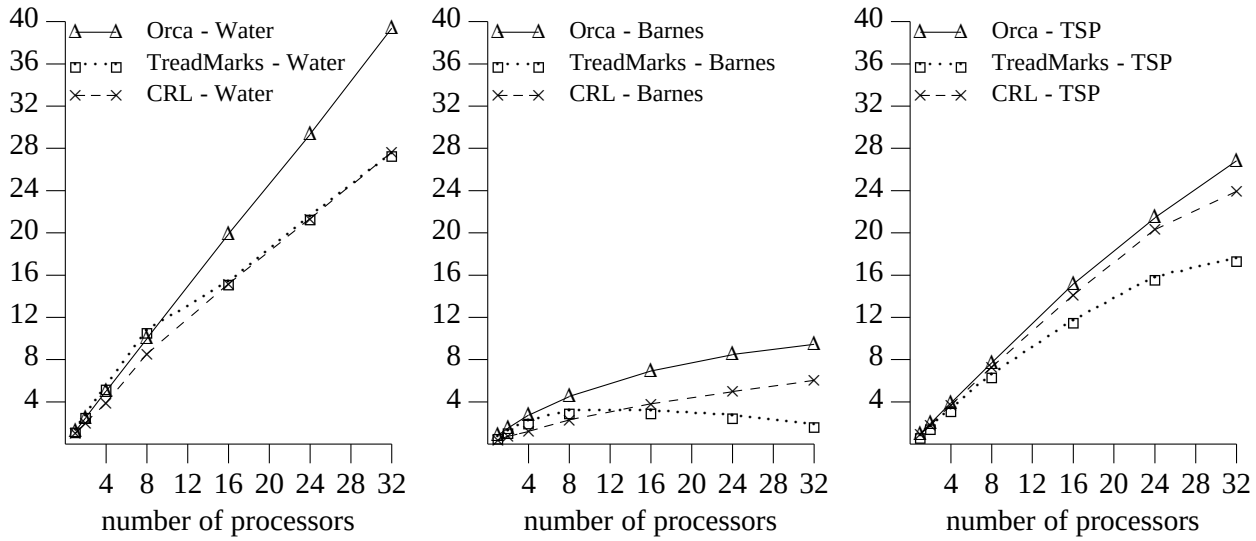

Figure 4: Speedups for the Orca, TreadMarks, and CRL applications on Myrinet.

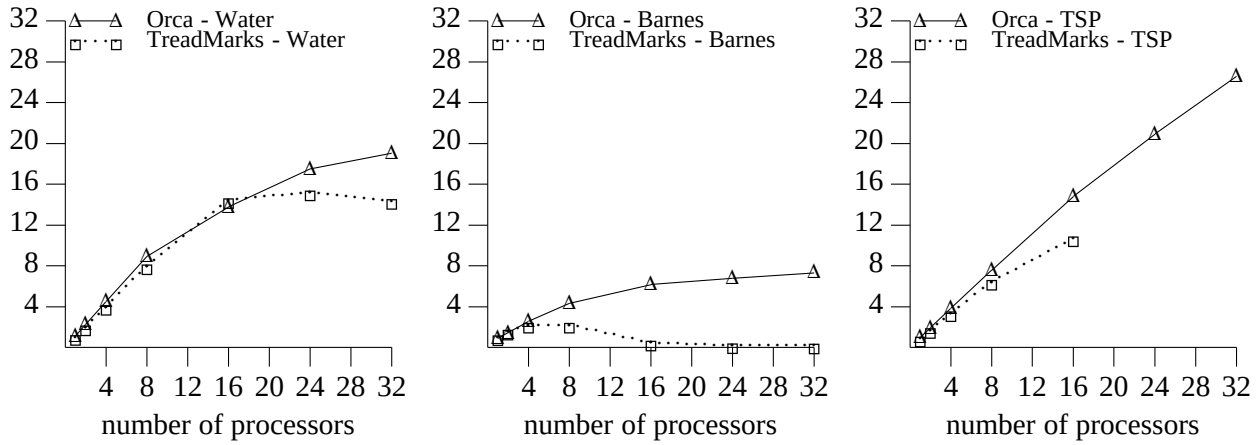

Figure 5: Speedups for the Orca and TreadMarks applications on Fast Ethernet.

reasonably well. It is also interesting to compare each program relative to itself (rather than to the sequential C program), to determine its scalability. The Orca program runs 30.6 times faster on 32 processors than on 1 processor. The TreadMarks program runs 19.5 times faster on 32 processors than on 1 processor. On 32 processors, the TreadMarks program takes 40% longer than the Orca program.

