Eliminating Synchronization Bottlenecks Using Adaptive Replication

MARTIN C. RINARD
Massachusetts Institute of Technology
and
PEDRO C. DINIZ
University of Southern California

This article presents a new technique, adaptive replication, for automatically eliminating synchronization bottlenecks in multithreaded programs that perform atomic operations on objects. Synchronization bottlenecks occur when multiple threads attempt to concurrently update the same object. It is often possible to eliminate synchronization bottlenecks by replicating objects. Each thread can then update its own local replica without synchronization and without interacting with other threads. When the computation needs to access the original object, it combines the replicas to produce the correct values in the original object. One potential problem is that eagerly replicating all objects may lead to performance degradation and excessive memory consumption.

Adaptive replication eliminates unnecessary replication by dynamically detecting contention at each object to find and replicate only those objects that would otherwise cause synchronization bottlenecks. We have implemented adaptive replication in the context of a parallelizing compiler for a subset of C++. Given an unannotated sequential program written in C++, the compiler automatically extracts the concurrency, determines when it is legal to apply adaptive replication, and generates parallel code that uses adaptive replication to efficiently eliminate synchronization bottlenecks.

In addition to automatic parallelization and adaptive replication, our compiler also implements a lock coarsening transformation that increases the granularity at which the computation locks objects. The advantage is a reduction in the frequency with which the computation acquires and releases locks; the potential disadvantage is the introduction of new synchronization bottlenecks caused by increases in the sizes of the critical sections. Because the adaptive replication transformation takes place at lock acquisition sites, there is a synergistic interaction between lock coarsening and adaptive replication. Lock coarsening drives down the overhead of using adaptive replication, and adaptive replication eliminates synchronization bottlenecks associated with the overaggressive use of lock coarsening.

Our experimental results show that, for our set of benchmark programs, the combination of lock coarsening and adaptive replication can eliminate synchronization bottlenecks and significantly

This research was supported in part by NSF grant CCR-9702297.
Authors' address: M. C. Rinard, MIT Laboratory for Computer Science, 545 Technology Square, NE43-620A, Cambridge, MA 02139; email: rinard@lcs.mit.edu; P. C. Diniz, USC/ISI, 4676 Admiralty Way, Suite 1001, Marina del Rey, CA 90202; email: pedro@isi.edu.

reduce the synchronization and replication overhead as compared to versions that use none or only one of the transformations.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Atomic operations, commutativity analysis, parallel computing, parallelizing compilers, replication, synchronization

1. INTRODUCTION

Problems such as nondeterministic behavior and deadlock complicate the development of multithreaded software. Programmers have responded to these problems by adopting a programming methodology in which each thread performs as a sequence of atomic operations on a unit of shared data such as a static collection of variables or an object [Hoare 1974; Brinch-Hansen 1972]. The result is a more structured, tractable programming model. The advantages of this model have led to its adoption in programming languages such as Concurrent Pascal, Modula-2+, Mesa, and Java [Brinch-Hansen 1975; Rovner 1986; Lampson and Redell 1980; Arnold and Gosling 1996].

The research presented in this article attacks a performance problem, synchronization bottlenecks, that arises in the context of multithreaded programs that perform atomic operations on objects. Synchronization bottlenecks occur when multiple threads attempt to concurrently update the same object. The mutual exclusion synchronization required to make the updates execute atomically serializes the execution of the threads performing the updates. This serialization can harm the performance by limiting the amount of concurrency available in the program. It can also lead to system-level anomalies such as lock convoys and priority inversions [Lampson and Redell 1980; Sha et al. 1990].

1.1 Adaptive Replication

In many programs, it is possible to eliminate synchronization bottlenecks by replicating frequently accessed objects that cause these bottlenecks. Each thread that updates such an object creates its own local replica and performs all updates locally on that replica with no synchronization and no interaction with other threads. When the computation needs to access the final value of the object, it combines the values stored in the replicas to generate the final value.

We have developed a program analysis algorithm (which determines when it is legal to replicate objects), a compiler transformation, and a run-time system that, together, automatically transform a program so that it replicates data to eliminate synchronization bottlenecks. A key problem that this system must solve is determining which objects to replicate. If the system eagerly replicates all objects, the resulting memory and computation overheads can degrade the performance. But as described above, failing to replicate objects that cause synchronization bottlenecks can also cause serious performance problems.

Our technique uses *adaptive replication* to determine which objects to replicate. As the automatically transformed program performs atomic operations on objects, it measures the amount of time it spends waiting to acquire exclusive

access to each object. The program uses this measurement to dynamically detect and replicate objects that would otherwise cause synchronization bottlenecks. In effect, the program dynamically adapts its replication policy so that it performs well for the specific dynamic access pattern of each execution.

1.2 Lock Coarsening and Synchronization Overhead

If the parallel threads in the computation usually update different objects, each lock is acquired immediately, and there is no synchronization bottleneck. In this case, the primary source of overhead is the synchronization overhead associated with executing the lock acquire and release operations. We have attacked this source of overhead by using *lock coarsening* to increase the granularity at which the computation locks objects [Diniz and Rinard 1998]. We have developed two kinds of lock coarsening: *computation lock coarsening* and *data lock coarsening*.

The basic idea behind computation lock coarsening is to automatically find sequences of operations that acquire and release the same lock. The compiler then transforms the computation to acquire the lock once, perform the sequence of operations without additional synchronization, then release the lock. Data lock coarsening finds groups of objects that tend to be accessed together, then transforms the computation so that all of the objects use the same lock. The computation lock coarsening transformation can then eliminate all but the first acquire and the last release from operations on objects that use the same lock. In the original computation, of course, all of the objects used different locks. The primary direct benefit of data lock coarsening is a reduction in the number of allocated locks. An important indirect benefit is that it increases the effectiveness of the computation lock coarsening transformation—data lock coarsening converts sequences of operations that acquire and release different locks to sequences that acquire and release the same lock. The primary advantage of lock coarsening in general is a decrease in the frequency with which the computation acquires and releases locks. The primary drawback is that the increases in the sizes of the critical sections may introduce new synchronization bottlenecks.

1.3 Interaction Between Adaptive Replication and Lock Coarsening

In previous research, we managed the trade-off between reducing the synchronization frequency and introducing or exacerbating synchronization bottlenecks by generating multiple versions of the code, each compiled with a different lock coarsening policy. The policies varied in how aggressively they applied the lock coarsening transformation to increase the sizes of the critical sections. The generated code then dynamically measured the overhead of each policy to choose the version with the least overhead [Diniz and Rinard 1999]. But adaptive replication can eliminate the need to navigate the trade-off between synchronization overhead and synchronization bottlenecks. There is a synergistic interaction between lock coarsening and adaptive replication. If lock coarsening introduces a synchronization bottleneck, adaptive replication can eliminate the bottleneck. If lock coarsening simply reduces the lock overhead without introducing a synchronization bottleneck, adaptive replication eliminates any generation of unnecessary replicas. And because the overhead

of detecting contention and managing replicas takes place at the lock acquire point, lock coarsening can reduce the overhead of using adaptive replication.

1.4 Concurrency Model

Our algorithm is designed to analyze structured parallel programs that execute an interleaved sequence of sequential and parallel phases. During each sequential phase, only a single thread executes. At the beginning of each parallel phase, the computation generates a set of threads, all of which execute in parallel; each parallel thread performs a sequence of atomic operations on shared objects. When all of the parallel threads terminate, execution continues on to the next sequential phase. Standard parallel constructs such as parallel loops produce programs that conform to this model.

Our analysis is interprocedural but does not analyze the whole program as a unit. It instead analyzes each parallel phase independently to find sets of objects that are, within that phase, candidates for adaptive replication. In particular, it may be possible to replicate a given set of objects in one parallel phase but not in another.

