Parallel Algorithms and Applications, Vol. 12, pp. 205-230
Reprints available directly from the publisher
Photocopying permitted by license only

HAWK: A RUNTIME SYSTEM FOR PARTITIONED OBJECTS*

SANIYA BEN HASSEN, HENRI E. BAL and ANDREW S. TANENBAUM

Vrije Universiteit, De Boelelaan 1081a, Amsterdam, The Netherlands

(Received 27 August 1996)

Hawk is a language-independent runtime system for writing data-parallel programs using partitioned objects. A partitioned object is a multidimensional array of elements that can be partitioned and distributed by the programmer. The Hawk runtime system uses the user-defined partitioning of objects and a runtime mechanism based on Partition Dependency Graphs *(PDGs)* to increase the granularity of data transfers and consistency checks to a partition. Hawk further optimizes the execution of parallel operations by prefetching data and overlapping communication and computation.

We first present the partitioned object model. Then, we give an overview of Hawk and describe how it uses PDGs to reduce communication overhead and optimize parallel operations. Finally, we discuss the effectiveness of our optimization technique with two applications written on top of Hawk.

Keywords: Data Parallelism; Partitioned Objects; Data Transfers; Performance

Classification Categories: 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.*

1 INTRODUCTION

Distributed and parallel computers have created new challenges for programmers and systems developers. Due to the difficulty involved in optimizing data distribution and synchronizing processes over multiple machines, writing efficient parallel programs is often difficult and time consuming.

* This work was supported in part by the CEC Human Capital Mobility program under contract no. ERBL-HBGCT920001 and by a PIONIER grant from the Netherlands Organization for Scientific Research (N.W.O.).

Parallel programming systems for distributed memory machines can be classified in three categories: message-passing systems, page-based Distributed Shared Memory (DSM) systems, and object-based DSM systems.

Message passing programming systems such as MPI [6] generally offer a set of low-level primitives with which processes executing in different address spaces exchange information using messages. These systems offer the most flexibility and can achieve good performance. Programmers, however, find them difficult to use since they are responsible themselves for placing, moving, and maintaining consistency of data.

In DSM systems, conceptually, shared data are located in an address space common to all processors. In page-based DSM [1,5,15], the unit of sharing between processes is a page. Physically, the pages are distributed over multiple processors. The system is responsible for placing them and for keeping them consistent. Page-based systems take advantage of hardware memory management to control access to shared data efficiently. On the other hand, software must carefully map data onto pages. This task turns out to be difficult to do well [1]. For example, a trivial mapping of data onto shared pages can result in false sharing if two unrelated data elements are accidentally stored on the same page, and used by different processors. The page will be transferred back and forth between the two processors on every access.

In object-based DSM [3,17,18,8], processes share objects. A shared object encapsulates shared data structures and the operations on the data, so it is essentially an instance of an Abstract Data Type (ADT). All processes may access a shared object, even if they are on different processors in a distributed memory system. Processes communicate by applying ADT operations to the shared objects.

An example of object-based DSM is Orca[1] [2,3]. Using the Orca model, a program forks processes on several machines. These processes communicate and synchronize through user-defined shared objects. The Orca runtime system (RTS) is responsible for the placement of objects and keeps them consistent when concurrent operations (methods) are called.

Writing parallel applications using the Orca model is often easy because of its clean semantics. Furthermore, the system does not rely on hardware mechanisms to implement shared memory consistency and thus is more portable than page-based systems. There are, however, two drawbacks to the Orca model. First, it was primarily designed for process parallelism and does not allow implementing partitioning or partial replication easily. For example, in order to partition a matrix row-wise, the programmer must define a shared

[1] URL http://www.cs.vu.nl/orca.

object for each row. The partitioning is hand-coded, and therefore not flexible. Data-parallel constructs must also be hand-coded by the programmer. Second, in the Orca system, a shared object is either placed on one processor or replicated on all processors invoking the object. For many data-parallel applications, placing objects on one processor creates a bottleneck and replicating them over all processors generates unnecessary data transfers and traffic on the network.

We have extended the shared object model of Orca with *partitioned objects* to support data-parallelism [10]. A partitioned object is a multidimensional array of elements that can be partitioned by the *programmer* using simple directives to reflect data access locality within operations. The partitions may then be distributed arbitrarily over multiple processors. The elements of a partitioned object can be accessed through operations that are either sequential or parallel.

This paper describes Hawk, a runtime system for partitioned objects. The system provides a set of primitives to create, partition, distribute, and invoke objects. The execution model of a parallel operation uses the *owner-computes* rule [11]: a processor executes parallel operations on the partitions it owns. As in the original Orca model, Hawk handles communication and synchronization transparently to the user and guarantees sequential consistency. Therefore, during the execution of a parallel operation, every time a processor accesses data it does not own, the system first checks whether local copies of the data are available. If there are not, the system fetches them from their owner. Consequently, transferring data between processors incurs overhead due to consistency checking and communication latency. Unlike other programming systems, Hawk reduces this overhead by using the partitioning of objects specified by the user: data transfers, computation partitioning, and consistency checks are done on a per-partition basis. This not only reduces the communication and consistency checking overhead but it also increases the potential for data prefetching and for overlapping communication and computation. In message-passing systems, the potential for overlap of communication and computation is the highest since data transfers are customized to the characteristics of applications. However, this optimization is left to the user to handle which makes writing parallel applications significantly more difficult. Page-based systems use hardware memory management to decrease consistency checking overhead but do not automatically overlap communication and computation. Optimization of the execution of parallel statements must be handled entirely by the user.

The remainder of the paper is organized as follows. Section 2 presents our partitioned object model. Section 3 gives an overview of a prototype of Hawk implemented on top of Amoeba[2] [24]. Section 4 describes how Hawk executes parallel operations to reduce communication and consistency checking overhead. Section 5 discusses two applications, Successive Over-Relaxation (SOR) and Fast Fourier Transform (FFT), written on top of Hawk and presents some performance results. Section 6 discusses related work. Finally, Section 7 provides conclusions and directions for further research.

2 THE PARTITIONED OBJECT MODEL

As in the Orca model, a partitioned object is an instance of an abstract data type that encapsulates state and operations. The state of a partitioned object is composed of elements that are grouped into partitions (see Figure 1). The partitions are then distributed arbitrarily over multiple processors. In this section, we describe the state of partitioned objects and their operations. Then, we discuss partitioning and distribution.

2.1 Object State

A partitioned object encapsulates a data structure as a multidimensional array of elements. An element (as defined by the model) is not itself an object. There are no operations defined on individual elements. An element can be accessed only through an operation defined at the object level. Within an operation, an element is addressed by indices.

