Object Distribution in Orca
using Compile-Time and Run-Time Techniques

Henri E. Bal[1]

Vrije Universiteit
Dept. of Mathematics and Computer Science
Amsterdam, The Netherlands
bal@cs.vu.nl

M. Frans Kaashoek[2]

M.I.T. Laboratory for Computer Science
Cambridge, MA
kaashoek@lcs.mit.edu

ABSTRACT

Orca is a language for parallel programming on distributed systems. Communication in Orca is based on shared data-objects, which is a form of distributed shared memory. The performance of Orca programs depends strongly on how shared data-objects are distributed among the local physical memories of the processors. This paper studies a new and efficient solution to this problem, based on an integration of compile-time and run-time techniques. The Orca compiler has been extended to determine the access patterns of processes to shared objects. The compiler passes a summary of this information to the run-time system, which uses it to make good decisions about which objects to replicate and where to store nonreplicated objects. Measurements show that the new system gives better overall performance than any previous implementation of Orca.

[1] This research was supported in part by a PIONIER grant from the Netherlands Organization for Scientific Research (N.W.O.).

[2] This research was done while the author was at the Vrije Universiteit.

This paper will be published in the proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '93), 26 September - 1 October 1993, Washington D.C.

1. INTRODUCTION

Shared objects are an attractive alternative to message passing for programming distributed systems. The shared object model provides a form of logically shared data, which makes programming easier. The main problem, however, is how to implement shared objects efficiently on distributed-memory architectures. A key issue is how to solve the problem of mapping logically shared data onto physically distributed memories. This problem can be left to the user, the compiler, the language run-time system (RTS), the operating system, the hardware, or a combination of these.

In this paper we will introduce a novel approach to the data distribution problem based on a collaboration between the compiler and RTS. This method has been designed for the Orca programming language [1, 2]. Orca is a language for implementing coarse-grained, explicitly-parallel applications on distributed systems, such as collections of workstations connected by a Local Area Network. Logically shared data structures in Orca are encapsulated in variables of abstract data types, called *shared data-objects* [3], which have user-defined indivisible operations. Objects in Orca can be

migrated or replicated by the system, without any intervention from the user.

Orca has been designed to hide data distribution from the programmer, but to allow a compiler and RTS to implement shared objects efficiently. Distribution is hidden from the programmer, because it is a difficult, implementation- and architecture-dependent problem. In the past, we have studied two implementations of Orca's shared objects. A simple approach is to replicate shared data on all processors. The key idea that makes this scheme reasonably efficient is to update the replicas using a fast reliable broadcast protocol. Another solution we have looked at uses heuristic rules based on run-time statistics of the access patterns to shared data to decide where to store or replicate shared data. The problem here is how to find heuristics that work well for all applications. A comparison of these two approaches has shown that in most (but not all) cases, full replication performs better [4].

To a certain extent, both solutions benefit from information generated by the compiler. For example, the compiler recognizes read-only operations on shared objects, which are usually executed locally by the RTS, thus reducing the number of messages generated. However, Orca has been designed to allow much more advanced compiler optimizations. Shared data in Orca are always accessed through user-defined operations on abstract objects, rather than through low-level instructions. Also, features such as pointers and global variables, which complicate compile-time analysis of programs, have been deliberately omitted from Orca. These two properties allow the compiler to determine how shared data-objects are accessed.

With the new approach to the data mapping problem in Orca, the RTS relies on the compiler to generate information about how objects are used. The chief advantage of relying on the compiler is that the compiler can *look ahead* into the code and predict how shared objects are going to be used. In contrast, an RTS can only collect information from the past.

The most important new aspect of our work is the way the compiler and RTS together hide object distribution from the programmer. The compiler determines how shared objects are used by each type of process. This information is passed to the RTS. Whenever a new Orca process is forked, the RTS uses the compiler-generated information to determine whether the current mapping of objects is still desirable. If not, it may decide to start or stop replicating an object or to migrate an object from one processor to another.

Our goal in this paper is to show that with this approach the compiler and RTS can often make the right decisions concerning data distribution. Of course, as with any approach based on compile-time predictions, programmers will always be able to write programs for which the approach is ineffective. However, our experience with the compiler and RTS so far has been most encouraging. This implementation gives better overall performance than any previous Orca implementation.

The issue of data distribution is also addressed in several other languages and systems. In many parallel languages for numerical programming, the user indicates how shared arrays are to be decomposed and mapped onto the different processors [5, 6, 7]. In these languages, parallelism itself is often dealt with by the compiler, by analyzing FOR-loops, but data distribution is left (at least partly) to the programmer. The focus or our work, however, is more on non-numerical applications. Also, our goal is not to parallelize dusty deck programs, but to use modern language techniques for writing explicitly-parallel programs.

The Shared Virtual Memory of Li and Hudak [8] implements data distribution in the operating system, by partitioning the shared address space into fixed-size pages that are moved and copied transparently among the local memories of the processors. Linda [9] also hides data distribution from programmers. Replication and partitioning of the Linda Tuple Space is mainly taken care of by the RTS, sometimes also using information generated by the Linda compiler. In Munin [10] the RTS takes care of data mapping, using annotations from the user about how shared variables are accessed. In some object-based languages the RTS

provides the mechanisms for data distribution, but the actual decision to move objects is usually made by the programmer, using annotations or special statements [11, 12]. Finally, multiprocessor systems such as Dash [13] and Alewife [14] try to solve the data distribution in hardware.

The structure of the rest of this paper is as follows. In Section 2, we briefly describe the Orca language and its implementation. In Section 3, we give an overview of the new compiler and RTS. Next, in Section 4, we describe the implementation of the compiler and RTS. In Section 5, we report on our experiences with the new system and give performance results. In Section 6, we discuss the advantages and disadvantages of our approach and we describe future research. In Section 7, we give a more detailed comparison with related work. Finally, in Section 8, we present some conclusions.

2. THE ORCA LANGUAGE AND ITS IMPLEMENTATION

In this section we will give a brief description of the Orca language and its implementation. The goal is to give just enough detail to make the rest of the paper understandable. More detailed descriptions are given elsewhere [1, 2, 3].