In the Water algorithm, each processor needs data from half of the other processors. The total amount of data transferred increases linearly with the number of processors. The Orca program indeed reflects this behavior. If the number of processors is increased from 8 to 32, the program sends 11 times more messages and 4 times more data (see Table 7). For the ThreadMarks program, the number of messages increases by a factor 6.7 and the amount of data by a factor of 9.7, so the amount of data increases much faster than in the Orca program. To explain the differences in communication behavior, we discuss the Orca and TreadMarks programs for Water in more detail below.

Program	#CPUs	System	time (sec)	#unicast messages	unicast data (MB)	#broadcast messages	broadcast data (MB)
Water	8	Orca	56.45	1,815	8.08	0	0.00
	32	Orca	14.46	20,608	32.38	0	0.00
	8	TreadMarks	51.45	8,886	20.71		
	32	TreadMarks	20.23	59,512	201.31		
	8	CRL	65.18	461,080	103.76	88	0.00
	32	CRL	20.17	1,831,460	409.45	88	0.00
Barnes	8	Orca	12.14	700	0.01	136	15.51
	32	Orca	5.83	3,100	0.02	544	15.52
	8	TreadMarks	17.18	493,009	82.84		
	32	TreadMarks	28.70	5,951,213	463.67		
	8	CRL	23.87	1,154,546	33.90	92	0.00
	32	CRL	9.13	2,585,341	71.97	92	0.00
TSP	8	Orca	12.83	8,850	0.18	83	0.00
	32	Orca	3.69	9,347	0.26	767	0.01
	8	TreadMarks	15.00	8,973	1.74		
	32	TreadMarks	5.63	56,948	11.74		
	8	CRL	13.56	11,274	1.33	8	0.00
	32	CRL	4.14	16,484	5.82	8	0.00

Table 7: Performance data for the Orca, TreadMarks, and CRL applications on Myrinet.

In the Orca version of Water, the molecule data are partitioned over multiple shared objects, one per processor. The RTS does not replicate these objects, because they have a low read/write ratio. At the beginning of the force calculation phase, each processor reads the data it needs from the other processors' objects, using a single operation invocation per processor. During the force computation, all changes are accumulated locally. After the computation is finished, the forces in all shared objects are updated using remote object invocations. These optimizations are easy to express in Orca, by defining the appropriate abstract data type operations for the objects containing molecule data.

The TreadMarks version for Water uses similar optimizations. The molecule data has been split into shared and nonshared parts, and only the shared part is allocated in shared memory [35]. So, only the shared parts are sent over the network. Also, the force calculations have been optimized to accumulate force changes in private memory and update the shared data structures after all local computations have completed. Still, on 32 CPUs, the TreadMarks program sends 2.9 times more messages and 6.2 times more data than the Orca program. As explained in [35], there are several reasons for this behavior. The force value of a molecule is modified by about half of the processors and TreadMarks's multiple writers protocol needs to get diffs from all these processors to obtain the new value of the data. In addition, the program suffers somewhat from false sharing.

On Fast Ethernet (see Figure 5), the high communication volume generated by TreadMarks also causes a high collision rate on the network. On more than 8 processors this even results in lost messages. On 32 nodes, TreadMarks retransmits about 1600 messages in total (after a timeout of 100 milliseconds each), which contributes to the performance degradation.

Barnes-Hut. The Barnes-Hut program executes 11 iterations, but (as is usual for this application) the first three iterations are excluded from the timing measurements, to avoid cold start effects. On 1 CPU, the

Orca program is somewhat faster than the TreadMarks program (64.7 versus 70.9 seconds) but slower than the sequential C program (which takes 55 seconds). The TreadMarks Barnes-Hut program obtains reasonable speedups up to 8 processors, but its performance drops on more processors. If the number of processors is increased from 8 to 32, the number of messages increases by a factor of 12 and the amount of data by a factor 5.6. On 32 nodes, the program sends 464 MBytes of data in 28.7 seconds. In [35], the overhead of the SPLASH-1 version of Barnes-Hut on 8 nodes is analyzed and is attributed mainly to false sharing and to the multiple-writer protocol. Also, the program suffers from a large number of page faults, since the octree data is split over many pages. In [1], an adaptive protocol for TreadMarks is described that switches (on a per-page basis) between a single-writer and a multiple-writer protocol. For Barnes-Hut, the adaptive protocol achieves slightly better performance than the multiple-writer protocol.