To illustrate the model, consider the example of an iterative algorithm such as Successive Over-Relaxation (SOR) with red and black partitioning [23]. A partitioned object that implements SOR is shown in Figure 2 in an

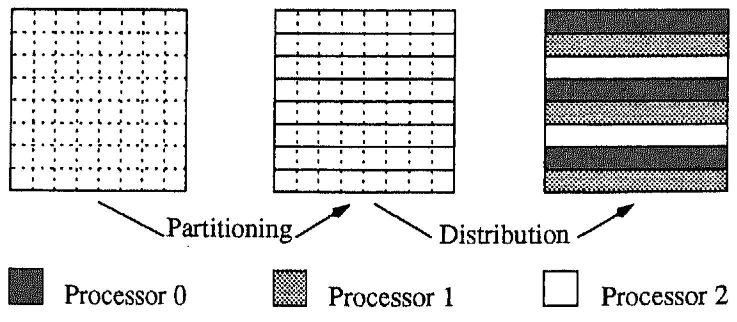


FIGURE 1 Example of a partitioned object. The object is a two-dimensional grid of elements. It is partitioned row-wise and each row is allocated to one of three processors.

[2] URL http://www.am.cs.vu.nl.

```
OBJECT IMPLEMENTATION grid [1..N : integer, 1..M : integer];
    G: real;

OPERATION PrintGrid ();
    BEGIN
       FOR row IN 1..N DO
             FOR col IN 1..M DO write (G[row, col], " "); OD;
             WriteLine;
       OD;
    END;

PARALLEL OPERATION [row, col] UpdateGrid (c : color):
             REDUCE real WITH max;
       avg, diff: real;
    BEGIN
       diff := 0;
       IF iscolor (row, col) = c AND row>1 AND col>1
                                  AND row<N AND col<M THEN
          avg := (G [row – 1, col] + G[row+1, col]
                +G[row, col – 1] + G[row, col+1])/4;
          diff := ABS (avg – G[row, col]);
          G[row, col] := G[row, col] + OMEGA * (avg – G[row, col]);
       FI;
       RETURN diff;
    END;
END;
```

FIGURE 2 Implementation of a Grid Object.

Orca-like program. The **grid** object is a two-dimensional array of N rows and M columns. Each element consists of one floating-point number.

2.2 Operations on Partitioned Objects

Operations on a partitioned object are atomic and the system guarantees that the execution of concurrent operations is equivalent to some serial execution of these operations. Therefore, objects are sequentially consistent. Operations are applied sequentially or in parallel to the elements of the object.

Sequential operations. As an example, the **grid** object includes one such operation, **PrintGrid()** (see Figure 2), that prints the contents of the grid. It accesses each element sequentially to write its value.

Parallel operations. A parallel operation is applied to all elements in parallel. The elements are referenced within an operation by means of *operation variables.* The pseudo-code for the parallel operation **UpdateGrid()** that implements the update of red or black cells in **SOR** is shown in Figure 2. The ranges of the operation variables **row** and **col** are the intervals **[1..N]** and **[1..M]**, respectively, the same intervals as the ones specified in the grid declaration. They are used within the body of the parallel operation to index the partitioned data structure **G**, which is accessed like a two-dimensional array.

Conceptually, there is one thread of control per element, in which only that element can be updated. For example, in one thread of control of the **UpdateGrid()** operation, only **G[row,col]** is updated. Arbitrary cells may be read; their values are the ones before the execution of the parallel operation started. In **UpdateGrid()**, the difference between **avg** and the cell value is computed in **diff** and returned, filtered through a reduction function, **max()**, specified in the header of the operation. This function takes two floating-point numbers as arguments and returns their maximum value. The reduction process combines the values returned for all cells. Thus, the maximum value of **diff** for all cells will be returned to the caller. Several predefined reduction functions, such as **min()** and **sum()**, are provided by the system. They can also be defined by the programmer. These functions must be associative and commutative so they can be applied to elements and partial results in a nondeterministic order. The reduction process can be implemented efficiently by taking into account the underlying network architecture.

An alternative solution is to return one floating-point number for each element of the grid. The caller is returned an array of values, and is responsible for combining them, if needed.

2.3 Partitioning and Distribution

A suitable partitioning of each object helps the system reduce the communication overhead due to data transfer between processors: rather than transferring the elements of an object individually, the RTS transfers data at partition granularity. The partitioning must reflect data access locality within invocations. So far, in the definition of the **grid** object, we did not mention how the data structure is distributed, because partitioning and distribution are orthogonal to the implementation of an object. As a result, if the partitioning changes either during the same run or from one run of an application to another, the operations on the object need not be changed. The partitioning and distribution of an object is specified by the programmer at the time the object is instantiated. It can be modified dynamically, during program execution.

Let us consider the example of SOR again. In the operation **UpdateGrid()**, the update of an element might require the value of a remote cell. Transferring the cells individually would be too costly. Therefore, the grid is partitioned into sets of rows (or columns) because entire rows can then be transferred from one processor to another. In an operation, the first time an element of a remote partition is accessed, the entire partition is fetched by the system. Access to another element of that same partition does not incur any additional communication or access overhead. How this is implemented in Hawk is discussed in Sections 3 and 4.

In this implementation of SOR, during each iteration, all the elements of the rows are transferred between processors. In a message passing version, the program can be written to transfer only red or black cells of the rows, thus half the communication volume. In general, the message passing implementation of an application would be less costly in terms of communication volume. However, it would also require the programmer to use low-level, thus error-prone, communication primitives whereas our model provides high-level partitioning and distribution directives. Most DSM systems attempt to minimize the communication volume during data transfer. It is not clear, however, that the cost of automatic marshaling/unmarshaling and the size of the resulting messages is minimal in such systems: each element must be sent along with an identification (e.g., its indices),.which requires additional space. In contrast, with our approach, only partitions must be identified within a message.

Several predefined directives allow the programmer to partition an object row-wise, column-wise, or block-wise and to distribute adjacent partitions to the same processor or cyclically. A programmer can also specify arbitrary partitioning and distribution schemes provided that each partition is an array structure. A processor owns all partitions it is allocated. The distribution of partitions can be changed at run-time (atomically) for example to balance the workload.

The pseudo-code in Figure 3 shows how the **grid** object is partitioned and invoked to compute a stable state of the grid. The grid is partitioned row-wise and distributed in blocks.

With this object model, the programmer has some control over data and workload distribution without having to worry about keeping data consistent. Often, the programmer can find an efficient partitioning by visualizing and analyzing data structures and access patterns. In principle, the partitioning and

```
#Create a partitioned object
SORgrid: grid [1..50, 1..100];

SORgrid$$partition (ROWWISE); # Row-wise partitioning
SORgrid$$distribute (BLOCK); # Block distribution
SORgrid$init (); # Initialize the grid data
REPEAT
    # Invoke data parallel operation for each phase
    maxdiff := SORgrid$UpdateGrid (red);
    maxdiff := max (maxdiff, SORgrid$UpdateGrid (black));
UNTIL maxdiff < epsilon;
SORgrid$PrintGrid ();
```

FIGURE 3 A data-parallel Successive Over-Relaxation program using the grid object.

distribution directives could also be generated by a compiler. In this case, the user would only have to define the data structure encapsulated by the object and its operations.

3 THE HAWK RUNTIME SYSTEM

Hawk is an RTS that implements the partitioned object model. Using Hawk's primitives, processes create, partition, and distribute objects over several processors. The processes may call parallel and sequential operations on objects concurrently: the RTS is responsible for keeping objects consistent and transfers data between processors transparently to the user whenever necessary. In this section, we first give an overview of the RTS. Then we describe fragments and Partition Dependency Graphs. Finally, we discuss the most important aspects of object creation and invocation.