2.1. Orca

Orca is a language for writing parallel programs for systems that do not have physical shared memory. Processes in Orca communicate through *shared data-objects*, which are instances of abstract data types. Processes can share objects even if they run on different machines. The objects are accessed solely through the operations defined by the abstract data type. The following trivial example specifies an object type *IntObject* encapsulating a single integer:

```
object specification IntObject;
        operation Value(): integer;
          # return value
        operation Assign(v: integer);
          # assign new value
end;
```

The implementation part (not shown here) contains the data used to represent instances (variables) of the type, the implementation code of the operations, and code for initializing variables of the type. Objects are created and used as follows:

```
X: IntObject;
   # create (declare) an object
X$Assign(3);
   # apply operation "Assign" to X
```

An Orca program also defines one or more process types, which are similar to procedure declarations:

```
process p(n: integer; X: shared IntObject);
begin ... end;
```

Processes of type *p* take two parameters: a value parameter *n* and a *shared* (or *call-by-reference*) parameter *X*. Only objects may be passed as shared parameters. Processes are created dynamically, through a **fork** statement, which specifies the process type, the actual parameters, and (optionally) the processor on which the new process is to be run. The statement "**fork** p(23, X) **on**(3);" creates a new process on CPU 3, passing 23 as value parameter and object *X* as shared parameter. Passing objects as shared parameter in a **fork** statement is the only way to express sharing in Orca, since there are no globally shared objects.

The semantics of the model are straightforward. All operations are applied to single objects, making the model efficient to implement. Operations are executed sequentially consistent [15]. They are also executed indivisibly, which simplifies programming, since mutual exclusion synchronization is done automatically [2].

Although Orca's communication model is based on objects, Orca is not an object-oriented language. It does not support inheritance or dynamic binding, and neither does it treat all entities as objects. Our primary research objectives are the design of an easy-to-use communication and synchronization model and its efficient distributed implementation. Integrating this model in an object-oriented framework thus is not our major concern, although we do have an experimental version of our model embedded in C++ [16].

2.2. A distributed implementation of Orca

The Orca compiler partitions operations on objects into two classes: *read operations* and *write operations*. A read operation is an operation that does not modify the object's local data. All other operations are write operations. A write operation may also read the object, but in contrast to a read operation it potentially modifies the object.

Orca can be implemented efficiently on a distributed system by replicating shared objects in each processor's local memory [3]. If a processor has a local copy of an object, it can do read operations locally, without doing any communication. Write operations are broadcast to all nodes containing a copy. All these nodes update their copy by applying the write operation to the copy. We use Amoeba's [17] *totally-ordered* reliable broadcast protocol [3, 18], which delivers all messages reliably and in the same order at all receivers. Because the broadcast is totally-ordered, copies of objects are updated in a consistent way.

Our broadcast protocol uses the hardware multicast or broadcast facility of the underlying network, if available, and is most efficient on such networks. Many networks have hardware support for multicast, for example Ethernet and some ring networks. Future Gigabit LANs [19] and ATM switches [20] are also expected to support multicast or broadcast.

An important issue is the *replication strategy* [4]. The most efficient RTS we had implemented so far simply replicates shared objects on all processors. We have used this RTS as a starting point for our present work and we will also compare the performance of the new system with this RTS.

3. OVERVIEW OF THE NEW COMPILER AND RUN-TIME SYSTEM

The Orca implementation described above works reasonably well for most applications. The approach of replicating all objects everywhere is efficient for programs in which data are transferred from one process to all other processes. Unfortunately, there are also cases where the approach is less efficient. In particular, full replication is inefficient if an object is shared by only a few processes or if it has only write operations.

Our new system uses a different strategy. For each object, the new system decides:

- Whether or not to replicate the object
- For non-replicated objects: *where* to store the object.

So, objects are either replicated everywhere or are stored on exactly one processor. As we will see, the two decisions may change dynamically, when new processes are created.

An important design issue is where to make these decisions, in the compiler or RTS. Although in many cases the compiler is able to make the right decision without any help from the RTS, there are two disadvantages of such an approach. First, the compiler will have to do complicated analysis, especially if dynamic arrays of objects are used. Second, and more fundamental, the optimal strategy in general depends on the underlying hardware architecture and communication protocols. The compiler would then become architecture-dependent, which we want to avoid.

With the approach described here, the compiler does the bulk of the analysis, but the RTS makes and implements the actual decisions. For each type of process, the compiler analyzes how processes of this type access shared objects. If a process creates an object itself, the compiler generates a call to the run-time routine *Score*, which tells the RTS how the process is going to use the new object. In addition, the compiler generates a *process descriptor* for every process type, showing how processes of this type will access parameters that are shared objects. As a trivial example, consider the declaration of a process type *AddOne* with two shared-object parameters:

```
process AddOne(X, Y: shared IntObject);
begin
        Y$Assign(X$Value() + 1);
          # X is read once,
          # Y is written once
end;
```

The compiler will determine that processes of type *AddOne* will read their first parameter once and write their second parameter once. The compiler stores this information in the process descriptor for *AddOne*. After a fork statement of the form "**fork** AddOne(A, B) **on**(cpu);" the RTS uses this descriptor to determine that the new process will read *A* once and write *B* once.

Equipped with this information about new processes and objects, the RTS chooses a suitable replication strategy for each object. The implementation of the new RTS is described in Section 4.2. Since the RTS is distributed among multiple processors, some interprocess communication will be needed during the decision making. As we will see, however, the number of messages needed is small and the communication overhead is negligible.

Let us now describe in more detail the information that the compiler passes to the RTS and the method used to compute this information. For the distribution strategy described in this paper it would be sufficient just to estimate the *ratio* of read operations and write operations executed by each process. For more advanced optimizations, however, it is useful to have the access patterns themselves available. For some optimizations it makes a difference whether a process alternates read operations with write operations or first does *n* reads and then *n* writes. In both cases the read-write ratio will be 1, but the access patterns are very different.

The compiler therefore computes, for each process, the actual access pattern. It first generates a description of how the process reads and writes its shared objects, taking into account the control flow of that process. Based on this information, the compiler computes two values for each object the process can access:

Nreads	An estimate of the number of read operations.
Nwrites	An estimate of the number of write operations.

These values are passed to the RTS. In the future, we may want to pass the actual access patterns to the RTS, to make better decisions at run-time.

4. IMPLEMENTATION OF THE NEW COMPILER AND RUN-TIME SYSTEM

In this section we describe the most important modifications made to the compiler and RTS to implement the new replication strategy.