The Orca version of Barnes-Hut stores all data of the octree in a single shared object that is replicated on all machines. During every iteration, each processor reads the part of the data it needs and stores it in a local data structure. At the end of the iteration, each processor in turn updates the shared object. This implementation results in much broadcast traffic (15 MBytes). If the number of processors is increased, the volume of the broadcast data remains constant, but the number of broadcast messages increases. The speedup of the program increases up to 32 processors. On 32 processors, the TreadMarks program takes 4.9 times longer than the Orca program, due to the huge difference in the number of messages being sent.

On Fast Ethernet (see Figure 5) the communication behavior of TreadMarks causes a high collision rate on the network, resulting in many messages being lost. On 32 processors, over 36,000 messages (about 0.5% of all messages) are retransmitted in total, which explains the dramatic performance degradation of the program. The performance of the Orca program increases up to 32 processors.

TSP. For TSP, Orca is somewhat faster than TreadMarks on 1 CPU (101 versus 114 seconds) and almost as fast as the sequential C program (which takes 99 seconds). On 32 CPUs, the TreadMarks program takes 53% longer than the Orca program (5.63 versus 3.69 seconds). On multiple CPUs, the original TSP programs behave very nondeterministically, with widely varying execution times for the same input. This behavior is caused by search overhead: depending on how fast the optimal solution is found, the program may search different numbers of routes. To avoid this behavior, we applied a (well-known) trick: the global bound is initialized with the optimal value computed during an earlier run. In this way, the number of nodes searched is constant. The most important source of communication overhead is the work distribution. On 32 nodes, the TreadMarks program generates almost 6 times as many messages and 43 times as much data as the Orca program, causing the Orca program to scale better. On Fast Ethernet (Figure 5), the TreadMarks program fails to execute on more than 16 CPUs, because the consistency data that TreadMarks generates becomes larger than the maximum UDP message size.

4.5. Performance comparison between Orca and CRL

For the comparison between Orca and CRL, we use the same three applications as described above. The CRL versions of Barnes and Water are described in [23]. The designers of CRL ported these SPLASH applications to CRL, using regions to encapsulate shared data structures, but making only minimal changes to the SPLASH codes. The TSP program is part of the CRL distribution; we modified it to let it use the same (and better) pruning heuristic as in the TreadMarks and Orca program. The performance results (on Myrinet) are shown in

Figure 4 and Table 7.

Water. On 1 CPU, the CRL Water program takes 21% longer than the Orca program, but 4% less than the sequential C program; as stated before, these differences are probably due to caching effects. The CRL program scales well, although it achieves lower speedups than Orca.

The performance differences can be explained as follows. The CRL Water program uses one region for every molecule. The unit of sharing in the CRL program thus is much smaller than in the Orca version, which uses one shared object (with many molecules) per processor. Also, unlike the Orca and TreadMarks versions, the CRL program does not split the shared and nonshared parts of the molecule data and neither does it accumulate the updates to the forces locally. As a result, the CRL program sends far more messages and far more data than the Orca program. The CRL program obtains almost the same speedup as the TreadMarks program, although it sends more messages and data; this may be caused by the less bursty communication behavior of the CRL program.

Barnes-Hut. On a single processor, the CRL Barnes-Hut program runs a factor 2.7 slower than the sequential C program. Such a large difference was also reported in [23] for the CM-5; it is due to the large number of software access checks that are executed. The Orca program does not have this high overhead, because each processor copies the octree data into a local (nonshared) data structure. On one processor, the Orca program is 2.3 times faster than the CRL program. On 32 processors, the Orca program is about 1.6 times faster than the CRL program. The CRL program thus scales better than the Orca program, but on 32 nodes the Orca program still obtains the shortest execution time.

The CRL Barnes-Hut program uses one region for every octree data structure (body, cell, or leaf), whereas the Orca program stores all data in one shared (replicated) object. The two programs therefore have very different communication behavior. With CRL's write-invalidate strategy, a body will be replicated (or ''cached'') on demand; the replicas are invalidated on every write. If a processor wants to read a body after it has been written, it fetches a new copy. Orca's write-update protocol, on the other hand, broadcasts all write operations to all machines, but allows all read operations to be performed locally (without communication). A write operation broadcasts data to all processors, including those processors that are *not* going to read the body.