3.1 Overview

An application program consists of processes, implemented by threads, and partitioned objects, implemented by one fragment for each object on each processor. To each fragment corresponds an invocation thread connected to other invocation threads and to client processes by communication channels. The channels are used to send and receive invocation messages and partitions over the network.

When a process calls an operation, it sends a message to one or more invocation threads of the object invoked. The thread or threads carry out the operation. They use its Partition Dependency Graph (PDG) to determine if there are dependencies between the partitions the operation accesses and validates their local copies as needed. PDGs will be discussed further in this section.

An operation is implemented by an *operation code.* For sequential operations, it is just the body of the operation and is called once by an invocation thread. For a parallel operation, the operation code executes the body of the operation for *every element of a partition.* For example, a simplified version of the operation code for **UpdateGrid()** is shown in Figure 4. The body of **UpdateGrid()** is applied to every element of a partition **part_num**, which is an argument to the call. An invocation thread must call the operation code for each partition owned by its processor.

Hawk's user interface contains four categories of primitives: *PDG construction, class creation, object instantiation,* and *object invocation.* We discuss some of the primitives in detail after describing fragments.

```
void UpdateGridOpCode (object_p object,
                             int part_num, void **args) {
    double avg, diff;
    /* Initialize return values. */
    . . .
    for every element [row, col] in part_num {
        if ((iscolor (row, col) = = c) && (row>1) && (col>1)
            && (row<N) && (col<M) {
                avg= (object→ state [row − 1, col]
                        + object → state [row+1, col]
                        + object → state [row, col−1]
                        + object → state [row, col+1])/4;
                diff = abs (avg − object → state [row, col]);
                object → temp [row, col] =
                    object → temp [row, col] +
                    OMEGA * (avg − object → temp [row, col]);
        }
    . . .
}
```

FIGURE 4 Operation code for the operation UpdateGrid ().

3.2 Fragments

The information on the class and the set of partitions of an object handled by a processor is called a *fragment*, just as a copy of a replicated object is called a replica. A fragment consists of descriptors for an object and the operations that can be applied to it (see Figure 5). Each descriptor is given a system-wide unique identifier and is replicated on all processors.

A *class descriptor* contains the shape of the partitioned object, i.e., the number of dimensions and the size of the elements, and an operation descriptor for each one of its operations.

To guarantee object and operation semantics, Hawk determines how to execute the operation depending on its type (whether the operation is sequential or parallel and whether it updates the state of the object). The operation type is stored in the *operation descriptor* along with a description of the arguments, a pointer to the operation code, and a pointer to a PDG constructor.

An *object descriptor* contains the length of each dimension of the object instantiated, a pointer to its state, a description of its partitioning and distribution, and optionally a PDG for each operation defined on the object. The description of the partitioning includes the length of the partitions along each dimension. For example, if a partitioned object of N×M elements is partitioned row-wise, the partitions have length 1 along the first dimension and M along the second dimension. The description of the distribution includes the owner of each partition.

When a client process modifies the partitioning and distribution of an object, it sends a group message to all processors. Upon reception of the message, each

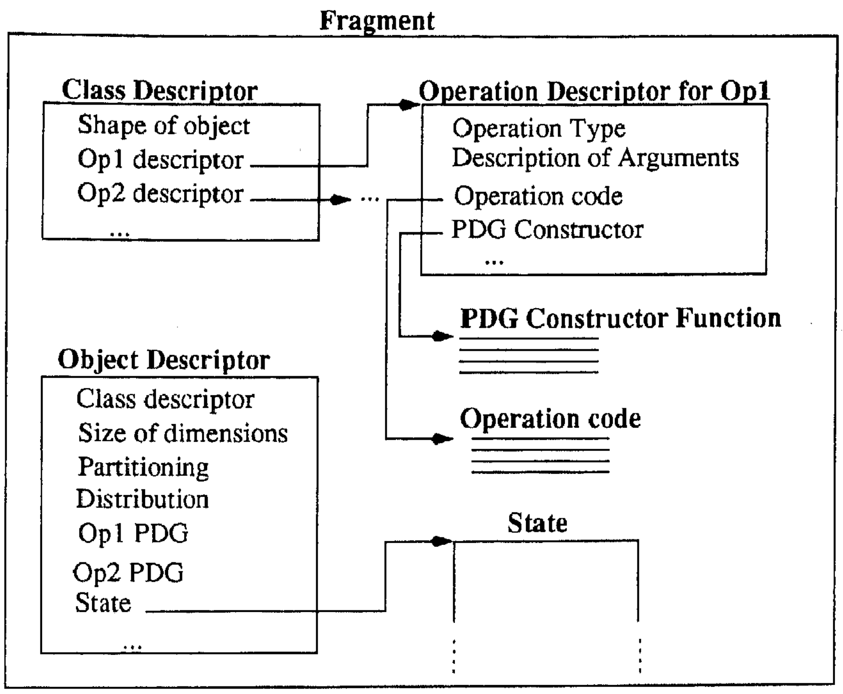


FIGURE 5 Hawk keeps track of classes, objects, and operations in the descriptors shown above.

processor updates its fragment. These messages and invocation messages are totally ordered [12]. Therefore, the update of the partitioning and distribution information is atomic and the fragments remain consistent on all processors.

3.3 Partition Dependency Graphs

During the execution of a parallel operation, on each processor, Hawk makes sure that all partitions accessed by a processor are consistent. If they are not, the RTS transfers copies from their respective owners. Both the consistency checks and the communication overhead may lead to significant performance drops. To reduce this overhead, Hawk uses the PDGs of operations. Below we describe PDGs and their constructors. How they are exploited by Hawk is described in Section 4.

A PDG G_{op} is associated with an operation op() of an object. It specifies the dependencies between *partitions* during the execution of op(). The nodes of the PDG are labeled with partition numbers. An edge (or *dependency*) (p_s, p_d) from a source node p_s to a destination node p_d specifies that, in op(), some element of p_s depends on some element of p_d, i.e., the execution of op() on some element of p_s requires access to some element in p_d. A dependency (p_s, p_d) is *local* if p_s and p_d belong to the same processor. Otherwise, it is a *remote* dependency. Before executing op() on a partition p_s, Hawk *resolves* each remote dependency of p_s in the graph: for example, for each remote dependency (p_s, p_d), it checks whether the local copy of p_d is valid; if the copy is not valid, the system fetches it from its owner.

Figure 6 shows the PDG for **UpdateGrid()** when the grid is partitioned row-wise and distributed in blocks over three processors. The update of each element, except for boundary ones, requires accessing neighboring elements, which belong either to the same row or to a neighboring row. Therefore, in the graph, there is an edge from each partition to its neighboring partitions.

Generating a PDG for an operation requires that the corresponding object has been partitioned, because the graph defines dependencies between partitions. To decide whether a dependency is local or remote, generating the graph also requires knowing where each partition is located. Since partitioning and distribution occur at runtime and may change during the execution of the same application, the user (or compiler) does not provide a graph for an operation, but rather, provides a *PDG constructor* that is used by the runtime system to build the graph once the object has been instantiated, partitioned, and distributed. In addition to being independent of the object's partitioning and distribution, the PDG constructor must be independent of the size of each dimension so it can be used for all instances of the same class.