4.1. The New Compiler

We will first describe the read-write patterns used by the new compiler, and then discuss how the patterns are generated and analyzed. The patterns give a static description of how each Orca process accesses its shared objects. The pattern does not contain information about nonshared objects or normal (local) variables. As an example, consider the Orca code fragment of Figure 1.

```
function foo(X: shared IntObject);
        i: integer;
begin
        for i in 1 .. 100 do X$Assign(i);
end;

process bar(P: shared IntObject);
        A: IntObject;
        tmp: integer;
begin
        P$Assign(10);
        if cond then
                tmp := A$Value();
        else
                foo(A);
        fi;
end;
```

Figure 1. An Orca code fragment.

The read-write pattern for process *bar* is the following regular expression:

process bar: #1$W ; [A$R | {A$W}]

which specifies that the process will first write its first parameter ("#1") and then either read its object *A* once or it will repeatedly write *A*. (Selection is indicated by square brackets and the vertical bar; for repetition, curly brackets are used.) The original *Value* and *Assign* operations defined by the

programmer have been replaced by a *R* (for Read) and *W* (for Write) respectively. The function *foo* is absent in the pattern, although its effects (repeatedly writing *A*) are included.

The compiler computes the read-write patterns in two phases. It first performs a local analysis of each process and function, and then computes the final patterns by simulating inline substitution. Computing the patterns is straightforward, since features that make control flow and data flow analysis difficult, such as "goto" statements, global variables, and pointers have been intentionally omitted from Orca. Also, process types are part of the language syntax, so techniques such as separating control flow graphs [21] are not needed. The only difficulty is handling recursive (or mutually recursive) functions. The current compiler stops simulating inline substitutions after a certain depth. A more advanced implementation should at least handle tail-recursion similarly to iteration.

The second phase is done by the pattern analyzer. It analyzes the pattern generated for each process type and determines the *Nreads* and *Nwrites* values for the shared objects the process can access. It considers each process type in turn. A process can access any shared objects declared by itself or passed to it as shared parameter. For each such object the analyzer determines how the object is used by the process. It computes an indication of how many times the process will read and write the object. It uses the heuristic that operations inside a loop will be executed more frequently than operations outside a loop, so it multiplies their values by a constant factor (currently set to 16). Also, operations inside a selection (**if**-statement) are less likely to be executed than operations outside a selection, so their values are halved.

For example, if a process has the following pattern:

{ X$W; Y$W; Z$W; {X$R}; [Y$R]; Z$R }

the analyzer determines that *X* is mostly read, *Y* is mostly written, and *Z* has about as many reads as writes. Based on this information, the analyzer computes *Nreads* and *Nwrites* for the process and passes these values to the runtime system.

4.2. The New Run-Time System

Based on the information passed to it by the compiler, the RTS decides whether or not to replicate objects and where to store those objects that are not replicated. If a process invokes an operation on a nonreplicated shared object stored on a remote processor, the new RTS sends the operation and its parameters to this machine, asking it to execute the operation and return the results. For a nonreplicated shared object stored locally, the RTS simply executes the operation itself, without telling other processors about this event.

Which replication strategy is optimal depends on the underlying architecture and communication protocols. The RTS we will consider here runs on a collection of processors connected by an Ethernet. It uses Amoeba Remote Procedure Calls for point-to-point communication and updates replicated objects using Amoeba's reliable broadcast protocol (mentioned in Section 2.2). The protocol is efficient for this architecture, since it uses the Ethernet hardware multicast facility. So, in this RTS, updates are relatively cheap, making replication attractive.

The RTS on each CPU maintains state information for each shared object. The state of an object X includes the following information for *every* processor P:

1. The total number of reads for all processes on P.
2. The total number of writes for all processes on P.

Both numbers are estimates, based on the *Nreads* and *Nwrites* values generated by the compiler. Each processor keeps track of how every processor uses each object. This information is updated on every **fork** statement. Although a **fork** statement creates a process on only one processor, **fork** statements are always broadcast, using the same totally-ordered broadcast protocol as for operations. Hence, the state information on all processors is always consistent and all processors will make the same decisions.

As an example, assume CPU 0 executes a statement

```
fork AddOne(A, B) on 2;
  # AddOne was defined in §3
```

The RTS on CPU 0 broadcasts a message "[FORK, AddOne, A, B, 2]" to all processors. Each RTS receives this message and updates the state information for objects A and B on CPU 2. The *Nreads* value for object A is incremented by one, since A is read once by *AddOne*. Also, the *Nwrites* value of B is incremented by one, since B is written once. Based on this updated information, all processors now reconsider their decisions for A and B, using heuristics explained below. If A was not replicated before, the improved read-write ratio may now reverse this decision. Likewise, if B was replicated, the lower ratio may now cause it to drop the replicas. In the latter case, all processors also decide where to store the single copy of B. If A or B were stored on only one processor, the system may also decide to migrate these copies.

In any case, due to the totally-ordered reliable broadcast, all processors come to the same decisions. If, for example, they decide to no longer replicate B but store it only on CPU 5, all processors except for CPU 5 will delete their copies of B. The processor on which the new process is forked (CPU 2 in our example) will also install copies of A and B (if required) and create a new process.

The heuristic for choosing the replication strategy is quite simple. Let R_i and W_i be the *Nreads* and *Nwrites* values for CPU i for a given object X. First, if X is not replicated we store it on the processor that most frequently accesses it, which is the CPU with the highest value for $R_i + W_i$. This processor is called the *owner* of X.

To decide whether or not to replicate object X, we compare the number of messages that will be generated with and without replication. With replication, one broadcast message will be generated for each write operation by any process. We compute the number of write operations by summing W_i for all i. Without replication, one Remote Procedure Call will be needed for every access by any processor except for the owner. We determine the processor that would become the owner (i.e., has the highest value for $R_i + W_i$) and sum $R_i + W_i$ for all i except this owner. Finally, we compute the total communication costs for both cases, using the average costs for RPC and broadcast messages. For the platform we use (see Section 5), an RPC costs about 2.5 msec and a broadcast 2.7 msec. We compare these costs and choose the strategy that has the least communication overhead.

The overhead of this decision making process is actually quite small. It is only needed for objects that are shared among multiple processes. The number of shared objects in Orca programs is usually small, so the state information requires little memory. The state is only updated after **fork** statements.[3] As Orca programs use course-grained parallelism, such statements occur infrequently. With replicated workers parallelism, for example, there is about one **fork** statement for every processor. The only disadvantage of the current implementation is that all **fork** statements must be broadcast.