In Barnes-Hut, every body is read by several (but not all) processors, so both approaches have their disadvantages. CRL will send many point-to-point messages, to fetch the octree data on demand. Orca will unnecessarily send the data to all processors and interrupts all processors. The performance data in Table 7 clearly reflect these differences. If the number of processors is increased from 8 to 32, CRL sends only 2.2 times as many messages and 2.1 times as much data. In Orca, the amount of broadcast data remains constant (15 MBytes), but the number of broadcast messages increases with a factor 4 (i.e., linear with the number of processors). On 32 CPUs, Orca sends 15 MBytes of broadcast data, whereas CRL sends 72 MBytes of point-to-point data. These two numbers are hard to compare. In Orca, each receiver has to process 15 MBytes of data, whereas in CRL every processor receives only a small part of the 72 MBytes (2.25 MBytes on average), giving CRL a clear advantage. On the other hand, CRL sends far more messages, because (as for Water) it does not accumulate force updates locally and it uses point-to-point messages instead of broadcasts. We have measured that the CRL program gets about 546,000 read misses and 195,000 write misses (each requiring several protocol messages), which explains most of the 2.6 million point-to-point messages being sent.

TSP. On 1 CPU, the Orca TSP program is somewhat faster than the CRL program (101 versus 106 seconds). The Orca program also obtains somewhat better speedups. Both programs scale well, due to the low communication overhead.

The CRL program lets every processor read the entire jobqueue at the start of the execution. Since every processor will only read (and execute) part of the jobs, the CRL program transfers unnecessary data. These data transfers use efficient bulk transfers, however, so their overhead is small. Moreover, a processor that needs a new job now only has to fetch a job number; it already has the job's data. In the Orca program, the job queue is stored on the first processor; other processors fetch a new job (including the job data) when they are finished with the current job. As can be seen in Table 7, CRL sends more data than Orca. On 32 nodes, the Orca program is somewhat faster than the CRL program (3.69 versus 4.14 seconds).

4.6. Summary

The comparison between Orca and TreadMarks shows that the TreadMarks programs send far more messages and more data than the Orca programs, and also obtain lower speedups. If the number of processors is increased from 8 to 32, the TreadMarks Water program generates almost a factor 10 more data, the Barnes-Hut program generates a factor 12 more messages, and the TSP program generates a factor of 6 to 7 more messages and data. Even on a fast network like Myrinet, this results in substantially lower speedups for TreadMarks than for Orca. On a Fast Ethernet network (with a single switch), the communication behavior generates congestion and lost messages, resulting in poor (and sometimes meaningless) speedups. On the other hand, TreadMarks requires less programming effort than Orca, since it provides a shared memory model with a flat address space. Especially, it is easier to run existing shared-memory codes in TreadMarks.

The Orca programs also run faster than the CRL programs (and send fewer messages and data). The performance difference is caused partly by differences in programming style. In the CRL programs, shared data are encapsulated in regions by using a straightforward transformation from the shared-memory codes, resulting in a smaller unit of sharing (and more messages) than in Orca. Of particular interest is the Barnes-Hut application, which illustrates the difference between Orca's write-update protocol and CRL's write-invalidate protocol. Orca broadcasts all updates of the octree data to all processors, whereas in CRL each processor gets only the data it requires, but using many point-to-point messages. Up to 32 CPUs, the Orca program is faster, but the CRL program seems to scale better. Also, on 32 processors, CRL obtains better performance than TreadMarks for Barnes-Hut and TSP and the same performance for Water.

5. RELATED WORK

We discuss three areas of related work: distributed shared memory, parallel languages, and sharing patterns.

Distributed Shared Memory

Many other distributed shared memory systems have been built [6, 7, 11, 17, 26, 28, 36, 37, 40, 43, 44, 45]. In the paper, we mainly discussed page-based (e.g., TreadMarks) and object-based (e.g., CRL) DSM. Another interesting class of DSM systems is *fine-grained* DSM, which tries to reduce false sharing by using a unit of sharing that is smaller than a physical page. Examples are Blizzard-S [45] and Shasta [43]. We discuss Shasta here. Shasta supports multiple granularities of sharing within a single application. The unit of sharing (and

coherence) is called a block. A program may use blocks of different sizes, which can be defined by the user. The Shasta compiler rewrites the execution binary and inserts access checks at memory loads and stores to determine if the processor has a valid copy of the block being accessed. Shasta uses many optimization techniques to reduce the overhead of these software checks. Shasta is implemented completely in user-space software, without requiring any special support from the operating system or hardware. In comparison with Orca, Shasta has the advantage of offering a flat shared address space, making it easy to run existing shared-memory applications. The Orca system, on the other hand, is easier to port than Shasta, since its implementation does not depend in any way on the processor architecture.

Another DSM system related to Orca is SAM [44]. An important goal in SAM is to give the user more control over communication. The programmer must indicate how shared data will be accessed, and this information is used to reduce the communication and synchronization overhead, using prefetching and caching techniques. Two kinds of shared data are defined: values with single-assignment semantics and accumulators (which reflect producer-consumer behavior). Orca provides a simpler, more uniform object model, and does not use information provided by the programmer. Instead, the Orca system lets the compiler and runtime system together optimize communication.