In general, a PDG is constructed as follows. Let p_i denote the partition an element e_i belongs to. Let G_{op} be a PDG for op(). Initially, there are no edges

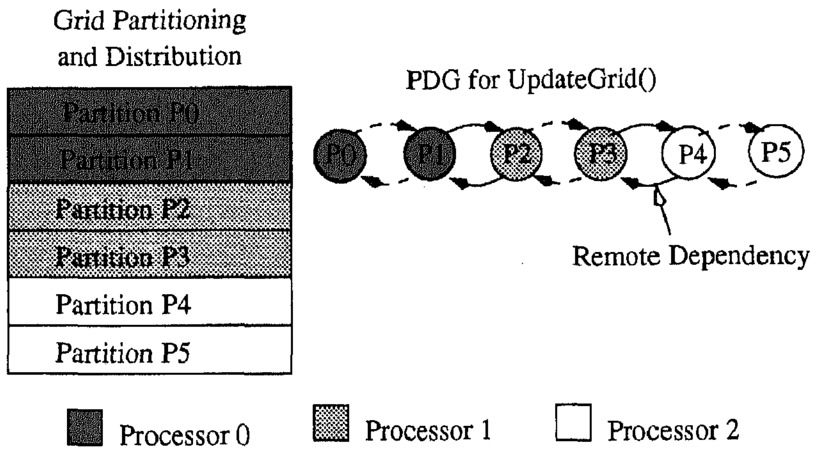


FIGURE 6 The PDG for UpdateGrid (). The grid is partitioned row-wise. Each row depends on the neighboring rows, as shown by the PDG. Plain edges represent remote dependencies.

in G_{op}. For every element e_s of the object, if in operation op(), e_s depe on element e_d, then an edge (p_s, p_d) is added to G_{op}. If p_s and p_d belon different processors, the dependency is marked as being remote. Adding same edge to the graph more than once has no effect. There is no edge fı a node to itself.

Hawk provides several primitives for writing PDG constructors to cre a PDG, add or remove dependencies in a PDG, and determine the partit number of an element. A PDG constructor uses the dependencies betw *elements* to build the dependencies between *partitions*: a loop goes over e element of the object and determines the partition **source** it belongs to. each index expression used within the operation, the routine determines corresponding partition **dest**. Then, it adds an edge from **source** to **dest** to graph, provided that **dest** is a valid partition. The RTS marks the depende as being local or remote according to the current distribution of the obj This constructor is independent of the size, partitioning, and distribution the object. We could also write a PDG constructor that is customized t particular partitioning and distribution of the grid. Such a constructor wo be more efficient but also error-prone and less flexible since the partition and distribution may not be changed.

The PDG constructor for an operation of an object is given with the desc tion of the operation through a call to **new_operation**(). The constructo generated by a compiler or written by the user and Hawk executes it a the object is partitioned and distributed. If no PDG constructor is provid by default, the system makes a conservative assumption which may not optimal: it assumes that there are dependencies between all partitions in object. When dependencies are resolved for the operation, all partitions replicated on all processors using collective communication.

The overhead generated by the PDG construction for an operation depe on how often the operation is called. If the operation is called many tim the graph construction time is negligible compared to the execution ti of the operation. For example, for SOR, we measured an overhead rang from 1% for a 400 × 400 matrix (470 iterations) to 3% for a 100 × 1 matrix (192 iterations). For other operations, the overhead might be m substantial.

3.4 Class Creation

To create a class, all processors call the primitive **new_po**() and give t number of dimensions of the object and a description of each operati (basically, all the information stored in an operation descriptor). Using t information, Hawk builds the class and operation descriptors on all processors and gives them a system-wide unique identifier.

3.5 Object Creation

Given a class identifier, a client process calls **new_instance()** to instantiate an object of that class and specifies the actual size of each dimension of the object. Hawk creates the corresponding object descriptor on each processor and returns a handle to the creator of the object. This handle may then be used to partition, distribute, and invoke the object.

Once the object has been instantiated, a process partitions the state of the object either by giving the length of each dimension of a partition or by using a predefined directive. Then, the process distributes the state of the object by mapping each partition onto one processor, which becomes the owner of the partition. The owner of a partition retains the master copy of it. Other copies are secondary copies. After Hawk has stored partitioning and distribution information in the object descriptor, it executes the PDG constructors, if any, and puts the corresponding PDGs in the object descriptor. Hawk finally creates point-to-point and collective communication channels to transfer partitions between processors.

The object may not be invoked before all the steps described above are completed. Partitioning and distribution may be changed between invocations. If the partitioning is changed, the object may not be invoked before a new distribution is defined. Hawk executes the PDG constructors every time a new distribution scheme is defined.

The calls to the RTS primitives for object instantiation, partitioning, and distribution are broadcast to all processors using a totally ordered multicast protocol [12]. In this protocol, when two messages are broadcast concurrently to a group of processes, all of them receive these messages in the same order. Therefore, instantiation, partitioning and distribution are atomic and performed in the same order on all processors.

3.6 Operation Execution

Each sequential or parallel operation is classified as a *read* or a *write* operation. A *read* operation does not update the state of the object; a *write* operation does. If a remote partition is accessed during the execution of the operation, the system makes sure the local copy is consistent. If not, it fetches it from the owner. During a *write* operation, each processor creates a temporary copy of the partitions it is updating. All write accesses to owned partitions are performed on the temporary copies. If a processor requests the copy of a partition, the original copy is handed out. The original is the value of the partition before the invocation started and is guaranteed to be consistent. All write accesses to non-owned partitions (e.g., in a sequential operation) are performed on the state of the object.

A *read* sequential operation is executed on the processor where the invocation was issued (Figure 7(a)). If the local copies of the partitions accessed by the operation are not consistent, they are fetched from their owners before the execution of the operation code.

A *write* sequential operation is broadcast using total ordering of messages (Figure 7(b)). Each processor executes the operation code once and may update the entire state of the object. When done, the processors reach a barrier and commit the update of the object. Results are returned from the local execution.

A parallel operation is also broadcast to all processors (Figure 7(c)). If the operation is a *write* operation, temporary copies are created for each partition owned locally. The processors execute the operation code on the elements of their partitions, reach a barrier, and then commit the updates. Return values are combined during the barrier either by gathering them into a single array structure or by reducing them.

This implementation of the operations guarantees totally ordered and atomic execution of operations. Two concurrent invocations of an object are ordered by the broadcast communication protocol. The synchronization phase after each *write* sequential operation and each parallel operation guarantees the consistency of partitions.