As stated before, in many cases the decision making process could also be done in the compiler instead of the RTS. We have in fact also implemented such a compiler, and it can make the right decisions most of the time. However, the RTS has more accurate information about which objects each process accesses. Therefore, there exist cases in which the RTS can make good decisions and the compiler cannot. As an example, if an element of an array of objects is passed as a shared parameter to a new process, the compiler may not be able to determine what the value of the index expression is, thus making static analysis hard if not impossible. So, the solution we have chosen is more general and simplifies the compiler analysis, at the cost of a small run-time overhead.

Since the replication strategy may change dynamically, new mechanisms had to be implemented in the RTS. The new RTS must be able to

[3] Strictly speaking, the state should also be updated after a process exits, but we did not implement this yet. For all applications mentioned in this paper, however, all processes continue to exist until the whole program terminates, so there would be no difference in performance.

drop the replicas of a replicated object, to replicate an unreplicated object, and to migrate an object from one processor to another. The latter case occurs if the RTS decides to change the owner of a nonreplicated object. The protocols implementing these mechanisms are relatively straightforward, as Orca has only passive objects and since simple communication primitives with clean semantics (RPC and totally-ordered broadcast) are used. In Emerald, for example, object migration is more complex, because objects may contain processes [11].

5. PERFORMANCE OF THE NEW SYSTEM

We have used the new system for several Orca applications. Here, we will look at three such applications. We study the first application in detail and we give the execution times and speedups for all programs for the RTS based on full replication and for the new system. A more precise analysis of the performance (in terms of number of messages sent and their sizes) is given in [18]. The Orca systems used for the measurements run on top of the Amoeba distributed operating system. The hardware we use is a collection of 20MHz MC68030s connected through a 10 Mbit/s Ethernet.

5.1. The Traveling Salesman Problem

The Traveling Salesman Problem (TSP) requires to find the shortest route for a salesman to visit each city in a given set exactly once. The problem is solved in Orca using a master/slave type of program based on a branch-and-bound algorithm. The master generates partial routes and stores them in a job queue. Each slave repeatedly takes a job (route) from the queue and generates all possible full paths starting with the initial route and using the "closest-city-next" heuristic. All slaves keep track of the current shortest full route. As soon as a slave finds a better route, it gives the length of the route to all other processes. This value is used to prune part of the search tree. The Orca program uses the following shared objects:

q — A queue containing all generated jobs that have not yet been handled by a slave.

min — The length of the best route found so far.

These objects are created by the master process and passed as shared parameters to all slave processes, using the statement "**fork** slave(q, min);". The patterns generated for the master and slaves are as follows.

```
process master:
  min$W;     initialize global minimum
  {q$W};     generate the jobs
  min$R;     get final value of minimum

process slave:
  { #1$W;    get job & delete it from q
    [#2$R; [#2$W | [
       {[#2$R; [#2$W | ]  | ] } | ]  ] | ]
              execute job
  }
```

The second pattern is somewhat complicated, because the slave process uses a recursive function. The pattern generator estimates the effect of this function by expanding it up to a certain depth. (Here we have used depth 2; in practice, a higher depth is used.) Also note that the notation "[pattern |]" represents an **if**-statement (selection); the **then**-part is represented by the given pattern and the **else**-part does not contain any operation invocations.

First consider the job queue. Both the master and the slaves apply only write-operations to this object, since adding and deleting jobs both change the queue's data structures. Therefore, object *q* of the master and the first parameter of the slaves will be assigned positive values for *Nwrites*. The *Nreads* value for these objects will be zero. The RTS will therefore decide not to replicate this object. The master process runs on the same CPU as one of the slave processes (since the master is not computationally expensive). The RTS will store the queue object on this CPU, since it contains two processes that write the object.

Now consider the second object, *min*, which contains the global minimum. The master reads and writes the minimum exactly once. The pattern for the slaves is complicated, but it is still easy to see

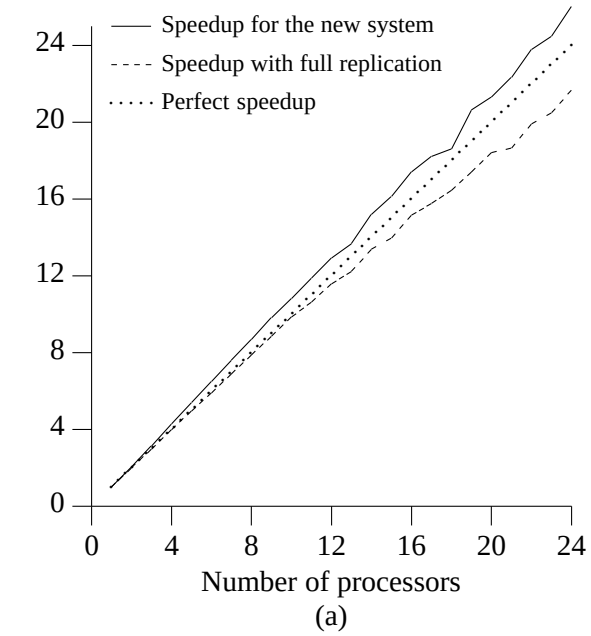
that the average read-write ratio for this object (parameter #2) is higher than 1. (Within the outermost selection, the pattern contains a read operation, followed by a nested selection; the write operation in this nested selection can only be executed if the read operation has been executed first). In the slave processes, the *Nreads* value will therefore be higher than the *Nwrites* value. Consequently, the RTS will decide to fully replicate this object.

The net effect is that the job queue is no longer replicated but stored on the CPU where the master runs. This decision reduces the communication overhead of the program, because a slave can now get a job without troubling other processors. In other words, the operation to get a job will be sent point-to-point instead of being broadcast. The global minimum is still replicated, which is important since, in practice, it is read much more frequently than it is written [2].

The compiler and RTS make the right decisions for both objects. The impact of the new strategy on the speedup and absolute execution time for a randomly generated problem with 14 cities is shown in Figure 2. (For this problem, the master generates 1716 jobs, each containing a partial route with 4 cities.) This figure gives the speedups and execution times for the original system based on full replication and for the new system. The two systems differ only in their replication strategy: the system based on full replication replicates all objects everywhere, while the new system uses the strategy discussed in this paper. The speedups shown in the figure are relative to the parallel Orca program running on one CPU.

The performance improvement is significant, especially for a large number of processors. The new system even achieves superlinear speedup. This is due to the fact that one processor quickly finds a close to optimal shortest path, which other processors use to prune parts of their search tree.