DiSOM [11], like Orca, is an object-based, all-software, user-space DSM system. Both Orca and DiSOM use a write-update protocol to implement write operations on replicated objects. The two systems differ in that Orca uses function shipping whereas DiSOM uses data shipping; on a write operation, DiSOM transfers (part of) the object state, while Orca applies the operation to all copies. Also, the programming models of Orca and DiSOM differ, in that DiSOM uses explicit synchronization primitives.

Parallel languages

Most DSM systems are implemented as libraries that can be linked with programs written in an existing, sequential language. In contrast, we use a language-based approach. The shared object model described in the paper is supported by the Orca programming language, which was designed specifically for parallel programming. It therefore is relevant to also compare Orca with other parallel languages, in particular with object-oriented languages. A large number of parallel languages exist that extend sequential object-oriented languages such as C++ [32].

An interesting example is Mentat [19], which, like Orca, is implemented with portable, user-space software, using a layered approach. Mentat's programming model is quite different from Orca's model. The basic idea in Mentat is to let programmers only express *what* should be executed in parallel. More precisely, programmers indicate which classes have methods that are coarse-grained enough to let them execute in parallel. Operation invocations on such classes are executed asynchronously, using a macro dataflow model. An operation can start as soon as all its inputs are available. The relative overhead of Mentat's RTS is substantial, especially on machines with fast communication [20]. In return, Mentat offers a high-level programming model and lets the RTS take over many responsibilities from the programmer, such as communication, synchronization, and load balancing.

Sharing patterns

The Orca system uses a combination of compile-time and runtime techniques to determine the placement of objects. Many techniques exist that are related to ours. Early work on runtime techniques for data placement includes Lucco's implementation of Linda, which dynamically monitored the usage of tuples [31].

CVM [25] is a software DSM (similar to TreadMarks) that has been used as a target for a parallelizing compiler. CVM uses a hybrid invalidate/update protocol. The compiler determines which pages have a communication behavior for which an update protocol is better than an invalidation protocol (which is the case, for example, for producer-consumer patterns). For such pages, CVM keeps track at runtime which processors try to access the page. After every barrier synchronization, CVM then sends the page (or a diff) to these processors, so they don't get a page fault when they next try to use the page. If the compiler analysis fails, the processors still get a page fault (and get the page ''on demand''), resulting in less efficient but correct programs. Note that this approach exploits the release consistency semantics, which informally means that updates only have to be visible at certain synchronization points, such as barriers.

Some languages also provide notations with which the programmer can help the system to make the right decisions about data placement. In High Performance Fortran, the programmer can directly specify the preferred distribution (partitioning) of shared arrays. Braid (which is a data-parallel extension to Mentat) allows the programmer to specify a communication pattern, which the RTS uses to determine the distribution of data-parallel objects [47]. The Orca system described in this paper does not partition objects. A data-parallel extension to Orca has been designed, however, that allows array-based objects to be partitioned, using directives from the programmer [22].

Greenberg et al. use a high-level notation for describing communication patterns [18]. Their notation specifies communication between processors via a permutation on the processor set (e.g., a transpose or a shift), so a pattern gives a global description of the communication behavior of many processors. In our system, on the other hand, a communication pattern describes the behavior of a single process. Patterns in our system can be generated automatically, however, whereas Greenberg et al. require the user to provide the pattern.

6. CONCLUSIONS

Orca is a language-based distributed shared memory system. It supports an object-based programming model that integrates synchronization and data accesses and provides sequential consistency. The implementation of Orca differs in several ways from most other (more recent) DSM systems. For replicated objects, Orca uses a write-update protocol with function shipping, implemented on top of totally-ordered group communication. To decrease communication overhead, the Orca system dynamically determines which objects to replicate, using a combination of compile-time and runtime information. The Orca system is implemented entirely in software, using several layers, which run mostly or entirely in user space.

The main contributions of the paper are a detailed analysis of the design choices made in the Orca system and a performance comparison with two other DSM systems, TreadMarks and CRL. The quantitative analysis of the Orca system has revealed several interesting insights. A write-update protocol with function shipping indeed is a good approach to implement object-based DSM, especially if it is used in combination with other techniques that avoid replicating objects with a low read/write ratio. Unlike in paged-based DSMs (or

hardware DSMs [12, 29]), multiple consecutive writes to the same shared data structure are typically expressed using a single (logical) operation, resulting in only one communication event. A write-invalidate protocol would have a much higher communication overhead for Orca, since it would frequently have to transfer the object state. A possible alternative to a write-update protocol would be a write-invalidate protocol that invalidates (and transfers) only part of the object state (e.g., by using fine-grain sharing [45] or page-diffing [14] within objects).