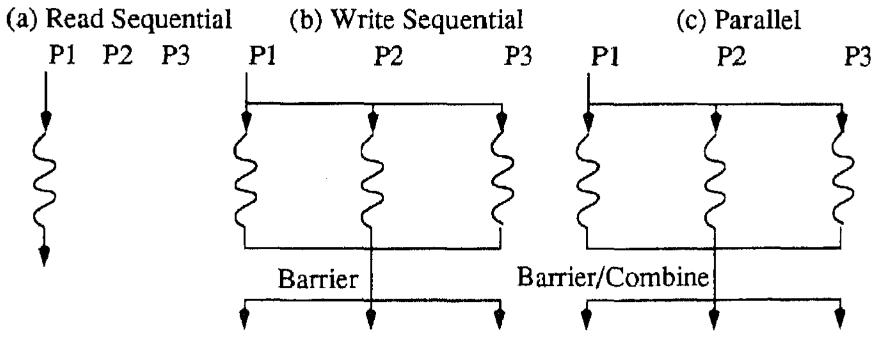


FIGURE 7 Implementation of (a) *read* sequential operations, (b) *write* sequential operations, and (c) parallel operations. There are three processors named P1, P2, and P3. A client process on P1 issues the call to the operation.

4 OPTIMIZING PARALLEL OPERATION EXECUTION

Accessing remote partitions is time consuming: a processor might spend most of its time checking whether the elements it needs to access are consistent or blocked waiting for copies of partitions to be transferred. To reduce consistency checking overhead, Hawk uses PDGs so that the consistency of a partition is checked only once rather than every time one of its elements is accessed. The latency for transferring partitions between processors during the execution of an operation also leads to a critical performance issue. To hide communication latency, Hawk prefetches data as early as possible during the execution of a parallel operation and overlaps communication and computation.

This section discusses consistency checking and communication latency hiding in more detail. First, we present the communication protocols Hawk uses to transfer partitions between processors. Second, we show how Hawk executes parallel operations in a way that reduces communication overhead. The protocols and primitives discussed in this section are not part of Hawk's user interface but are used only within the RTS to execute parallel operations on objects.

4.1 Protocols for Data Transfer

A status field is associated with each partition that tells whether the partition is consistent or not. The status field can take one of three different values: *consistent, inconsistent*, or *in transfer*. The copy of a *consistent* partition can be accessed immediately. If a partition is *inconsistent*, the original copy of the partition has been updated by its owner and the local copy has been invalidated; the original copy must be fetched before the partition is accessed. If a partition is *in transfer*, the original copy of the partition has been updated by the owner; a consistent copy has been requested by the system and is on its way.

Hawk supports three reliable protocols to transfer partitions between processors: a *receiver-initiated* protocol, a *sender-initiated* protocol, and a *collective* protocol. With the receiver-initiated protocol, a processor requests a valid copy of a partition from the owner whenever it needs to access the partition and its own copy is out of date. If (based on the PDG) the owner knows which processor will need the copy, it can use the sender-initiated protocol and send a copy of the partition without waiting for a request. This strategy saves a request message and it allows the owner to broadcast a partition when several processors need it. Hawk also supports a collective protocol to replicate the entire state of an object on one or on all processors. It is used when one or more processors access a large number of partitions. All primitives fo transfer are nonblocking and must be followed by a call to a synch primitive to check whether transfers have ended.

The protocols described above are low-level protocols for partitic between processors. Hawk also uses high-level primitives to transfer using a PDG. These procedures are briefly discussed below. They during operation execution to validate and control access to data.

If **op()** is an operation of object **obj, pdg_send (obj, op, p)** send **p** to all processors that need the partition during the execution of do so, it traverses the PDG of **op()** and for each remote dependen it sends a consistent copy of **p** to the owner of **r** using the sende protocol. If all processors issue a call to **pdg_send (obj, op, p)** owned partition **p**, eventually, all remote dependencies of **r** will b on the processor that owns **r**. A call to **pdg_fetch (obj, op, p)** als dependencies for **p** but uses the réceiver-initiated protocol. For ea dependency **(p, r)** in the PDG of **op()**, if the local copy of **r** is requests a valid one from its owner. The primitive **pdg_access (o** blocks until the local copies of all **p**'s remote dependencies are val

4.2 Parallel Operation Execution

During the execution of an operation, the system applies the oper to each partition owned by a processor in turn. For each partitic the PDG of the operation to make sure all the partition's depende been resolved. Instead of performing consistency checks before acce remote *element*, the runtime system uses the PDGs to perform c checks before accessing each remote *partition*. This increases the g for managing data consistency and reduces access overhead.

Furthermore, Hawk uses PDGs to overlap communication and co In such cases, it prefetches remote dependencies early during the of an operation while executing the operation code on partitions th remote dependencies.

Hawk uses one of three prefetching strategies: **no-prefetch prefetching**, and **partial-prefetching**. Below, we describe each o Each strategy is implemented independently from the operation system allows other strategies to be defined and provided by the ones already available in Hawk are not appropriate [9].

Execution without Prefetching. In this strategy, there is no computation and communication. It is available for experimental We use it in Section 5 to measure the effectiveness of other strateg

Assume that processor *P* owns 1 partitions that have only local dependencies and r partitions that have remote dependencies. The partition numbers are stored respectively in arrays L and R. In this strategy, Hawk first executes the operation code on each L[i]. Because the dependencies of each L[i] are owned by the local processor, there is no need to check their consistency. Then, for each R[i], it fetches copies of the partitions R[i] depends on. When their local copies are validated, it executes the operation code on R[i]. The algorithm for a **no-prefetching** strategy is shown in Figure 8.

Execution with Full Prefetching. This strategy attempts to hide communication latency by overlapping communication and computation. While remote dependencies are being resolved, the system executes the operation code on partitions that have no remote dependencies. The system uses the sender-initiated protocol for partition transfers. Upon invocation of an object, a processor *P* first sends all the partitions it owns to processors that need their copies. While the transfer is being carried out, the operation code is applied to all partitions L[i] since these do not require transfer of data. Then, for each partition R[*i*], *P* waits until the partitions R[*i*] depends on are consistent and then executes the operation code on R[i]. The algorithm for this strategy is shown in Figure 9.

```
for (i = 0;  i < 1;  i++)
    /* Execute operation code on L[i]. */
    (*op) (obj, L[i], args);
for (i = 0;  i < r;  i++) {
    /* Fetch R [i]'s remote dependencies. */
    pdg_fetch (obj, op, R[i]);
    /* Wait for end of transfer. */
    pdg_access (obj, op, R[i]);
    /* Execute operation code on R[i]. */
    (*op) (obj, R[i], args);
}
```

FIGURE 8 Execution of a parallel operation op () with **no-prefetching**.

```
/* Send R[i] to processors that need it. */
for (i = 0;  i < r;  i++) pdg_send (obj, op, R[i]);
/* Send L[i] to processors that need it. */
for (i = 0;  i < 1;  i++) pdg_send (obj, op, L[i]);
for (i = 0;  i < 1;  i++)
    /* Execute operation code on L[i]. */
    (* op) (obj, L[i], args);
for (i = 0;  i < r;  i++) {
    /* Wait until R[i]'s dependencies are resolved. */
    pdg_access (obj, op, R[i]);
    /* Execute operation code on R[i]. */
    (* op) (obj, R[i], args);
}
```

FIGURE 9 Execution of a parallel operation op () with **full-prefetching**.

```
if (r >= 1) pdg_fetch (obj, op, R[1]);
for (i = 1; i < 1; i++)
      /* Execute operation code on L[i]. */
      (* op) (obj, L[i], args);
for (i = 1; i <= r; i++) {
      /* Wait until R[i]'s dependencies are resolved. */
      pdg_access (obj, op, R[i]);
      /* Prefetch dependencies for next partition. */
      if (i<r) pdg_fetch (obj, op, R[i+1]);
      /* Execute operation code on R[i]. */
      (* op) (obj, R[i], args);
}
```

FIGURE 10 Execution of a parallel operation op () with **partial-prefetching**.