Most TSP problems we generated give superlinear speedup, which indicates that our single-processor algorithm (based on the "closest-city-next" heuristic) is not optimal. We certainly do not claim that the superlinear speedup is due solely to our new distribution strategy. The point we do wish to make, however, is that the new system performs significantly better than the one based on full replication.



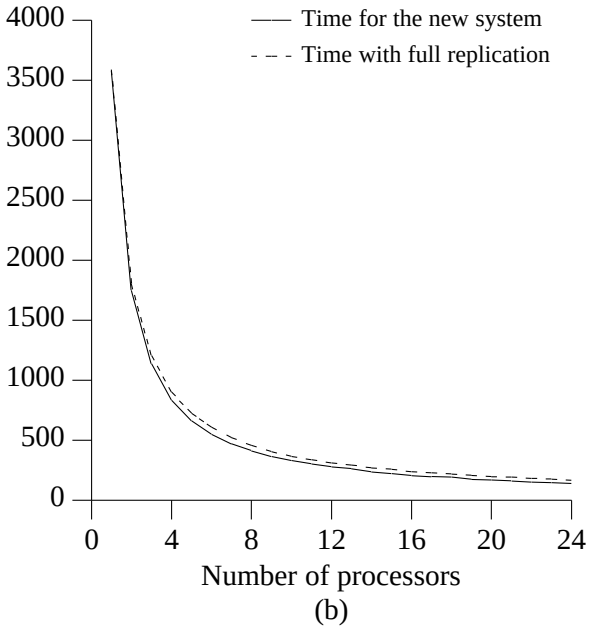


Figure 2. Performance for the Traveling Salesman Problem using a 14-city problem. (a) Speedups for each implementation of the RTS. (b) Absolute running times in seconds for each implementation of the RTS.

5.2. The All-Pairs Shortest Paths Problem

The second application we consider is the All-Pairs Shortest Paths problem (ASP). In this problem, it is desired to find the length of the shortest path from any node *i* to any other node *j* in a given graph with *N* nodes. The parallel algorithm we use is similar to the one given in [22], which itself is a parallel version of Floyd's algorithm.

The distances between the nodes are represented in a matrix. Each processor contains a *worker* process that computes part of the result matrix. The parallel algorithm performs *N* iterations. Before each iteration, one of the workers sends a *pivot row* of the matrix to all the other workers. Since the pivot row contains *N* integers and is sent to all processors, this requires a non-trivial amount of communication.

The pattern for the worker processes is shown below (the master process does not access any of the important shared objects):

```
process worker:
  {[#1$W | #1$R]}
     # parameter 1 is the object Rowk
```

The master process forks a number of workers, passing an object called *Rowk* as a shared parameter (parameter #1). This object is used for transferring the pivot rows. The process containing the pivot row stores this row into the shared *Rowk* object, where it can be read by all other processes.

The pattern for the workers clearly reflects this style of communication. A worker executes zero or more iterations, and during each iteration it either reads or writes the *Rowk* object. A worker writes the object if it contains the pivot row for the current iteration, else it waits until another worker has put the row in the object and then reads the row.

As far as the pattern analyzer can see, the expected read/write ratio of worker processes for the *Rowk* object is exactly 1, because it does not know which of the two alternatives will be executed most frequently. Using more aggressive optimization techniques (or perhaps even execution profiles), it might be possible to make a more accurate estimate. For *Rowk*, the compiler will therefore pass to the RTS a value for *Nread* that is equal to *Nwrite*. When in doubt, the RTS always adheres to the original replication strategy, which is to replicate objects everywhere. For ASP, replicating the object is essential, because it means that the pivot rows will be broadcast instead of being sent point-to-point. The performance of ASP will therefore be the same with the optimized and unoptimized compilers, as shown in Figure 3.

5.3. Successive Overrelaxation

Successive overrelaxation (SOR) is an iterative method for solving discretized Laplace equations on a grid. During each iteration, the algorithm considers all non-boundary points of the grid. For each point, SOR first computes the average value of its four neighbors and then updates the point using this value.

Our implementation of SOR in Orca is based on the parallel Red/Black SOR algorithm used for the Amber system [12]. The grid is partitioned among a number of worker processes, one per processor. Each worker contains a vertical slice of the grid. The processes are organized in a linear row. At the beginning of an iteration, each worker needs to exchange edge values with its left and right neighbor. This can easily be implemented through shared buffer objects. We use two buffer objects for each pair of neighbors, for communication in each direction. Since each worker (except the first and last) has two neighbors, each worker accesses four shared objects, which are passed as parameters. The compiler will have assigned a positive *Nwrites* value to each parameter and a zero *Nreads* value, since the worker will apply only write-operations to the objects. (Both storing data into the buffer and taking data out of it modify the buffer's internal data structures, so both are write operations.)

The RTS will therefore discover that each object is used on two processors and that it is written (and not read) by both of them. So, the RTS will decide not to replicate these objects, and stores each object on one of the processors that accesses it, the latter choice being arbitrary but consistent, so the objects get evenly distributed among the processors.