1.5 Results and Contributions

We have implemented both adaptive replication and lock coarsening in the context of a parallelizing compiler for object-based languages [Rinard and Diniz 1997]. Given a sequential program written in a subset of C++, the compiler automatically generates parallel code that uses lock coarsening and adaptive replication to efficiently eliminate synchronization bottlenecks. This article presents experimental results that characterize the impact of these two techniques on the performance of several benchmark applications. All of these programs perform computations that are of interest in the field of scientific and engineering computation. Our experimental results show that, for our set of benchmark programs, the combination of lock coarsening and adaptive replication can eliminate synchronization bottlenecks and reduce the synchronization and replication overhead as compared to the initial parallel version or versions that use only one of the transformations.

This article makes the following contributions:

—*Program Analysis*: It presents a static program analysis algorithm that determines when it is legal to replicate objects to eliminate synchronization bottlenecks.

—*Transformation*: It presents a program transformation algorithm that enables the generated code to correctly replicate objects.

—*Adaptive Replication*: It presents an adaptive replication algorithm that dynamically adapts to the access pattern of each execution of the program to choose which objects to replicate.

—*Interaction*: It identifies a synergistic interaction between lock coarsening and adaptive replication.

—*Results*: It presents experimental results that characterize the performance impact of lock coarsening and adaptive replication on three benchmark

```
class sumAccumulator {
  public: double value;
  void update(double v) {
    value += v;
  }
};

class maxAccumulator {
  public: double value;
  void update(double v) {
    if (value < v) value = v;
  }
};

class accumulator {
  public:
  sumAccumulator sum;
  maxAccumulator max;
  void update(double v) {
    sum.update(v);
    max.update(v);
  }
};

class vector {
  double elements[N];
  public:
  void compute(accumulator *a) {
    for (int i = 0; i < N; i++) {
      if (elements[i] > 0.0) {
        a->update(elements[i]);
      }
    }
  }
};
```

Fig. 1. Example serial program.

applications. These results show that the combination of lock coarsening and adaptive replication can eliminate synchronization bottlenecks and significantly reduce the synchronization and replication overhead as compared to versions that use none or only one of the transformations.

The remainder of the article is structured as follows: Section 2 presents an example that illustrates the issues associated with lock coarsening and adaptive replication. Section 3 presents the replication analysis and code generation algorithms. Section 4 presents experimental results that characterize the impact of lock coarsening and adaptive replication on the performance and memory consumption. We discuss related work in Section 5 and conclude in Section 6.

2. AN EXAMPLE

We next provide an example that illustrates the issues associated with data lock coarsening, computation lock coarsening, and adaptive replication. The `compute` operation of the `vector` class in Figure 1 sequentially scans a vector to compute the sum and the maximum of all of the elements that are greater than zero. It uses three accumulator classes: the `sumAccumulator` class accumulates the sum of the numbers passed to its `update` operation, the `maxAccumulator` class accumulates the maximum of the numbers passed to its `update` operation, and the `accumulator` class uses a sum accumulator and a maximum accumulator to accumulate both sums and maximums.

2.1 Commutativity Analysis and Automatic Parallelization

Our compiler first uses commutativity analysis to automatically parallelize the computation [Rinard and Diniz 1997]. Commutativity analysis is designed to parallelize object-based programs. Such programs structure the computation

as a set of operations on objects. Each object implements its state using a set of instance variables. An instance variable can be a nested object, a reference to an object, a primitive data item such as an `int` or a `double`, or an array of any of the preceding types. Each operation has several parameters and a distinguished object on which it operates. This distinguished object is syntactically identified at the operation invocation site by its placement before the operation name, separated from the operation name by either a period (if the distinguished object is identified by name) or the symbol → (if the distinguished object is identified using a reference to the object).

When an operation executes, it can read and write the instance variables of the object on which it operates, access the parameters, or invoke other operations. While the structure present in this model of computation significantly simplifies the application of dynamic replication to object-based programs, it is possible to generalize our approach to handle programs with different models of computation; Section 3.8 discusses this possibility in more detail.

The commutativity analysis algorithm analyzes the program at the granularity of *operations* on *objects* to determine if the operations commute, that is, if they generate the same result regardless of the order in which the operations commute. If all operations in a given computation commute, the compiler can automatically generate parallel code.

To test that two operations A and B commute, the compiler considers two execution orders: the execution order A;B in which A executes first, then B executes, and the execution order B;A in which B executes first, then A executes. The two operations commute if they meet the following commutativity testing conditions:

—*Instance Variables*: The new value of each instance variable of the objects that A and B update under the execution order A;B must be the same as the new value under the execution order B;A.

—*Invoked Operations*: The multiset of operations directly invoked by either A or B under the execution order A;B must be the same as the multiset of operations directly invoked by either A or B under the execution order B;A.

Both commutativity testing conditions are trivially satisfied if the two operations access different objects or if neither operation writes an instance variable that the other accesses—in both of these cases the operations are independent. If the operations may not be independent, the compiler uses symbolic execution and algebraic simplification to reason about the values computed in the two execution orders.

In our example, the compiler determines that all of the operations in the example commute. It can therefore parallelize the loop in the `compute` operation that scans the elements of the vector.

2.2 Generating Parallel Code

Commutativity analysis assumes that the operations in the parallel phases execute atomically. The compiler therefore augments each potentially updated object with a *mutual exclusion lock*. If an operation accesses an object that

```
class sumAccumulator {
  public:
  double value;
  lock l;
  void update(double v) {
    l.acquire();
    value += v;
    l.release();
  }
};
```

```
class maxAccumulator {
  public:
  double value;
  lock l;
  void update(double v) {
    l.acquire();
    if (value < v) value = v;
    l.release();
  }
};
```

Fig. 2. Generated parallel code for accumulator classes.

```
class vector {
  double elements[N];
  public:
  void compute(accumulator *a) {
    parallel for (int i = 0; i < N; i += B) {
      int u = i+B;
      if (N < u) u = N;
      for (int j = i; j < u; j++) {
        if (elements[i] > 0.0) {
          a->update(elements[i]);
        }
      }
    }
  }
};
```

Fig. 3. Generated parallel loop.

is updated during the parallel phase, it first acquires the object's lock, performs its accesses to the object, then releases the lock. This synchronization ensures that the operations execute atomically. In our example, the compiler augments the `sumAccumulator` and `maxAccumulator` objects with locks. It also augments the `update` operation in these classes with constructs that acquire and release the updated object's lock. Figure 2 presents these versions of the two accumulator classes.

The generated parallel loop code exposes the concurrency to a run-time system that schedules the iterations. To reduce the concurrency exploitation overhead, the scheduler dynamically allocates blocks of iterations to processors instead of scheduling the parallel computation at the granularity of individual loop iterations. Our implemented compiler uses guided self-scheduling [Polychronopoulos and Kuck 1987]. The code in Figure 3, which presents a high-level version of the generated parallel code, simply uses a fixed blocking factor `B`.

At this point, the compiler has generated a parallel program with a simple model of parallel computation. The program consists of a sequence of parallel phases and sequential phases. In our example, the parallel phase starts when execution reaches the parallel loop. The parallel phase ends when all of the iterations of the loop complete, at which point a single sequential thread of control continues executing after the loop. Each parallel phase executes a set of

```
class sumAccumulator {
  public: double value;
  void update(double v) {
    value += v;
  }
};

class maxAccumulator {
  public: double value;
  void update(double v) {
    if (value < v) value = v;
  }
};
```

```
class accumulator {
  public:
  lock l;
  sumAccumulator sum;
  maxAccumulator max;
  void update(double v) {
    l.acquire();
    sum.update(v);
    l.release();
    l.acquire();
    max.update(v);
    l.release();
  }
};
```

Fig. 4. Accumulator classes after data lock coarsening.

operations on objects, with each operation executing atomically with respect to all other operations in the parallel phase. It is important to note that the lock coarsening and adaptive replication transformations are designed to operate on *any* parallel program that conforms to this model of computation—they are conceptually independent of the specific mechanism used to obtain the parallel program.