Execution with Partial Prefetching. In the **full-prefetching** strategy, all partitions are sent by all processors at the start of the operation. This may yield to network contention and actually slow down the execution of the operation in communication intensive applications. For such applications, an alternative strategy resolves dependencies for one partition at a time rather than for all of them at once (see Figure 10). The RTS transfers partitions to resolve the dependencies of R[i+1] while executing the operation on R[i]. This strategy uses a receiver-initiated protocol since the receiver must control the moment at which it needs partitions to be transferred.

5 PERFORMANCE

We have implemented a prototype of Hawk on top of Amoeba [24] on a network of workstations connected by a 10 Mbps switched. Ethernet. The processor boards are SPARC classic clones running at 50 MHz with 32 MB of RAM. To increase network bandwidth, multiple Ethernet segments are connected by a cross-bar switch.

This section describes performance measurements on SOR and FFT. In both applications, objects are created, instantiated, and called using the runtime system primitives. Although (or because) on our system, both applications do not scale well to a large number of processors, the experiments show the benefits *and* the limits of prefetching.

5.1 Successive Over-Relaxation

The first experiment compares the flexibility and performance of an Orca program with those of the same program using partitioned objects. For that, we have written two versions of SOR with red/black partitioning: one in Orca (using Orca objects) and one in the extended version of Orca that supports partitioned objects. Using Orca objects, the grid is partitioned manually. Each processor contains a number of adjacent rows, that are stored in local data. Before each iteration, neighboring processes exchange boundary rows, using shared objects for communication. The version that uses partitioned objects is shown in Figures 2 and 3. We obtain similar performance behaviors with the two programs: the difference in response time is less than 1% for matrices of 100×100, 200×200, 300×300, and 400×400 elements. In the Orca version, as we explained above, row-wise partitioning of the grid is hard-wired into the program. Changing the partitioning (e.g., to port the program to another type of machine) would require many changes to the source code. In the version using partitioned object, only the partitioning directive would have to be changed and its arguments can be given as an input to the program. We conclude that, in this case, we have gained ease of programming and flexibility without loosing performance.

The second experiment illustrates the benefits of prefetching in Hawk. Figure 11 shows the response time of two implementations of SOR with partitioned objects, one that uses **full-prefetching** and one that uses **no-prefetching**. Each variant of the program was executed on a grid of four different sizes partitioned row-wise and distributed in blocks. Prefetching performs better in all experiments.[3] The relative effectiveness of full prefetching, however, varies from less than 5% to more than 30%.

There are several explanations for these large differences in performance gain. A first one is the amount of computation carried out during an operation. Even when prefetching eliminates the idle time of the invocation threads entirely, prefetching has relatively little effect if the computation time of an operation is very large.

The large differences in performance gain are also due to the fine-grained nature of the algorithm. If the computation carried out on each partition is large enough, the invocation thread is never idle, waiting for remote partitions to be transferred. If the computation part is not large enough, however, the invocation thread may become idle for a certain amount of time. In Figure 11, if the grid size is small, the computation time is short, and therefore, the gain of prefetching is small. As the grid size increases, the computation time increases and prefetching becomes more effective.

Finally, the effectiveness of prefetching is limited by network bandwidth. As the operation response time is reduced or the number of processors increases,

[3] For a large number of processors, the poor speedup is due to the high latency of interprocessor communication on the experimental platform we used.

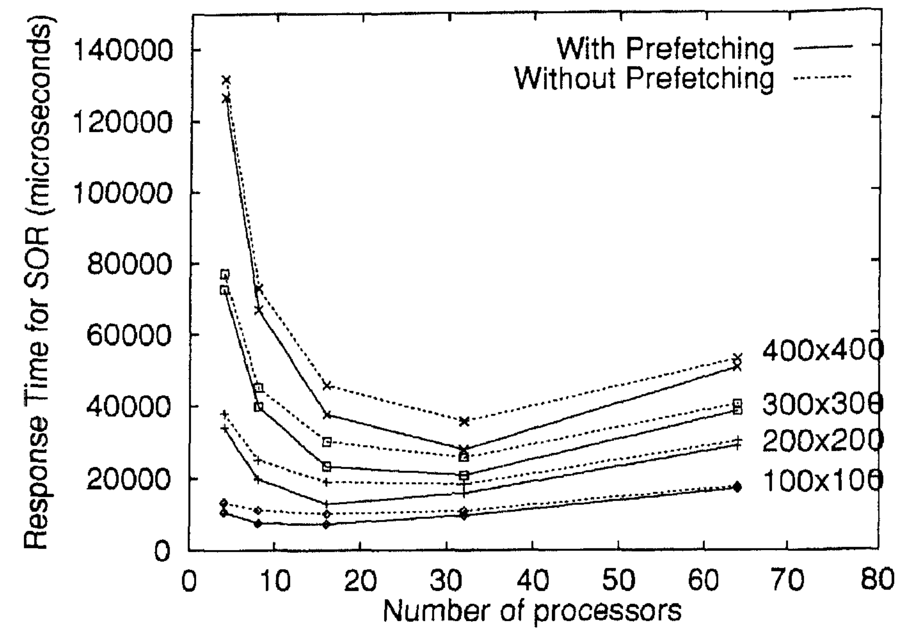


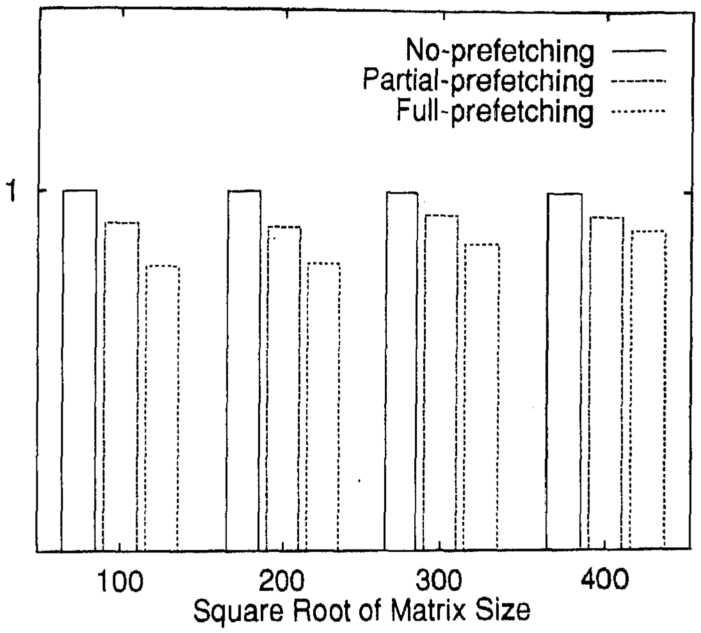
FIGURE 11 Performance of SOR executed with **full-prefetching** and with **no-prefetching**.

more partitions are transferred at a higher rate and require a higher bandwidth. If this additional bandwidth is not available, prefetching may even be counter-productive because it overloads the network. (This has been pointed out in earlier publications, e.g., see [20].) In the **full-prefetching** strategy, overloading of the network is increased because all threads initiate the transfers right after receiving the invocation message sent by the caller of the operation. Therefore, prefetching starts approximately at the same time on all processors. In this case, we can expect the network to become a bottleneck.