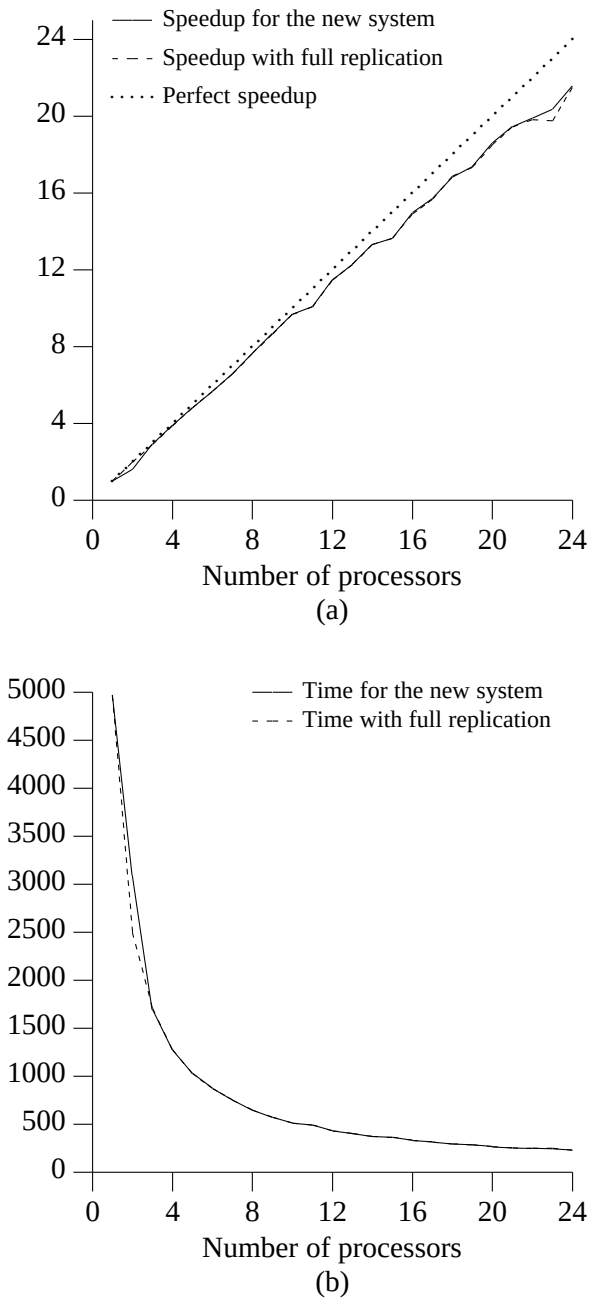


Figure 3. Performance for the All-Pairs Shortest Paths Problem using an input graph with 500 nodes. (a) Speedups for each implementation of the RTS. (b) Absolute running times in seconds for each implementation of the RTS.

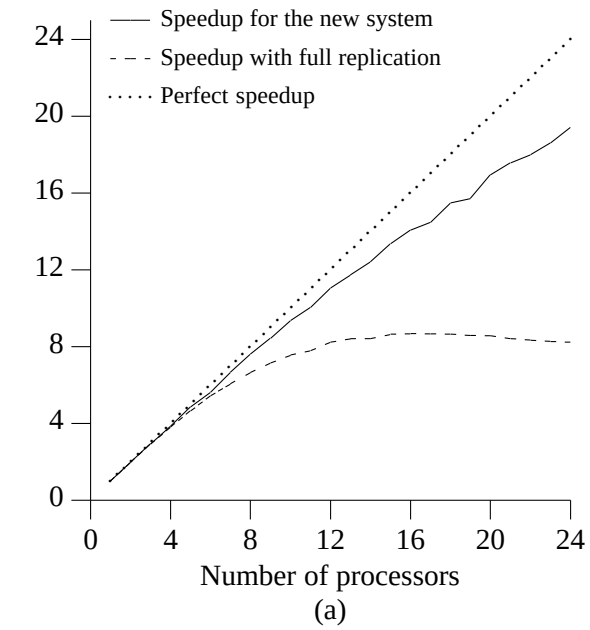
What we have gained is that each buffer object is now stored on only one processor, rather than being replicated on all. Communication between two neighboring processes is now implemented with point-to-point messages instead of broadcasting. The new strategy improves the speedup and execution time of the program substantially, as is shown in Figure 4. With more than 16 processors, the performance of the program with full replication even decreases, due to the large number of broadcast messages received by each processor. With the new distribution strategy, the program does not have this problem.

6. DISCUSSION

The implementation of shared objects in Orca described above integrates compile-time and run-time techniques. The compiler determines the access patterns of processes to objects and the run-time system makes the actual decisions regarding object placement and replication. We have looked at the performance of this system for three applications, two of which are suitable for our model of replicated objects, and a third (SOR) which uses essentially a message passing algorithm implemented with buffer objects. Below, we will first study the efficiency of our approach. Next, we will discuss portability and we will look at further optimizations that we may add in the future.

Let us first take a look at the execution time and memory overhead of our implementation. The only significant execution time overhead of making decisions dynamically is due to the fact that **fork** statements are always broadcast. Since we are not aiming at fine-grained parallelism, however, such statements are executed infrequently. The overhead of broadcasting a **fork** statement is roughly the same as that of a write operation on a replicated object, which is about 4 msec for operations with 4 bytes of parameters and 8 msec for operations with 1Kb of parameters [4]. Normal operations are not slowed down by the decision making process; the overhead only occurs during **fork** statements.

Regarding memory overhead, the compiler recognizes most objects that are not shared, so the RTS does not keep track of the *Nreads* and *Nwrites* values. Local objects therefore do not have any bookkeeping overhead. The number of shared objects in Orca programs usually is small. In TSP



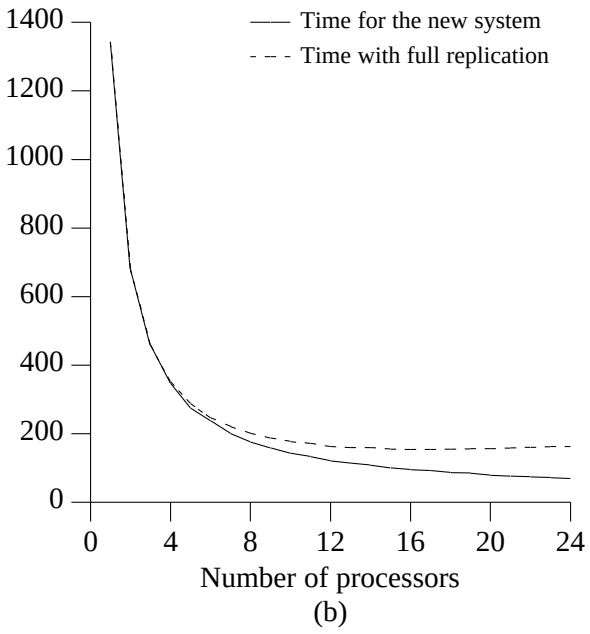


Figure 4. Performance for Successive Overrelaxation for an 80 by 482 grid. (a) Speedups for each implementation of the RTS. (b) Absolute running times in seconds for each implementation of the RTS.

and ASP the number is fixed and in SOR it is approximately twice the number of CPUs used. So, the total memory overhead is also small. In object-oriented languages, where all entities are objects, the memory overhead might become more severe, although object clustering techniques might be used to reduce it. Although the typical memory overhead for Orca is low, we are nevertheless investigating methods to further reduce the amount of state at each processor and the cost for decision making to allow programs to scale to large numbers of processors and processes.

We should point out that our current system supports objects of any grain size, as defined by the user. The *min* object in TSP, for example, encapsulates only a single integer, so it is very fine-grained. For this object, the relative memory overhead is of course high, and there also is some execution time penalty for using an object type instead of a regular integer. Still, the ability to share the global bound more than compensates this disadvantage, since it allows efficient pruning of the search tree [1].