Orca's write-update protocol is implemented on top of totally-ordered group communication, because this simplifies the coherence protocols significantly. Performance analysis shows that the overhead of the totally-ordered protocol on application performance is small. On a 32-node Myrinet-based system, the overhead involved in communicating with the sequencer was 4% for one application and less than 0.2% for all other applications.

An important goal in the Orca system is to determine the best placement strategy for each object without involvement from the programmer. The Orca runtime system uses information about object usage provided by the compiler and also maintains dynamic statistics (counts). By combining this information, the runtime system determines which objects to replicate and where to store nonreplicated objects. An analysis of ten applications shows that the system is able to make near-optimal decisions in most cases. Most programs achieve a speedup within 1% of that obtained by a version in which the programmer makes all decisions. The analysis also shows that the runtime statistics are more important than the compiler information and that maintaining the statistics has negligible runtime overhead (less than 1%).

We have compared Orca with TreadMarks and CRL, using three parallel programs written mostly by the respective designers of the DSM systems. The Orca programs generally have a much lower communication overhead and better speedup than the TreadMarks and CRL programs, sometimes at the cost of a higher programming effort. In the Orca programs, all shared data structures have to be encapsulated in shared objects, and often the object types are optimized to reduce the communication overhead. The TreadMarks programs implement similar optimizations, but do not require the programmer to encapsulate data in objects. The CRL programs were written in a shared-memory style, with little or no specific optimizations.

An important insight from our work is that two decisions have had a profound impact on the design and performance of the Orca system. The first decision was to access shared data only through abstract data type operations. Although this property requires work from the programmer, it is the key to an efficient implementation. It often reduces the communication overhead, because an operation always results in only one communication event, even if it accesses large chunks of data. In particular, an operation that modifies many memory words of an object is treated like a single write-operation by the runtime system. Also, the property reduces the overhead of access checks (since this overhead is amortized over many memory accesses), allowing an all-software implementation. In addition to these performance advantages, using abstract data types for shared data also leads to well-structured programs that are easy to understand and maintain.

A second important decision was to let the Orca system replicate only those objects that have a high read/write ratio. Since replicated data are written relatively infrequently, it becomes feasible to use a write-update protocol for replicated objects. In addition, it allows the usage of totally-ordered broadcast, which greatly simplifies the implementation of the write-update protocol.

ACKNOWLEDGEMENTS

We would like to thank Aske Plaat and the anonymous reviewers for their very useful comments on the paper. We thank Alan Cox and Honghui Lu for their help with TreadMarks.

Note

A distribution of the Orca and Panda software (for Solaris 2) is available from the World Wide Web, URL http://www.cs.vu.nl/orca/.

References

1. C. Amza, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel, ‘‘Software DSM Protocols that Adapt between Single Writer and Multiple Writer,’’ *Proc. 3rd Int. Symp. on High-Performance Computer Architecture*, San Antonio, TX, pp. 261-271 (Feb. 1997).
2. H.E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and M.F. Kaashoek, ‘‘Portability in the Orca Shared Object System,’’ Technical Report, Vrije Universiteit, Amsterdam, The Netherlands (Sept. 1997).
3. H.E. Bal, R. Bhoedjang, R. Hofman, T. Rühl, C. Jacobs, K. Langendoen, and K. Verstoep, ‘‘Performance of a High-Level Parallel Language on a High-Speed Network,’’ *Journal of Parallel and Distributed Computing* **40**(1), pp. 49-64 (Jan. 1997).
4. H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum, ‘‘Orca: A Language for Parallel Programming of Distributed Systems,’’ *IEEE Trans. on Software Engineering* **18**(3), pp. 190-205 (March 1992).
5. H.E. Bal and A.S. Tanenbaum, ‘‘Distributed Programming with Shared Data,’’ *Proc. IEEE CS 1988 Int. Conf. on Computer Languages*, Miami, FL, pp. 82-91 (Oct. 1988).
6. J.K. Bennett, J.B. Carter, and W. Zwaenepoel, ‘‘Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence,’’ *Proc. Second Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, pp. 168-176 (March 1990).
7. B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon, ‘‘The Midway Distributed Shared Memory System,’’ *Proc. COMPCON 1993*, pp. 528-537 (1993).
8. M. Blumrich, C. Dubnicki, E. Felten, K. Li, and M. Mesarina, ‘‘Virtual-Memory Mapped Network Interface,’’ *IEEE Micro* **15**(1), pp. 21-28 (Febr. 1995).
9. N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawi, C.L. Seitz, J.N. Seizovic, and W-K. Su, ‘‘Myrinet: A Gigabit-per-Second Local Area Network,’’ *IEEE Micro*, pp. 29-36 (Febr. 1995).
10. J. Carreira, J.G. Silva, K. Langendoen, and H. Bal, ‘‘Implementing Tuple Space with Threads,’’ *International Conference on Parallel and Distributed Systems (Euro-PDS’97)*, Barcelona, Spain (June 1997).
11. M. Castro, P. Guedes, M. Sequeira, and M. Costa, ‘‘Efficient and Flexible Object Sharing,’’ *Proc. 1996 Int. Conf. on Parallel Processing (Vol. I)*, Bloomingdale, IL, pp. 128-137 (Aug. 1996).
12. D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, ‘‘Directory-Based Cache Coherence in Large-Scale Multiprocessors,’’ *IEEE Computer*, pp. 49-58 (June 1990).