2.3 Data Lock Coarsening

The generated code acquires two locks for each update: one for the `sumAccumulator` object and the other for the `maxAccumulator` object. Data lock coarsening lifts the locks out of these two objects, replacing them with a single new lock in the enclosing `accumulator` object. It then transforms the code to acquire this new lock whenever it updates one of the `sumAccumulator` or `maxAccumulator` objects. Figure 4 presents the accumulator classes after data lock coarsening.

Although this transformation does not reduce the number of executed synchronization operations, it does reduce the number of locks that the program allocates. It also transforms sequences of operations that acquire and release several different locks into sequences that repeatedly acquire and release the same lock. Once the sequence acquires and releases the same lock, the computation lock coarsening algorithm described below can eliminate all acquires and releases except the initial acquire and the final release. The net combined effect is to convert a sequence of operations that acquire and release several different locks into a computation that acquires one lock, performs the sequence of operations without further synchronization, then releases the lock.

The key issues in data lock coarsening are determining which objects to group together to use the same lock and ensuring the the correctness of the transformation. We use a heuristic that attempts lift locks out of nested objects into the enclosing object. To ensure the correctness of the transformation, the compiler examines the call graph for the code in the parallel phase to ensure that all accesses to the nested objects take place via operations that execute on the enclosing object. By default, the new lock acquire and release operations

```
class accumulator {
  public:
  lock l;
  sumAccumulator sum;
  maxAccumulator max;
  void update(double v) {
    sum.update(v);
    max.update(v);
  }
};
```

```
class vector {
  double elements[N];
  public:
  void compute(accumulator *a) {
    int i;
    parallel for (i = 0; i < N; i += B) {
      int u = i+B;
      if (N < u) u = N;
      a->l.acquire();
      for (int j = i; j < u; j++) {
        if (elements[i] > 0.0) {
          a->update(elements[i]);
        }
      }
      a->l.release();
    }
  }
};
```

Fig. 5. Vector and accumulator classes after computation lock coarsening.

are placed immediately before and after the calls to the operations on nested objects. The compiler also attempts to reuse existing synchronization in the enclosing object. If the enclosing object already has a lock, the compiler reuses that lock for the nested objects and does not add another lock to the enclosing object. If an operation on the enclosing object already acquires and releases the enclosing object's lock, the compiler does not insert additional synchronization around the calls to operations on nested objects.

An obvious extension to our existing lock coarsening algorithm is to group objects together based on their referencing relationships: if one object refers to several other objects, lift the lock into the first object. Other researchers have developed analyses that could be used to establish the correctness of extended data lock coarsening transformations that rely on the referencing information [Dolby 1997; Aldrich et al. 1999].

2.4 Computation Lock Coarsening

The basic idea behind computation lock coarsening is to transform sequences of operations that repeatedly acquire and release the same lock into sequences that acquire the lock once, perform the operations without further synchronization, then release the lock. In our example, after data lock coarsening, each processor will repeatedly acquire and release the lock in the `accumulator` object as it updates the sum and max accumulators. Computation lock coarsening reduces the synchronization frequency by eliminating the release and acquire between the update to the sum and max accumulators, then lifting the remaining acquire and release operations out of the `update` operation in the `accumulator` object, and out of the sequential loop in the `compute` operation in the `vector` object. Figure 5 presents the new `accumulator` and `vector` classes after the computation lock coarsening transformation.

The computation lock coarsening transformation works by examining the call graph for the code in the parallel phase. It finds *closed* operations, or operations

whose entire computation is sequential and acquires and releases only the lock in the distinguished object that the closed operation accesses. For each closed operation, the compiler generates code that acquires the lock at the beginning of the operation and releases the lock at the end. The compiler also generates specialized synchronization-free versions of each operation that the closed operation (directly or indirectly) invokes, and transforms the closed operation to invoke these synchronization-free versions. The compiler also treats the body of each parallel loop as a separate operation, enabling it to lift the lock acquire and release operations to include the entire body of the parallel loop.

While the computation lock coarsening transformation reduces the frequency with which the computation acquires and releases locks, it can also introduce a synchronization bottleneck. Even at the original lock granularity, the `accumulator` object in our example can become a sequential bottleneck if many of the elements in the vector are greater than zero. After lock coarsening, each processor acquires the lock before it even starts to execute its next block of iterations. The transformation therefore serializes the computation.

2.5 Adaptive Replication

One way to eliminate the sequential bottleneck is to use adaptive replication to replicate the `accumulator` object. Figure 6 presents a high-level version of the program after adaptive replication. The basic idea is to dynamically detect if the `accumulator` object is creating a synchronization bottleneck. If so, the program creates its own local version of the accumulator and performs all of the updates on the local version. The program dynamically detects the synchronization bottleneck using the `l.tryAcquire()` operation. This operation attempts to acquire the lock `l`, returning true if the acquire was successful. If the lock is held by a different processor, the operation returns false.

In our example, the generated code first uses the `lookup` operation to check for an existing local replica. If one exists, it performs the updates locally on that replica with no synchronization. If a local replica does not exist, it attempts to acquire the lock on the `accumulator` object. If the lock acquire attempt succeeds, the generated code updates the original `accumulator` object, then releases the lock. Finally, if another processor holds the lock, the code creates a new local replica, uses the `insert` operation to record the association between the original object and the replica on that processor, and performs all of the updates on the new replica. Whenever contention on the lock indicates the presence of a synchronization bottleneck, the generated code uses replication to eliminate the bottleneck. If there is no contention, the generated code avoids replication overhead by simply performing the updates on the original object.

2.6 Issues

There are several issues associated with the automatic application of adaptive replication. Our approach deals with each of these issues as follows:

—*Determining Which Objects to Replicate*: It is important to replicate only those objects that would otherwise cause synchronization bottlenecks. The program recognizes these objects using the `tryAcquire` construct. If a processor is

```
class accumulator {
  public:
  lock l;
  sumAccumulator sum;
  maxAccumulator max;
  accumulator *replicate() {
    accumulator *a = new accumulator();
    a->sum.value = 0;
    a->max.value = 0;
    insert(a,this);
    return a;
  }
  void update(double v) {
    sum.update(v);
    max.update(v);
  }
};
void combineReplicas() {
  for all local hash tables h
    for all pairs <original, replica> in h
      remove(h,original,replica);
      original->sum.value += replica->sum.value;
      if (original->max.value < replica-
>max.value) {
        original->max.value = replica->max.value;
      }
      delete replica;
    }
  }
}
class vector {
  double elements[N];
  public:
  void compute(accumulator *a) {
    int i;
    parallel for (i = 0; i < N; i += B) {
      int u = i+B;
      if (N < u) u = N;
      accumulator *r = lookup(this);
      if (r == NULL) {
        if (a->l.tryAcquire()) r = a;
        else r = a->replicate();
      }
      for (int j = i; j < u; j++) {
        if (elements[i] > 0.0) {
          r->update(elements[i]);
        }
      }
      if (r == a) a->l.release();
    }
    combineReplicas();
  }
};
```

Fig. 6. Vector and accumulator classes after adaptive replication.

unable to acquire an object's lock, it assumes that the object may cause a synchronization bottleneck. It therefore obtains a local replica of the object and performs the update locally on the replica.

—*Limiting Memory Consumption*: If a program replicates objects without limit, it may consume an unacceptable amount of memory. Our compiler generates code that records the amount of memory devoted to object replicas. Before it replicates an object, it checks that the amount of memory consumed by replicas does not exceed a predefined limit. If the allocation of a replica would exceed this limit, the code does not allocate the replica and instead forces the operation to wait until it can acquire the lock and execute on the original object. For clarity, the example code in Figure 6 eliminates this check.