Figure 12 shows the performance of SOR for different matrix sizes on 8 processors using the three strategies: **no-prefetching, partial-prefetching,** and **full-prefetching**. The response times are scaled to 1 to improve the readability of the graph. In all cases, **full-prefetching** is the most efficient strategy and **no-prefetching** the least efficient one.

5.2 Fast Fourier Transform

FFT maps *n* complex values to *n* other complex values using a linear transformation. We have implemented FFT by encapsulating the complex values in a partitioned object **vector**. The transformation is applied by calling a parallel



ƎURE 12 Response time of SOR (scaled to 1) on eight processors using the three prefetching ategies.

ıeration **transform**() iteratively. Each processor holds a distinct part of the ctor and applies the transformation on that part. In **transform**(), communition patterns are dynamic. Therefore, a new PDG is generated before every ll to it.

In the operation **transform**(), all dependencies between partitions are remote ıes. Therefore, the simplest and most intuitive partitioning of the vector is define as many partitions as processors and map one partition on each ocessor. A better partitioning scheme is to allocate more than one partition ·r processor. This allows to overlap the update of one partition while the ·pendencies of others are being resolved. We executed the program five nes on 8 processors, each time dividing the vector in respectively, 8, 16, 32, ŀ, and 128 partitions. Each processor was thus allocated, respectively, 1, 2, 4, and 16 partitions. If the number of partitions held by a processor is larger, e potential for overlapping communication and computation increases. On e other hand, the overhead for managing the PDGs and the startup costs for nding partitions over the network also increase.

We executed FFT with the **no-prefetching, full-prefetching**, and **partial-·efetching** strategies on a vector of 2^{17} complex numbers on eight processors ee Figure 13). If there is no overlap of computation and communication,

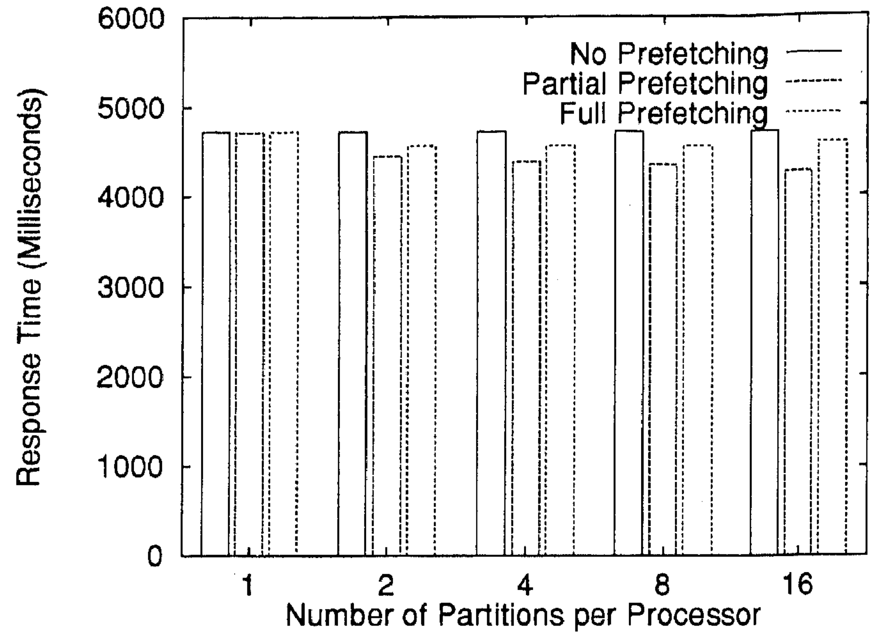


FIGURE 13 Performance of FFT on 2^{17} complex numbers using the three prefetching strategies.

a finer-grained partitioning does not increase the efficiency of the application. In Figure 13, no matter how many partitions there are, the execution with a **no-prefetching** strategy leads to the same response time. With **full-prefetching** and **partial-prefetching**, the response time decreases as the number of partitions increases, i.e., as the potential for overlap increases. The **partial-prefetching** strategy increases the execution time of FFT by up to 9.4% when compared to the execution without prefetching. Unlike in SOR, the **full-prefetching** strategy is not the most effective one because FFT has a high bandwidth requirement: for several iterations, the entire data structure is sent over the network (each processors sends its partitions to one other processor). Additionally, in this experiment, the input size is very large, thus a large amount of data is sent over the network.

6 RELATED WORK

Munin [5] and TreadMarks [1] are page-based DSM systems that implement shared memory with multiple consistency semantics on top of a loosely-coupled distributed system. The systems use memory management hardware to detect whether pages are locally consistent or not. When a variable stored on an invalidated page is accessed, a page-fault occurs and the system fetches the page on which the variable is stored (or the parts of the page that have been updated since the last transfer). Therefore, like in our runtime system, Munin and TreadMarks reduce the overhead for accessing inconsistent data. The advantage of such systems is that partitioning of objects (into pages) is done transparently to the user. On the other hand, the page size is fixed and the mapping of data onto pages is nontrivial and may be inappropriate, as illustrated in [16].

LCM (Loosely-Coherent Memory) [14] is a DSM system with a finer-grained memory access control than Munin and TreadMarks. Pages are divided into blocks. Access to an invalidated block causes a page access fault, followed by a block access fault, which in turn causes a block "reconciliation." Blocks are not reconciliated collectively. A block may not span multiple pages. Therefore, mapping objects onto pages is still a critical issue. If block faults are implemented in hardware, LCM also incurs less overhead than our runtime system which checks for the consistency of partitions using a procedure call. On the other hand, because they use hardware mechanisms, systems such as LCM, Munin, and TreadMarks are less portable than our system, which only uses software techniques. Furthermore, these systems are less flexible since they do not allow collective transfers or sender-initiated transfers of pages.

Midway [4] is also a DSM system that offers several consistency semantics for shared data. Unlike Munin, TreadMarks, and LCM, Midway is entirely software based and relies on compile-time support to manage the consistency of shared data. This software implementation avoids the problem of false sharing. In some instances, it also decreases the time spent in critical sections, when invalidated data items are accessed: in page-based DSM systems, accessing such items would generate a time consuming page-fault.