Our scheme thus makes good decisions for object distribution at acceptable runtime costs. Let us also look at some disadvantages of our approach, however. One problem with the system described here is that it replicates objects either everywhere or nowhere. For some applications, a partial replication scheme is better, in which objects are replicated on some —but not all— processors. Another problem is that parallel programs may consist of different *phases*, each with their own access patterns. The optimal approach would be to use a different replication strategy for each phase. Our current system cannot handle such cases in an optimal way, unless the programmer creates a new set of processes for each phase.

Besides performance, an important issue is that of *portability*. Our current system runs on a collection of processors connected by an Ethernet. On such a platform, point-to-point communication is slow, but broadcasting can be implemented reasonably efficiently. On message-passing multicomputers, broadcasting may be relatively expensive, although it can often be simulated using a spanning-tree algorithm.

If broadcasting is expensive, the RTS should be less eager to replicate objects. The RTS therefore should know the relative costs of broadcasting

and point-to-point messages, so it can use no replication (or partial replication) whenever that is more efficient for a given object.

We are currently working on such a portable implementation of Orca, based on the approach described in the paper, and intended to run on a variety of distributed systems and multicomputers. An important advantage of our approach is that architecture-dependent issues are localized in one part of the RTS. The compiler, for example, just predicts the behavior of the processes, but it does not know anything about the architecture on which they will run. Hence, we think we can easily adapt our system to different platforms.

We believe that our approach of explicitly computing read-write patterns opens a whole area for research on further optimizations as well. One important class of such optimizations is *adaptive caching*, proposed in the Munin project [10]. The idea is to classify shared objects based on the way they are used and to use different coherence mechanisms for different classes of objects. The current Munin implementation leaves the classification up to the programmer. For Orca, we hope to be able to do a similar classification automatically, so the programmer still sees a sequentially consistent memory model. The compiler can easily obtain synchronization information from the operations and use this together with the read-write patterns to classify objects. With such optimizations, we hope to be able to hide part of the communication latency. We could hide latency further by continuing computation while sending out an operation, if possible. Other architecture-dependent optimizations like using split-phase transactions as implemented on the CM-5 could also be exploited [23].

7. RELATED WORK

Most of the work on compiler optimizations for distributing shared data takes place in the area of numerical applications, where shared arrays have to be decomposed and partitioned among the memories of different machines. In FORTRAN-D [5], for example, the programmer can specify a strategy for decomposing arrays into blocks. The compiler uses this information to distribute the array among the physical memories and automatically generates send/receive primitives when needed. Parallelism in such programs can be obtained by executing different iterations of a loop on different processors, or by performing higher-level operations (e.g. matrix additions) in parallel. Such systems usually adhere to the Single Program Multiple Data (SPMD) [24] style. A wide body of work exists on parallelizing array-oriented programs that run on distributed-memory message-passing machines [6, 7, 25, 26, 27, 28, 29]. Also, some work on functional languages is related to this approach [30, 31].

Our work on Orca is much less focused on numerical applications and partitioned arrays. Parallelism in Orca is explicit (through a fork statement) and many forms of synchronization can be expressed, so Orca programs are not necessarily SPMD-like. Objects in Orca are not partitioned but replicated. Also, this replication is transparent to the user (except for performance). So, the goals of the new Orca compiler and the techniques used are different from those of the languages mentioned above.

Totty and Reed run a number of trace driven simulations to determine whether data structure-specific data management is superior to a single system-imposed policy [32]. Like our results their trace driven simulations show that a data structure-specific data management policy is better than a single system-imposed policy. Totty and Reed, however, do not address how a policy is selected and who is selecting it.

A system related to Orca is Linda [9], which is also based on explicit parallelism and communication through shared data. Linda uses extensive compile-time optimization [33], but this is mainly aimed at reducing the overhead of associative addressing of Tuple Space. As far as we know, however, no existing Linda system integrates compile-time and run-time optimizations in the same way as our system does.

Although Orca's communication model is based on objects, the language differs in many important ways from object-oriented languages such as Emerald [11] and POOL-T [34]. Objects in Orca

are passive and are replicated automatically. Also, in Orca all operations on objects are guaranteed to be indivisible. In most concurrent object-oriented languages, objects can be active, are not replicated, and communicate by sending messages to each other, rather than through indivisible operations.

Distributed Shared Memory (DSM) systems also support logically shared data on distributed-memory machines, often by simulating a physical shared memory. The best-known example is Kai Li's Shared Virtual Memory [8], which is a page-based DSM system. Page-based DSM systems, however, have important performance disadvantages when compared to object-based systems such as Orca [3]. Several modern DSM systems address this problem and increase performance by relaxing the semantics of the memory model. Examples of such systems are Munin [10], Midway [35] and several others [36]. Also, a recent extension of the Amber language has been proposed supporting version consistency for shared objects [37].

Orca does not relax the semantics of the model, since we believe that it would complicate programming. Orca therefore supports sequential consistency [15] and leaves it to the implementation to obtain high performance, by using techniques like the ones discussed in this paper.

8. CONCLUSIONS

We have described a new approach for implementing Orca's shared data-objects in a distributed environment. The approach uses compile-time analysis to determine how objects are used by the different processes. This information is passed to the RTS, which decides which objects to replicate and where to store nonreplicated objects. We have applied our method to several existing Orca applications. For all these applications, the compiler and RTS made the right decisions, leading to either the same or better performance for the application. The new system obtains significantly better speedups for two of the three applications discussed in this paper,

Nonetheless, our method will not be optimal in all cases. For example, if the access patterns change during different phases of a program, the decisions may not be optimal. Also, it may sometimes be more efficient to replicate objects on part of the processors, whereas our current scheme uses either full or no replication. We intend to look at these issues in the future.

In conclusion, we have shown that good decisions about the mapping of data objects in Orca can be made by integrating compile-time and run-time techniques. In addition, we think that our approach may also prove to be suitable for other architectures than LAN-based systems and for implementing additional optimizations for Orca.