—*Retrieving Replicas*: Each processor has its own local hash table in which it stores references to its object replicas. Each replica is indexed under the reference to the corresponding original object. The `insert` construct inserts a mapping from the original object to a replica, and, given a reference to an original object, the `lookup` construct retrieves the replica.

—*Initializing Replicas*: The updated instance variables in object replicas are initialized to the zero for the operator used to compute the updated value. In many cases objects also contain instance variables that are not updated during the course of the parallel computation. Because operations that execute on replicas may access these variables, their values from the original object are copied into the replica when it is created.

—*Combining Values*: At the end of the parallel phase, the generated code traverses the hash tables (recall that there is one hash table per processor) to find all of the replicas. As it visits each replica, it combines the updated values in the replica into the instance variables in the original object. It also removes the replica from the hash table and deallocates it.

The current version of the compiler generates code that performs the hash table traversals for different objects in parallel. For clarity, the code in Figure 6 performs the traversals sequentially.

A final issue is the order in which the generated code checks for an existing replica or attempts to acquire the lock in the original object. It is, unfortunately, necessary to check for a local replica before attempting to acquire the lock in the original object. We base our rationale for this order on two relative costs: (1) the cost of hash table lookups relative to the cost of lock acquisition attempts for objects that other processors frequently attempt to update and (2) the cost of updating local object replicas relative to the cost of updating objects that multiple processors frequently access.

Consider what may happen when a processor attempts to update an object that other processors are also attempting to update. With current implementations of locking primitives, the resulting attempts to acquire the object's lock may often generate expensive accesses to remote data, *even if the lock acquisition attempts do not succeed*.[1] On current architectures, we expect such lock

[1]Efficient lock implementations typically use synchronization instructions such as load linked/store conditional or compare and swap. When a processor uses these instructions to successfully acquire

```
class node {
  int value, sum;
  node *left, *right;
  public:
  void visit(int p) {
    sum = sum + p;
    if (left != NULL)
      left->visit(value);
    if (right != NULL)
      right->visit(value);
  }
};
```

Fig. 7. Sequential graph traversal.

acquisition attempts to take significantly longer than looking up a value in a locally available hash table. And even if a processor does acquire the lock and perform the update, it may often be the case that another processor performed the previous update. In this case, the update itself may generate additional accesses to remote data. In fact, the performance of early versions of the generated code that first attempted to acquire the lock, then checked for a local replica suffered from these two effects.

2.7 A Graph Traversal Example

The preceding accumulator example illustrates the interaction between lock coarsening and adaptive replication. We next present an example that illustrates the application of adaptive replication to a more complex graph traversal computation.

The program in Figure 7 implements a sequential graph traversal. The `visit` operation traverses a single node. It first adds the parameter `p` into the running sum stored in the `sum` instance variable, then recursively invokes the operations required to complete the traversal. The way to parallelize this computation is to execute the two recursive invocations of the `visit` operation in parallel. Our compiler is able to use commutativity analysis to statically detect this source of concurrency [Rinard and Diniz 1997].

Figure 8 presents the code that the compiler generates when it does not use adaptive replication. The transitions from sequential to parallel execution and from parallel back to sequential execution take place inside the `visit` operation. This operation first invokes the `parallelVisit` operation, then invokes the `wait` construct, which blocks until all parallel tasks created by the current task or its descendant tasks finish. The `parallelVisit` operation executes the recursive calls concurrently using the `spawn` construct, which creates a new task for each operation. A straightforward application of lazy task creation can increase the granularity of the resulting parallel computation [Mohr et al. 1990].

---

a lock, on most current machines it acquires exclusive access to the cache line in which the state of the lock is stored. When other processors attempt to acquire the lock, they fetch the current version of the cache line from the cache of the processor that most recently acquired the lock. These remote fetches typically take as much as two orders of magnitude longer than fetching a cache line from the local cache.

```
class node {
  lock l;
  int value, sum;
  node *left, *right;
  public:
  void visit(int p) {
    this->parallelVisit(p);
    wait();
  }
  void parallelVisit(int p) {
    l.acquire();
    sum = sum + p;
    l.release();
    if (left != NULL)
      spawn left-
>parallelVisit(value);
    if (right != NULL)
      spawn right-
>parallelVisit(value);
  }
};
```

Fig. 8. Parallel traversal without adaptive replication.

A problem with the code in Figure 8 is that it may suffer from synchronization bottlenecks if many of the nodes in the graph all refer to the same node. Because all of the updates use an operator (the + operator)[2] that is associative, commutative, and has a zero, it is possible to apply adaptive replication to eliminate these bottlenecks. Figure 9 presents a high level version of the code that the compiler generates when it uses this approach.

This computation illustrates several of the complications associated with applying adaptive replication in the context of general object-based computations. Computations may access an unbounded set of objects whose identities are determined only as the computation executes. It is, in general, not feasible to statically identify which objects will be involved in the computation, which objects will cause synchronization bottlenecks, or even if the computation will suffer from any synchronization bottlenecks at all. We believe that these uncertainties will negate any attempt to statically identify and replicate only those objects that will be potential sources of bottlenecks. We therefore developed an replication approach that adapts to the dynamic characteristics of the application by dynamically detecting bottlenecks and replicating only those objects required to eliminate the bottlenecks.

3. REPLICATION ANALYSIS AND CODE GENERATION

To generate code that uses adaptive replication, a compiler contains a *replication analysis algorithm*, which determines when it is legal to replicate objects,

[2]There may be some confusion between the two terms *operator* and *operation*. An operator is a binary function such as + that is used to combine two values. An operation is a piece of code associated with a class that executes on objects of that class. An example of an operation is the `visit` operation in Figure 7.

```
class node {
  lock l;
  int value, sum;
  node *left, *right;
  public:
  void visit(int p) {
    this->parallelVisit(p);
    wait();
    combineNodeReplicas();
  }
  void parallelVisit(int p) {
    node *replica = lookup(this);
    if (replica == NULL) {
      if (l.tryAcquire()) {
        sum = sum + p;
        l.release}();
        if (left != NULL)
          spawn(left->parallel_visit(value));
        if (right != NULL) {
          spawn right->parallel_visit(value);
          return;
        } else replica = this->replicate();
      }
      replica->replica_visit(p);
    }
    void replica_visit(int p) {
      sum = sum + p;
      if (left != NULL)
        spawn left->parallel_visit(value);
      if (right != NULL)
        spawn right->parallel_visit(value);
    }
    node *replicate() {
      node *replica = new node;
      replica->sum = 0;
      replica->value = value;
      replica->left = left;
      replica->right = right;
      insert(this, replica);
      return(replica);
    }
    friend void combine_node_replicas();
};
void combine_node_replicas() {
  for all local hash tables h {
    for all pairs <original, replica> in h {
      remove(h,original,replica);
      original->sum += replica->sum;
      delete replica;
    }
  }
}
```

Fig. 9. Parallel traversal with adaptive replication.

and a *code generation algorithm*, which generates code that uses adaptive replication to eliminate synchronization bottlenecks. We have implemented these algorithms in the context of a parallelizing compiler for object-based programs. The compiler uses commutativity analysis as its primary parallelization technique [Rinard and Diniz 1997]. Commutativity analysis is capable of parallelizing computations (such as marked graph traversals and computations that swap the values of instance variables) to which it is illegal to apply adaptive replication. We have therefore decoupled the commutativity analysis and replication analysis algorithms in the compiler. Replication analysis runs only after the commutativity analysis algorithm has successfully parallelized a phase of the computation. Note that the replication analysis cannot simply reuse the results of the commutativity analysis to find replicatable objects (objects that it can replicate without changing the semantics of the program). It is not always possible to replicate an object that is updated in a parallel phase, even if all of the operations on the object commute.