Most other optimization techniques are used in compile-time optimizers that are implemented on top of message passing platforms such as MPI [6]. Compilers for data-parallel languages, such as the FORTRAN D compiler [7] and Ptran II [11], use *message vectorization* and *message coalescing* to reduce communication latency and consistency checks. Using these techniques, a compiler detects remote data dependencies and places communication statements in the target programs to fetch or send remote data using the least possible number of messages. Some compilers carry out a data-flow analysis of programs, in addition to dependence analysis, to overlap communication and computation [13]. They pull communication statements out of loops whenever possible and use nonblocking communication to transfer data. These techniques are the most efficient when used with statically allocated distributed

structures so that the compiler knows the exact shape of the objects, t
distribution, and the number of processors that will execute the applicatic

For dynamically allocated distributed arrays and irregular communica
patterns, there exist techniques such as inspectors/executors [19,21,22]. F
for each data-parallel computation, an *inspector* analyzes access pattern
generate *communication schedules.* These schedules reduce communica
overhead by aggregating messages and eliminating redundant transfer
elements. Before carrying out the computation, an *executor* uses the comm
cation schedules to move data-elements between processors and then perfc
the computation. Inspectors/executors are similar to the PDG constructors
traversal procedures supported by Hawk. However, because the granul
of data transfer in our system is a partition (as opposed to an elem
Hawk allows more flexible execution strategies for parallel operations
less overhead. For example, it is easier to tune the overlapping of com
nication and computation by simply modifying the size of the partition
the application) or the order in which partitions are updated (in the F
(see [9] for a more detailed description of the operation execution mode
Hawk). To the best of our knowledge, reordering the execution of a par
statement on elements of an array to improve the overlap of computation
communication has not been implemented in other systems. On the c
hand, to use our RTS, compile-time optimizers or parallelizing comp
would have to automatically find suitable partitioning and distribution st
gies of arrays to exploit this potential if directives are not provided by
user. This can easily be done for applications with static access pattern
our model, when access patterns are dynamic, the programmer may spe
data-dependencies.

7 CONCLUSIONS AND FUTURE WORK

We have described partitioned objects and PDGs, a model and a runtime m
anism used to reduce the overhead for handling communication and co
tency in parallel programming systems. The partitioning of objects incre
the granularity of data consistency management. PDGs help optimizing co
tency checks and data transfers transparently to the user while hiding com
nication latency by prefetching data.

We have described Hawk, a runtime system for partitioned objects. U
two applications, we have shown that our prefetching mechanism is effec
We obtained a performance gain of 5% to 30% for SOR and over 9%
FFT. From the experiments presented in this paper, we conclude that

effectiveness of prefetching during the execution of an operation depends on three factors: the amount of computation, the amount of communication, and the bandwidth requirement. For communication intensive applications, full prefetching may not be as effective as partial prefetching if the additional bandwidth required by full prefetching is not available.

As mentioned in the introduction, our goal is to integrate the partitioned object model with the Orca shared object model to support mixed task- and data-parallelism in a single programming model and system [10]. The Orca compiler has been modified to support partitioned objects and generate PDG constructors for applications with static access patterns. In the future, we will look at how the compiler can generate efficient PDG constructors for dynamic access patterns as well.

Acknowledgments

We would like to thank Raoul Bhoedjang, Franz J. Hauck, Ceriel Jacobs, Tim Rühl, Kees Verstoep, and the anonymous reviewers for their useful comments during the preparation of this paper.

References

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

[2] Bal, H. (1989). *The Shared Data Object Model as a Paradigm For Programming Distributed Systems*, PhD thesis, Vrije Universiteit, Amsterdam.

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

[4] Bershad, B., Zekauskas, M. and Sawdon, W. (1993). The Midway distributed shared memory system, In *Proceedings of CompCon'93*.

[5] Carter, J., Bennett, J. and Zwaenepoel, W. (1991). Implementation and performance of Munin, In *13th annual ACM Symp. on Operating Systems Principles*, pages 152-164.

[6] Forum, M. P. I. (1994). MPI: A message-passing interface standard, Mar.

[7] Gupta, M., Midkiff, S., Schonberg, E., Sweeney, P., Wang, K. and Burke, M. (1993). PTRANII — a compiler for high performance Fortran, In *4th Workshop on Compilers for Parallel Computing*, Delft, The Netherlands.

[8] Haines, M., Hess, B., Mehrotra, P., van Rosendale, J. and Zima, H. (1995). Runtime support for data parallel tasks, In *Proc. Frontiers 95*, pages 432-439.

[9] Hassen, S. B., Athanasiu, I. and Bal, H. (1996). A flexible operation execution model for shared distributed objects, In *Proceeding of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '96)*, San Jose, California, Oct.

[10] Hassen, S. B. and Bal, H. (1996). Integrating task and data parallelism using shared objects, In *10th ACM Intl. Conference on Supercomputing*, Philadelphia, Pennsylvania, May. pages 317-324.

[11] Hiranandani, S., Kennedy, K. and Tseng, C. (1992). Compiling Fortran D for MIMD distributed memory machines, *CACM*, Aug. **35**(8).

[12] Kaashoek, M. (1992). *Group Communication in Distributed Computer Systems*, PhD thesis, Vrije Universiteit, Amsterdam.

[13] Kennedy, K. and Nedeljkoviċ, N. (1994). Combining dependence and data-flow analyses to optimize communication, Technical Report CRPC-TR94484-S, Computer Science Department, Rice University, Sept.

[14] Larus, J., Richards, B. and Visvanatan, G. (1994). LCM: Memory system support for parallel language implementation, In *Architectural Support for Programming Languages and Operating Systems*, Oct.

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

[16] Lu, H., Dwarkadas, S., Cox, A. and Zwaenepoel, W. (1995). Message passing versus distributed shared memory on networks of workstations, In *Proc. of Supercomputing '95*, Dec.

[17] Maffeis, S. (1993). ELECTRA — making distributed programs object-oriented, In *Usenix Symp. on Experiences with Distributed and Multiprocessor Systems*, San Diego, California, Sept. pages 143-156.

[18] Pardyak, P. (1992). Group communication in an object-based environment, In *2nd Intl. Workshop on Object Orientation in Operating Systems*, Dourdon, France, Sept. pages 106-116.

[19] Ponnusamy, R., Huang, Y., Das, R., Saltz, J., Choudhary, A. and Fox, G. (1995). Supporting irregular distributions using data-parallel languages, *IEEE Parallel and Distributed Technology*, pages 12-24, Spring.

[20] Saavedra, R., Mao, W. and Hwang, K. (1994). Performance and optimization of data prefetching strategies in scalable multiprocessors, *Journal of Parallel and Distributed Computing*, **22**, 427-448.

[21] Saltz, J. and Mirchandaney, R. (1991). The prepocessed doacross loop, In *Intl. Conference on Parallel Processing*, pages 174-179.

[22] Sharma, S., Ponnusamy, R., Moon, B., Hwang, Y., Das, R. and Saltz, J. (1994). Run-time and compiletime support for adaptive irregular problems, In *Supercomputing '94*, IEEE, pages 97-106.

[23] Stoer, J. and Bulirsh, R. (1983). *Introduction to Numerical Analysis.*, Springer-Verlag, New York, NY.

[24] Tanenbaum, A., van Renesse, R., van Staveren, H., Sharp, G., Mullender, S., Jansen, A. and van Rossum, G. (1990). Experiences with the Amoeba distributed operating system, *CACM*, Dec. **33**, 46-63.