REFERENCES

1. H.E. Bal, *Programming Distributed Systems*, Prentice Hall Int'l, Hemel Hempstead, UK (1991).
2. 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).
3. A.S. Tanenbaum, M.F. Kaashoek, and H.E. Bal, ''Parallel Programming using Shared Objects and Broadcasting,'' *IEEE Computer* **25**(8), pp. 10-19 (Aug. 1992).
4. H.E. Bal, M.F. Kaashoek, A.S. Tanenbaum, and J. Jansen, ''Replication Techniques for Speeding up Parallel Applications on Distributed Systems,'' *Concurrency Practice & Experience* **4**(5), pp. 337-355 (Aug. 1992).
5. G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C-W. Tseng, and M-Y. Wu, ''FORTRAN-D Language Specification,'' TR90-141, Rice University (Dec. 1990).
6. C. Koelbel, P. Mehrota, and J. van Rosendale, ''Supporting Shared Data Structures on Distributed Memory Architectures,'' *Proc. 2nd Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, pp. 177-186 (March 1990).
7. M. Rosing, R. Schnabel, and R. Weaver, ''The DINO parallel programming language,'' *Journal of Parallel and Distr. Computing* **13**(1), pp. 30-42 (Sept. 1991).

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

9. S. Ahuja, N. Carriero, and D. Gelernter, ‘‘Linda and Friends,’’ *IEEE Computer* **19**(8), pp. 26-34 (Aug. 1986).

10. J.K. Bennett, J.B. Carter, and W. Zwaenepoel, ‘‘Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence,’’ *Proc. 2nd Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, pp. 168-176 (March 1990).

11. E. Jul, H. Levy, N. Hutchinson, and A. Black, ‘‘Fine-Grained Mobility in the Emerald System,’’ *ACM Trans. Comp. Syst.* **6**(1), pp. 109-133 (Feb. 1988).

12. J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Littlefield, ‘‘The Amber System: Parallel Programming on a Network of Multiprocessors,’’ *Proc. of the 12th ACM Symp. on Operating System Principles*, Litchfield Park, AZ, pp. 147-158 (Dec. 1989).

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

14. D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, ‘‘Directory-Based Cache Coherence in Large-Scale Multiprocessors,’’ *IEEE Computer* (June 1990).

15. L. Lamport, ‘‘How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs,’’ *IEEE Trans. on Computers* **C-28**(9), pp. 690-691 (Sept. 1979).

16. L. Uljee and H.-J. Visscher, ‘‘C++/Orca,’’ Master’s thesis, Vrije Universiteit, Amsterdam (Sept. 1992).

17. A.S. Tanenbaum, *Modern Operating Systems*, Prentice-Hall, Englewood Cliffs, NJ (1992).

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

19. H.T. Kung, ‘‘Gigabit Local Area Networks: a Systems Perspective,’’ *IEEE Communications Magazine* **30**(4), pp. 79-89 (April 1992).

20. E. Biagioni, E. Cooper, and R. Sansom, ‘‘Designing a Practical ATM LAN,’’ *IEEE Network* **7**(2), pp. 32-39 (March 1993).

21. T.E. Jeremiassen and S.J. Eggers, ‘‘Computing Per-Process Summary Side-Effect Information,’’ *Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, pp. 115-122 (Aug. 1992).

22. J.-F. Jenq and S. Sahni, ‘‘All Pairs Shortest Paths on a Hypercube Multiprocessor,’’ *Proc. 1987 Int. Conf. Parallel Processing*, St. Charles, IL, pp. 713-716 (Aug. 1987).

23. T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser, ‘‘Active Messages: a Mechanism for Integrated Communication and Computation,’’ *Proc. 19th Int. Symp. on Computer Architecture*, Gold Coast, Australia, pp. 256-266 (May 1992).

24. A.H. Karp, ‘‘Programming for Parallelism,’’ *IEEE Computer* **20**(5), pp. 43-57 (May 1987).

25. B. Chapman, P. Mehrota, and H. Zima, ‘‘User Defined Mappings in Vienna FORTRAN,’’ *ACM SIGPLAN Notices (Proc. Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors)* **28**(1), pp. 72-75 (Jan. 1993).

26. S.K.S. Gupta, S.D. Kaushik, C.-H. Huang, J.R. Johnson, R.W. Johnson, and P. Sadayappan, ‘‘A Methodology for Generating Data Distributions to Optimize Communication,’’ *Proc. 4th IEEE Symp. on Parallel and Distributed Processing*, pp. 436-441 (Dec. 1992).

27. M. Gupta and P. Banerjee, ‘‘Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers,’’ *IEEE Trans. on Parallel and Distributed Systems* **3**(2), pp. 179-193 (March 1992).

28. E.M. Paalvast, A.J. van Gemund, and H.J. Sips, ‘‘A Method for Parallel Program Generation with an Application to the Booster Language,’’ *Proc. 1990 ACM Int. Conf. on Supercomputing*, Amsterdam (June 1990).

29. M.W. Hall, S. Hirananani, K. Kennedy, and C. Tseng, ‘‘Interprocedural Compilation of FORTRAN-D for MIMD Distributed-Memory Machines,’’ *Proc. Supercomputing’92*, Minneapolis, pp. 522-534 (Nov. 1992).

30. A. Rogers and K. Pingali, ‘‘Process Decomposition Through Locality of Reference,’’ *ACM SIGPLAN Notices (Proc. SIGPLAN’89 Conf. on Progr. Lang. Design & Impl.)*, Portland, Oregon **24**(7), pp. 69-80 (July 1989).

31. M. Chen, Y. Choo, and J. Li, ‘‘Compiling parallel programs by optimizing performance,’’ *Journal of Supercomputing* **1**(2), pp. 171-207 (July 1988).

32. B.K. Totty and D.A. Reed, ‘‘Dynamic Object Management for Distributed Data Structures,’’ *Proc. Supercomputing’92*, Minneapolis, pp. 692-701 (Nov. 1992).

33. N. Carriero, ‘‘The Implementation of Tuple Space Machines,’’ Research Report 567 (Ph.D. dissertation), Yale University, New Haven, CT (Dec. 1987).

34. P. America, ‘‘POOL-T: A Parallel Object-Oriented Language,’’ pp. 199-220 in *Object-Oriented Concurrent Programming*, ed. A. Yonezawa and M. Tokoro, M.I.T. Press, Cambridge, MA (1987).

35. B.N. Bershad and M.J. Zekauskas, ‘‘Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors,’’ CMU-CS-91-170, CMU (Sept. 1991).

36. D. Mosberger, ‘‘Memory Consistency Models,’’ *ACM Operating Systems Reviews* **28**(1), pp. 18-26 (Jan. 1993).

37. M.J. Freeley and H.M. Levy, ‘‘Distributed Shared Memory with Versioned Objects,’’ *Proc. Conf. Object-Oriented Programming Systems, Languages and Applications*, pp. 247-262 (1992).