This section presents the replication analysis and code generation algorithms. In practice, we run the replication analysis after the lock coarsening transformation. As mentioned earlier, the synergistic interaction between these two transformations enables this combination to reduce the synchronization and replication overhead without risking the introduction of synchronization bottlenecks.

3.1 Notation

We next present some notation that we will use when we present the algorithms. The program defines a set of classes $\texttt{cl} \in \texttt{CL}$, a set of instance variables $\texttt{v} \in \texttt{V}$, and a set of operations $\texttt{op} \in \texttt{OP}$. We assume that no two classes share an instance variable. The program also defines a set of instance variables $\texttt{v} \in \texttt{V}$. The function **instanceVariables**(cl) returns the instance variables of the class cl. No two classes share an instance variable—that is, **instanceVariables**($\texttt{cl}_1$) $\cap$ **instanceVariables**($\texttt{cl}_2$) $= \emptyset$ if $\texttt{cl}_1 \neq \texttt{cl}_2$.

3.2 Program Representation

As part of the parallelization process, the commutativity analysis algorithm produces the set of operations that the parallel phase may invoke and the set of instance variables that the phase may update [Rinard and Diniz 1997]. It determines each of these sets by traversing the call graph of the phase. The set of operations is simply the set of operations in the call graph of the phase; the set of instance variables is simply the union of the sets of instance variables that the invoked operations may update. For each operation, the compiler also produces a set of *update expressions* that represent how the operation updates instance variables and a multiset of *invocation expressions* that represent the multiset of operations that the operation may invoke. There is one update expression for each instance variable that the operation modifies and one invocation expression for each operation invocation site. Except where noted, the update and invocation expressions contain only instance variables and parameters—the algorithm uses symbolic execution to eliminate local variables from the update

and invocation expressions [King 1976; Rinard and Diniz 1997]. The compiler manipulates the following kinds of update expressions:

—`v=exp`: An update expression of the form `v=exp` represents an update to a scalar instance variable `v`. The symbolic expression `exp` denotes the new value of `v`.

—`v[exp′]=exp`: An update expression of the form `v[exp′]=exp` represents an update to the array instance variable `v`.

—**for** (`i=exp`$_1$; `i<exp`$_2$; `i+=exp`$_3$) `upd`: An update expression of this form represents a loop that repeatedly performs the update `upd`. In this case, < can be an arbitrary comparison operator and += can be an arbitrary assignment operator. The induction variable `i` may appear in the symbolic expressions of `upd`.

—**if** (`exp`) `upd`: An update expression of the form **if** (`exp`) `upd` represents an update `upd` that is executed only if `exp` is true.

The compiler manipulates the following kinds of invocation expressions:

—`exp`$_0$→`op`(`exp`$_1$, ..., `exp`$_n$): An invocation expression `exp`$_0$→`op`(`exp`$_1$, ..., `exp`$_n$) represents an invocation of the operation `op`. The symbolic expression `exp`$_0$ denotes the distinguished object on which the operation will operate and the symbolic expressions `exp`$_1$, ..., `exp`$_n$ denote the parameters.

—**for** (`i=exp`$_1$; `i<exp`$_2$; `i+=exp`$_3$) `inv`: An invocation expression of this form represents a loop that repeatedly invokes the operation `inv`. In this case, < can be an arbitrary comparison operator and += can be an arbitrary assignment operator. The induction variable `i` may appear in the symbolic expressions of `inv`.

—**if** (`exp`) `inv`: An invocation expression of the form **if** (`exp`) `inv` represents an operation `inv` that is invoked only if `exp` is true.

For some operations, the compiler may be unable to generate update and invocation expressions that accurately represent the multiset of invoked operations. The commutativity analysis algorithm is unable to parallelize phases that may invoke such operations. Because the replication analysis and code generation algorithms run only after the commutativity analysis algorithm has successfully parallelized a phase, invocation expressions are available for all operations that the parallel phase many invoke.

3.3 Replication Conditions

The replication analysis algorithm is based on updates of the form `v = v` $\oplus$ `exp`, where $\oplus$ is an associative and commutative operator with a zero and `exp` does not depend on variables that are updated during the parallel phase.[3] We call such updates *replicatable updates*, because if all updates to a given instance variable `v` are of this form and use the same operator, then the final value of

[3]A zero for an operator $\oplus$ is an element z that has the property that $z \oplus v = v$ for all v. Note that the zero may be different for different operators: the zero for addition is 0, but the zero for multiplication is 1.

v is the same regardless of the order in which the individual contributions (the values of exp in the updates) are accumulated to generate the final result. In particular, accumulating locally computed contributions in local replicas, then combining the replicas at the end of the parallel phase, yields the same final result as sequentially accumulating the contributions into the original object. If all accesses to a variable v take place in replicatable updates to v, and all of the updates use the same operator, we call v a *replicatable variable*.

The replication analysis algorithm builds on the concept of replicatable variables as follows. In the transformed program, updates to replicated objects take place at the granularity of operations in the original program. For the generated program to produce the correct result, all operations that execute on replicated objects may update only replicatable variables. To determine if it is legal for an operation to execute on a replicated object, the analysis algorithm checks that the operation satisfies two conditions. The first condition is that the operation updates only replicatable variables. The second condition is that if the operation may invoke (either directly or indirectly) another operation with a replica as the distinguished object that the operation may update, then the invoked operation updates only replicatable variables.[4] If an operation satisfies these two conditions, we call the operation a *replicatable operation*. It is always legal to invoke a replicatable operation on a replica.

3.4 The Replication Analysis Algorithm

Figure 10 presents the replication analysis algorithm. The presented algorithm is simplified in the sense that it assumes that there is exactly one commutative, associative operator $\oplus$ with a zero.

The algorithm takes as parameters the set of invoked operations in the parallel phase, the set of updated variables, a function **updates**(op), which returns the set of update expressions that represent the updates that the operation op performs, and a function **invocations**(op), which returns the multiset of invocation expressions that represent the multiset of operations that the operation op invokes. There is also an auxiliary function called **variables**; **variables**(exp) returns the set of variables in the symbolic expression exp, **variables**(upd) returns the set of free variables in the update expression upd, and **variables**(inv) returns the set of free variables in the invocation expression inv.[5] The algorithm produces a set of instance variables that may be replicated and a set of operations that may execute on replicated versions of objects.

The algorithm first identifies the set of replicatable variables. It performs this computation by scanning all of the instance variable updates, eliminating

[4]The restrictions on replicatable variables and replicatable operations ensure that references to replicated objects are never stored in application data—they are stored only in the hash table that the generated program uses to look up replicas. The only way for one operation executing on a replicated object to invoke another operation on a replicated object is to use the this keyword at the operation invocation site. Any other expression used to identify the distinguished object of an invoked operation will never evaluate to a replica.