13. S. Chandra and J.R. Larus, ‘‘Optimizing Communication in HPF Programs on Fine-Grain Distributed Shared Memory,’’ *Proc. 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’97)*, Las Vegas, NV, pp. 100-111 (June 1997).

14. A.L. Cox, S. Dwarkadas, P. Keleher, and W. Zwaenepoel, ‘‘TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems,’’ *Proceedings of the Winter 94 Usenix Conference*, pp. 115-131 (Jan. 1994.).

15. E.W. Dijkstra, ‘‘Guarded Commands, Nondeterminacy, and Formal Derivation of Programs,’’ *Commun. ACM* **18**(8), pp. 453-457 (Aug. 1975).

16. S. Dwarkadas, A.L. Cox, and W. Zwaenepoel, ‘‘An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System,’’ *Proc. 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Cambridge, MA, pp. 186-197 (Oct. 1996).

17. B.D. Fleisch and G.J. Popek, ‘‘Mirage: A Coherent Distributed Shared Memory Design,’’ *Proc. of the 12th ACM Symp. on Operating System Principles*, Litchfield Park, AZ, pp. 211-223 (Dec. 1989).

18. D.S. Greenberg, J.K. Park, and E.J. Schwabe, ‘‘The Cost of Complex Communication on Simple Networks,’’ *Journal of Parallel and Distributed Computing* **35**, pp. 133-141 (June 1996).

19. A.S. Grimshaw, ‘‘Easy-to-Use Object-Oriented Parallel Processing with Mentat,’’ *IEEE Computer* **26**(5), pp. 39-51 (May 1993).

20. A.S. Grimshaw, J.B. Weissman, and W.T. Strayer, ‘‘Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing,’’ *ACM Trans. on Computer Systems* **14**(2), pp. 139-170 (May 1996).

21. M. Haines and K. Langendoen, ‘‘Platform-Independent Runtime Optimizations Using OpenThreads,’’ *Proc. 11th Int. Parallel Processing Symposium*, Geneva, Switzerland, pp. 460-466 (April 1997).

22. S. Ben Hassen and H.E. Bal, ‘‘Integrating Task and Data Parallelism Using Shared Objects ,’’ *10th ACM International Conference on Supercomputing*, Philadelphia, PA, pp. 317-324 (May 1996).

23. K.L. Johnson, M.F. Kaashoek, and D.A. Wallach, ‘‘CRL: High-performance All-software Distributed Shared Memory,’’ *Proc. 15th Symposium on Operating System Principles*, Copper Mountain Resort, CO, pp. 213-228 (Dec. 1995).

24. M.F. Kaashoek, ‘‘Group Communication in Distributed Computer Systems,’’ Ph.D. Thesis, Vrije Universiteit, Amsterdam (1992).

25. P. Keleher and C-W. Tseng, ‘‘Enhancing Software DSM for Compiler-Parallelized Applications,’’ *Proc. 11th Int. Parallel Processing Symposium*, Geneva, Switzerland, pp. 490-499 (April 1997).

26. L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, S. Dwarkadas, and M. Scott, ‘‘VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks,’’ *24th Ann. Int. Symp. on Computer Architecture*, Denver, CO, pp. 157-169 (June 1997).

27. K. Langendoen, J. Romein, R. Bhoedjang, and H. Bal, ‘‘Integrating Polling, Interrupts, and Thread Management,’’ *Proceedings of Frontiers’96*, Annapolis, MD, pp. 13-22 (Oct. 1996).