[5]The free variables of an update or invocation expression include all variables in the expression except the induction variables in expressions that represent for loops. In particular, the free variables in an update expression include the updated variable.

```
replicationAnalysis(invokedOperations, updatedVariables, updates, invocations)
  // Compute replicatable variables
  replicatableVariables = updatedVariables;
  for all op ∈ invokedOperations
    for all u ∈ updates(op)
      if (not isReplicatableUpdate(u, updatedVariables))
        replicatableVariables = replicatableVariables − variables(u);
    for all i ∈ invocations(op)
      replicatableVariables = replicatableVariables − variables(i);

  // Compute replicatable operations
  replicatableOperations = {op ∈ invokedOperations.
                            ∀u ∈ updates(op).updatedVariable(u) ∈ replicatableVariables}
  do
    s = {op ∈ replicatableOperations.∃i ∈ invocations(op).
         (i is of the form this.op(exp_1, ..., exp_n) or
         i is of the form if (exp)this.op(exp_1, ..., exp_n) or
         i is of the form for (i = exp_1; i < exp_2; i+ = exp_3)this.op(exp_1, ..., exp_n)) and
         operation(i) ∉ replicatableOperations}
    replicatableOperations = replicatableOperations − s
  while (s ≠ ∅)
  return ⟨replicatableVariables, replicatableOperations⟩;

isReplicatableUpdate(u, updatedVariables)
  if (u is of the form v = v ⊕ exp and variables(exp) ∩ updatedVariables = ∅)
    return true;
  if (u is of the form v[exp'] = v[exp'] ⊕ exp and
      (variables(exp) ∪ variables(exp')) ∩ updatedVariables = ∅)
    return true;
  if (u is of the form if (exp) upd and variables(exp) ∩ updatedVariables = ∅)
    return isReplicatableUpdate(upd, updatedVariables);
  if (u is of the form for (i = exp_1; i < exp_2; i+ = exp_3) upd
      and ∀_{1≤j≤3} variables(exp_j) ∩ updatedVariables = ∅)
    return isReplicatableUpdate(upd, updatedVariables);
  return false;

updatedVariable(u)
  if ( u is of the form v = exp ) return v;
  if (u is of the form v[exp'] = exp ) return v;
  if ( u is of the form if (exp) upd )
    return updatedVariable(upd);
  if (u is of the form for (i = exp_1; i < exp_2; i+ = exp_3) upd)
    return updatedVariable(upd);
```

Fig. 10. Replication analysis algorithm.

variables with updates that are not replicatable updates. The algorithm next scans the set of operations to identify the set of replicatable operations. For each operation, it tests if all of the operation's updates update replicatable variables and if the operation never invokes a nonreplicatable operation on a replica. If the operation passes both of these tests, it is classified as a replicatable operation.

The algorithm generalizes in a straightforward way to handle computations that contain multiple commutative and associative operators as follows. The

isReplicatableUpdate function accepts, as replicatable, updates performed using any of these operators. For each potentially replicatable variable, the algorithm records all of the operators used to update the variable. The algorithm has a notion of compatible operators; two operators are compatible if updates using these operators commute, reassociate, and have the same zero. For example, + and − are compatible operators, while + and / are not compatible. If all of the operators used to update a given variable are compatible, the variable is replicatable; otherwise, it is not replicatable. The part of the algorithm that deals with replicatable operations is unchanged. The version implemented in our prototype compiler can apply adaptive replication to computations that contain multiple commutative, associative operators with a zero.

Conceptually, the analysis divides the set of update expressions into equivalence classes, then checks that all update expressions in the same equivalence class use compatible operators. Any partition into equivalence classes is sound as long as any two update expressions that may update the same memory location are in the same equivalence class. Our current algorithm groups update expressions into the same equivalence class based on the names of the instance variables that they update—if two update expressions update the same instance variable, they are in the same equivalence class. It would be possible, however, to use memory disambiguation analyses such as pointer or array index analysis to come up with a finer and more precise partition.

3.5 Complexity of the Analysis

For each parallel phase, the replication analysis examines each update and invocation expression from each operation in the parallel phase. The complexity is therefore determined by the complexity of extracting and simplifying the update and invocation expressions. In the worst case, these operations as implemented in our compiler can take time exponential in the size of the expressions in the original program, although we have not observed this behavior in practice. If the update and invocation expressions were to be used only for replication analysis, we believe it would be possible to develop algorithms that do not use algebraic laws to fully simplify the expressions and therefore do not suffer from the possibility of exponential execution time.

In practice, the execution time of the compiler is dominated by the commutativity analysis algorithm, which examines all pairs of operations in each parallel phase. The replication analysis examines each operation once. For all of our benchmark programs, the commutativity analysis algorithm executes in less than two seconds on a Sun Microsystems Ultra I workstation with 64 Mbytes of memory [Rinard and Diniz 1997]. The addition of the replication analysis to this compiler does not significantly increase the compilation time.

3.6 The Code Generation Algorithm

The generic code generation algorithm starts with the set of replicatable variables and replicatable operations. It generates two versions of each replicatable operation: the standard version and the replica version.

The standard version executes on original objects, not replicas. If it may update an object, it first checks to see if a replica has already been created. If so, it invokes the replica version to perform the update on the replica. If not, it uses a `tryAcquire` construct to attempt to acquire the lock in the object that it will update. If the lock is acquired, the standard version performs the updates as in the original program, then releases the lock and completes its execution by invoking the standard version of each operation that it should invoke. If the lock is not acquired, the standard version invokes the replica creation operation to create a replica. If the replica creation operation returns NULL, it was unable to create a replica because of memory consumption constraints. In this case, the standard version waits until it acquires the lock in the original object, then performs the update on the object. If the replication creation operation successfully created a replica, the standard version invokes the replica version to perform the update on the replica.

The replica version executes on replicas, not original objects. It performs all of its updates to the replica without synchronization. It invokes the standard version of each operation that it should invoke unless (1) the invoked operation will operate on the same object as the replica version, and (2) this object is identified at the call site using the `this` keyword (which denotes distinguished object that the replica version executes on). In this case, it invokes the replica version of the invoked operation. This invocation strategy reduces the overhead by eliminating the trip through the standard version for sequences of operations on the same replicated object.

The code generation algorithm also produces a replica creation operation. This operation first performs a *memory consumption check*: it checks if allocating a replica would exceed the predefined limit on the amount of memory consumed by replicas. If not, it allocates a replica, initializes all of the replicatable variables in the object to the zero for the operator used to accumulate contributions to the variable, and initializes all of the other instance variables to the values in the original object. It then inserts the replica in the hash table, indexed under the original object, and returns the replica.

If the replica allocation would exceed the predefined limit, the replica creation operation returns NULL. In this case, the standard version waits until it acquires the lock on the original object and performs the operation on that object. In the example in Figure 6, the `replicate` operation is the replica creation operation. For clarity, we omitted the memory consumption check.

Finally, the code generation algorithm produces a combine function that is invoked at the end of the parallel phase. This function traverses the hash table to find all of the replicas, combines the values in the replicas to generate the correct final values in the original objects, then clears the hash table and deallocates the replicas. We considered several different ways to traverse the hash table. The first sequentially combines all of the values in the hash table. While this approach minimizes the amount of exploited concurrency, in many cases the size of the combination computation is small enough relative to the previous parallel phase to make it a feasible alternative. The second approach assigns different objects to different processors, with each processor combining

the values in all of the replicas of its assigned objects. This is the approach we use in our implemented compiler. It often effectively parallelizes the combination and is simple to implement. The potential drawbacks include memory traffic as the processors fetch the data in remote replicas and the possibility of limited concurrency if the computation replicated few objects. The third alternative would combine the replicas for each object in parallel using some sort of parallel reduction operation. An advantage of this alternative is that it would parallelize the combination even for computations that replicate small numbers of objects. It would also manage the communication of remote replicas more effectively than the other alternatives.

3.7 Interaction with Lock Coarsening

There is a synergistic interaction between lock coarsening and adaptive replication. The lock coarsening transformation reduces the frequency with which the program attempts to acquire locks and look up replicas. The adaptive replication transformation eliminates any synchronization bottlenecks that the lock coarsening transformation may have introduced. Our current compiler first applies the lock coarsening transformation, then the adaptive replication transformation. The result is an efficient program that maximizes the amount of exposed concurrency while minimizing the overhead of acquiring locks and looking up replicas.

3.8 Programs Written in Different Styles

To ensure the legality of the replication transformation, the replication analysis algorithm tests that, within a given phase, all updates to a given replicated piece of data are in fact replicatable updates. The compiler must therefore characterize the shared data that each atomic operation updates and check all updates to each piece of shared data. In general, aliasing can significantly complicate the static characterization of the association between atomic operations and updated data. Our current analysis uses the structure of the object-based approach to simplify these tasks.

In object-based computations, each atomic operation accesses only (1) the instance variables of the object on which the operation executes, (2) local data that is not shared between threads, and (3) data that may be read but not written during the analyzed parallel phase. These restrictions enable the compiler to apply the replication analysis algorithm at the granularity of the instance variables of each object. The absence of aliasing ensures that all updates to each instance variable access that variable directly using its name, instead of performing the access anonymously and indirectly using a pointer to the variable. This restriction allows the compiler to find all accesses to a given instance variable by scanning the program to find all accesses that mention the instance variable's name.

To use adaptive replication with programs written in a different style, the compiler would have to use a different approach to associate updated shared data with atomic operations. For programs that use dynamic allocation and pointers in their full generality, we anticipate that the compiler would have to

use some form of pointer analysis to disambiguate accesses via pointers [Emami et al. 1994; Wilson and Lam 1995; Rugina and Rinard 1999]. For programs without aliasing, the program could simply use the names of the accessed variables to perform the association. We anticipate that this approach would work well for traditional scientific programs that access global variables and statically allocated arrays.

Once the compiler had determined the legality of replication, it would have to decide on a replication granularity. To simplify the code generation algorithm, our current compiler replicates data at the granularity of objects, even though the program may never access some of the instance variables in the replicas. For programs that access statically allocated arrays, the compiler could replicate data either at the granularity of the entire array (potentially wasting space if there is little or no contention for most of the array elements) or at the granularity of individual array elements (increasing the overhead associated with updating each element).

Another issue involves the recognition of sequential and parallel phases. Our current implementation relies on the commutativity analysis algorithm to identify the phases that it has parallelized. The application of adaptive replication to general parallel programs would require an analysis that recognized parallel phases. The difficulty of this task depends on characteristics of the constructs used to express the parallel computation. For structured constructs such as parallel loops and parbegin/parend, a simple control flow analysis would suffice. For more general languages with unstructured forms of multithreading (e.g., Java), the analysis would have to match thread creation and termination constructs to separate the sequential and parallel phases.

4. EXPERIMENTAL RESULTS

We next present experimental results that characterize the performance and memory impact of both adaptive replication by itself and the combination of lock coarsening and adaptive replication. We present results for three automatically parallelized applications: Water [Singh et al. 1992], which simulates water molecules in the liquid state, Barnes-Hut [Barnes and Hut 1986], a hierarchical N-body solver, and String [Harris et al. 1990], which builds a velocity model of the geology between two oil wells. Each application performs a complete computation of interest to the scientific computing community. Water consists of approximately 1850 lines of sequential C++ code, Barnes-Hut consists of approximately 1500 lines of sequential C++ code, and String consists of approximately 2050 lines of sequential C++ code.

4.1 Application History

We obtained all three applications as part of our previous research on commutativity analysis [Rinard and Diniz 1997]. Our goal was to obtain a set of representative computations that were expressed in object-based form, with the data grouped into objects and the computation expressed as operations on

these objects. We recoded all of the applications into this form starting with versions written in C. The history of the applications is as follows.

—*Water*: Water computes the energy potential of a set of water molecules in the liquid state. The main data structure is an array of molecule objects. Almost all of the compute time is spent in two $O(N^2)$ phases, where N is the number of molecules. One phase computes the total force acting on each molecule; the other phase computes the potential energy of the collection of molecules.

The original source of Water is the Perfect Club benchmark MDG, which is written in Fortran [Berry et al. 1989]. Several students at Stanford University translated this benchmark from Fortran to C as part of a class project. We obtained the sequential C++ version by translating this existing sequential C version to C++. As part of this translation process, we identified intuitively appealing groupings of related data into objects and restructured the computation as a sequence of operations on these objects. For example, we developed a class with a three-dimensional vector of values, then used this class as a building block in a variety of higher-level classes such as atom and molecule classes.

As part of the translation process, we converted the $O(N^2)$ phases to use auxiliary objects tailored for the way each phase accesses data. Before each phase, the computation loads relevant data into an auxiliary object. At the end of the phase, the computation unloads the computed values from the auxiliary object to update the molecule objects. This modification increases the precision of the data usage analysis in our base parallelizing compiler, enabling it to recognize the concurrency in the phase.

—*Barnes-Hut*: Barnes-Hut simulates interactions between a set of bodies. It performs well, in part, because it employs a sophisticated pointer-based data structure: a space subdivision tree that dramatically improves the efficiency of a key phase in the algorithm. The space subdivision tree organizes the data as follows. The bodies are stored at the leaves of the tree; each internal node represents the center of mass of all bodies below that node in the tree. Each iteration of the computation first constructs a new space subdivision tree for the current positions of the bodies. It then computes the center of mass for all of the internal nodes in the new tree. The force computation phase executes next; this phase uses the space subdivision tree to compute the total force acting on each body. The final phase uses the computed forces to update the positions of the bodies.

We obtained sequential C++ code for this computation by acquiring the explicitly parallel C version from the SPLASH-2 benchmark set [Woo et al. 1995], then removing the parallel constructs to obtain a sequential version written in C. We then translated the sequential C version into sequential C++, obtaining a clean object-based version of the program. Most of the modifications involved the conversion of C structs into C++ objects. We preserved the memory allocation strategy of the C program, which allocated its data structures as a unit in the form of large arrays of structs, then overlaid linked data structures on this array using pointers to individual C structs within

the array. The C++ version uses basically the same allocation strategy, but allocates large arrays of objects instead of structs. Once we had encapsulated the key data structures in objects, we restructured the computation to use operations on these objects instead of directly accessing the fields of the objects.

As part of the translation, we eliminated several computations that dealt with parallel execution. For example, the parallel version used costzones partitioning to schedule the force computation phase [Singh 1993]. The sequential version eliminated the costzones code and the associated data structures. We also split a loop in the force computation phase into three loops. This transformation exposed the concurrency in the force computation phase, enabling the compiler to recognize that two of the resulting three loops could execute in parallel. As part of this transformation, we also introduced a new instance variable into the body class. The new variable holds the force acting on the body during the force computation phase.

When we ran the C++ version, we discovered that abstractions introduced during the translation process degraded the sequential performance. We therefore hand optimized the computation by removing abstractions in the performance-critical parts of the code until we had restored the original performance. These optimizations do not affect the parallelization; they simply improve the base performance of the computation.

—*String*: String uses seismic travel-time inversion to construct a two-dimensional discrete velocity model of the geology between two oil wells. Each element of the velocity model records how fast sound waves travel through the corresponding part of the geology. The seismic data are collected by firing nondestructive wave sources in one well and recording the waves digitally as they arrive at the other well. The travel times of the waves can be measured from the resulting seismic traces. The application uses the travel-time data to iteratively compute the velocity model. The computationally intensive phase of the application traces rays from one well to the other. The velocity model determines both the simulated path and the simulated travel time of each ray. The computation records the difference between the simulated and the measured travel times and backprojects the difference linearly along the path of the ray. At the end of the phase, the computation uses the backprojected differences to construct an improved velocity model. The process continues for a specified number of iterations. The serial computation stores the velocity model in a one-dimensional array and the backprojected differences in another one-dimensional array. Each element of the difference array stores the running sum of the backprojected differences for the corresponding element of the velocity model.