28. J.W. Lee, ‘‘Concord: Re-Thinking the Division of Labor in a Distributed Shared Memory System,’’ Report TR-93-12-05, University of Washington, Seattle, WA (1993).

29. D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam, ‘‘The Stanford DASH Multiprocessor,’’ *IEEE Computer* **25**(3), pp. 63-79 (March 1992).

30. K. Li and P. Hudak, ‘‘Memory Coherence in Shared Virtual Memory Systems,’’ *ACM Trans. Comp. Syst.* **7**(4), pp. 321-359 (Nov. 1989).

31. S.E. Lucco, ‘‘A Heuristic Linda Kernel for Hypercube Multiprocessors,’’ *Conf. on Hypercube Multiprocessors*, pp. 32-38 (1987).

32. G.V. Wilson and P. Lu (Editors), *Parallel Programming Using C++*, The MIT Press, Cambridge, MA (1996).

33. H. Lu, A.L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel, ‘‘Compiler and Software Distributed Shared Memory Support for Irregular Applications,’’ *Proc. 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’97)*, Las Vegas, NV, pp. 48-56 (June 1997).

34. H. Lu, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel, ‘‘Message Passing Versus Distributed Shared Memory on Networks of Workstations,’’ *Proceedings of Supercomputing ’95*, San Diego, CA (December 1995).

35. H. Lu, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel, ‘‘Quantifying the Performance Differences between PVM and TreadMarks,’’ *Journal of Parallel and Distributed Computing (to appear)* (1997).

36. R.G. Minnich and D.J. Farber, ‘‘Reducing Host Load, Network Load, and Latency in a Distributed Shared Memory,’’ *Proceedings 10th International Conference on Distributed Computing Systems*, Paris, pp. 468-475 (May 1990).

37. R. Nikhil, ‘‘Cid: A Parallel, Shared-memory C for Distributed-memory Machines,’’ *Proc. 7th Ann. Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY (Aug. 1994).

38. S. Pakin, M. Lauria, and A. Chien, ‘‘High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet,’’ *Supercomputing’95*, San Diego, CA (Dec. 1995).

39. L.L. Peterson, N. Hutchinson, S. O’Malley, and H. Rao, ‘‘The x-kernel: A Platform for Accessing Internet Resources,’’ *IEEE Computer* **23**(5), pp. 23-33 (May 1990).

40. M. Raghavachari and A. Rogers, ‘‘Ace: Linguistic Mechanisms for Customizable Protocols,’’ *6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Las Vegas, NV, pp. 80-89 (June 1997).

41. T. Rühl and H.E. Bal, ‘‘Optimizing Atomic Functions using Compile-Time Information,’’ *Working conference on Massively Parallel Programming Models (MPPM-95)*, Berlin, pp. 68-75 (Oct. 1995).

42. T. Rühl, H.E. Bal, R. Bhoedjang, K. Langendoen, and G. Benson, ‘‘Experience with a Portability Layer for Implementing Parallel Programming Systems,’’ *Int. Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’96)*, Sunnyvale, CA, pp. 1477-1488 (August 1996).

43. D.J. Scales, K. Gharachorloo, and C.A. Thekkath, ‘‘Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory,’’ *Proc. 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Cambridge, MA, pp. 174-185 (Oct. 1996).

44. D.J. Scales and M.S. Lam, ‘‘The Design and Evaluation of a Shared Object System for Distributed Memory Machines,’’ *Proc. First USENIX Symp. on Operating System Design and Implementation*,

pp. 101-114 (Nov. 1994).

45. I. Schoinas, B. Falsafi, A.R. Lebeck, S.K. Reinhardt, J.R. Larus, and D.A. Wood, ‘‘Fine-grain Access Control for Distributed Shared Memory,’’ *Proc. 6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pp. 297-306 (Oct. 1994).

46. J.P. Singh, W-D. Weber, and A. Gupta, ‘‘SPLASH: Stanford Parallel Applications for Shared Memory,’’ *ACM Comp. Arch. News* **20**(1), pp. 5-44 (March 1992).

47. E.A. West and A.S. Grimshaw, ‘‘Braid: Integrating Task and Data Parallelism,’’ *Proc. Frontiers 95*, McLean, VA, pp. 211-219 (Feb 1995).

48. G.V. Wilson and H.E. Bal, ‘‘An Empirical Assessment of the Usability of Orca Using the Cowichan Problems,’’ *IEEE Parallel and Distributed Technology* **4**(3), pp. pp. 36-44 (Fall 1996).