The first author obtained a version of the String application from researchers in the Stanford Geophysics department [Harris et al. 1990]. This version was written in a combination of Fortran and C, with the majority of the computation written in Fortran and some dynamic memory allocation procedures written in C. As part of the first author's Ph.D. research, this Fortran version was translated first into C, then into Jade, an implicitly

parallel extension of C [Rinard and Lam 1998]. We translated the Jade version into C++; the primary modifications involved the encapsulation of the velocity model and backprojected difference arrays into a class that exported operations on the arrays and restructuring the computation to use these operations.

4.2 Application Characteristics

All of the applications have access patterns that are difficult to analyze statically. Water uses references to manipulate a statically unbounded number of objects. The contention patterns depend on the precise timing of the executions of the parallel threads, which varies from run to run. Barnes-Hut uses a linked data structure (a space subdivision tree) and even though there is no contention in this application, the compiler would need to use some form of shape analysis [Chase et al. 1990; Ghiya and Hendren 1996; Sagiv et al. 1998] to statically recognize this fact. Unlike Water and Barnes-Hut, String allocates a small number of objects and it would be possible at compile time to determine the precise referencing pattern of the application at the granularity of these objects. But the primary object in this computation contains two large arrays. The computation does not access the elements of these arrays in any statically predictable way, in fact, the access pattern is a function of the input contents of the array and is therefore irregular and determined only as the program runs.

The parallel phases in all of the computations can be viewed as generalized reductions in the sense that all accesses to shared data combine a set of computed contributions. However, the parallel phases typically update multiple dynamically determined locations in multiple dynamically determined objects rather than computing a simple reduction into a single variable or performing a regular parallel prefix operation.

We believe that these applications are reasonably representative of a range of scientific computations written in object-based style, and that the adaptive replication optimization will prove to be useful for other such applications. Its utility for other kinds of multithreaded computations will depend in large part on the characteristics of these computations. We expect the technique to be useful whenever a group of threads computes results that are combined into a shared data structure. Potential situations where this might arise include bookkeeping operations in a multithreaded web server and optimized implementations of atomic operations in multithreaded database systems [Gray and Reuter 1993].

4.3 Methodology

We implemented a prototype parallelizing compiler that uses commutativity analysis as its basic analysis approach [Rinard and Diniz 1997]. The compiler includes an automatic lock coarsening algorithm and an automatic replication algorithm [Diniz and Rinard 1998; Rinard and Diniz 1999]. Flags determine the lock coarsening and replication policies that the generated code uses. We used the compiler to generate code with the following replication and lock coarsening

Table I. Execution Times for Water (seconds)

	Processors				
Version	1	4	8	16	24
Sequential	164.26	—	—	—	—
No Coarsening, No Replication	174.65	52.43	29.80	25.62	29.25
No Coarsening, Adaptive Replication	184.14	50.73	26.34	14.16	10.15
No Coarsening, Full Replication	197.57	51.69	27.12	14.94	10.94
Coarsening, No Replication	170.66	49.63	26.27	38.57	49.56
Coarsening, Adaptive Replication	169.27	46.99	24.75	13.08	9.54
Coarsening, Full Replication	166.57	45.05	22.97	12.40	8.98

policies:

—*No Coarsening*: The compiler applies no lock coarsening transformation. There is one lock per object and each operation that accesses an updated object acquires and releases that lock.

—*Coarsening*: The compiler applies both data lock coarsening and computation lock coarsening at the most aggressive level.

—*No Replication*: There is no replication of updated objects.

—*Adaptive Replication*: The generated code uses adaptive replication.

—*Full Replication*: Whenever possible, the generated code performs updates on local replicas.

We ran the applications with all possible combinations of the lock coarsening and replication policies. The exception is String, for which we did not run the Coarsening and No Replication combination. For this application, this combination completely serializes the execution.

We collected experimental results for the applications running on an SGI Challenge XL multiprocessor with 24 100 MHz R4400 processors and 768 Mbytes of memory running IRIX version 6.2. We compiled the generated parallel programs using version 7.1 of the MipsPro compiler from Silicon Graphics. The generated code uses the most efficient lock implementation available on this platform. The acquire is implemented as an inlined code sequence that uses `ll` and `sc` to atomically test and set a value that indicates whether the lock is free or not [Heinrich 1993]. The release simply clears the value.

4.4 Water

Table I presents the execution times for Water. In all applications, we report the minimum time from four runs for the one and two processor executions. For all other numbers of processors, we report the minimum time from at least eight runs. Half of the runs have profiling turned on;[6] half have profiling turned off. Table II presents the standard deviations of the execution times for the runs without profiling. Figure 11 presents the speedup curves. The speedup curves plot the running time of the original sequential version of each application divided by the running time of the automatically parallelized version as a function of the number of processors executing the parallel version. The original

[6]This profiler uses program-counter sampling [Graham et al. 1982; Knuth 1971].

Table II. Standard Deviations of Execution Times for Water (seconds)

	Processors				
Version	1	4	8	16	24
Sequential	25.65	—	—	—	—
No Coarsening, No Replication	0.12	0.29	0.64	0.05	0.89
No Coarsening, Adaptive Replication	31.04	0.55	0.07	0.14	0.20
No Coarsening, Full Replication	2.37	0.53	0.35	0.15	0.12
Coarsening, No Replication	0.13	0.44	0.50	0.99	2.87
Coarsening, Adaptive Replication	0.41	0.64	4.21	2.08	0.78
Coarsening, Full Replication	0.14	1.94	0.30	0.08	0.18

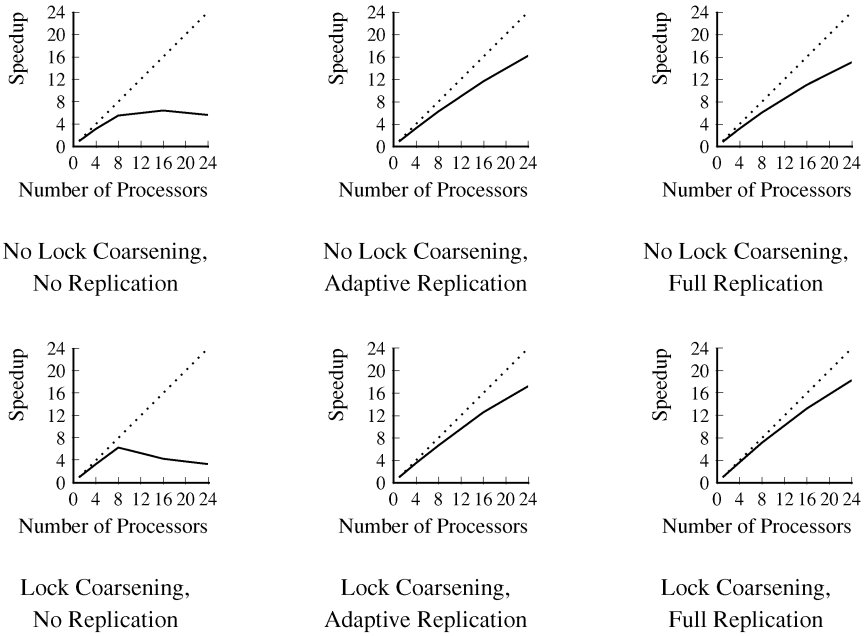


Fig. 11. Speedups for Water.

sequential version is a standard sequential C++ program that executes with no parallelization or synchronization overhead.

For this application, replication eliminates a key synchronization bottleneck, enabling the application to scale well with the number of processors. Applying the lock coarsening transformation without replication exacerbates the synchronization bottleneck, but the combination of lock coarsening and replication both eliminates the synchronization bottleneck and reduces the synchronization and replication overhead.

We used program counter sampling [Graham et al. 1982; Knuth 1971] to measure how much time each version spends in different parts of the parallel computation. We break the execution time down into the following components:

—*Replication*: Time spent because of replication. This component includes time spent allocating and deallocating replicas, initializing replicas, looking up replicas, and combining the values in replicas back into the original objects

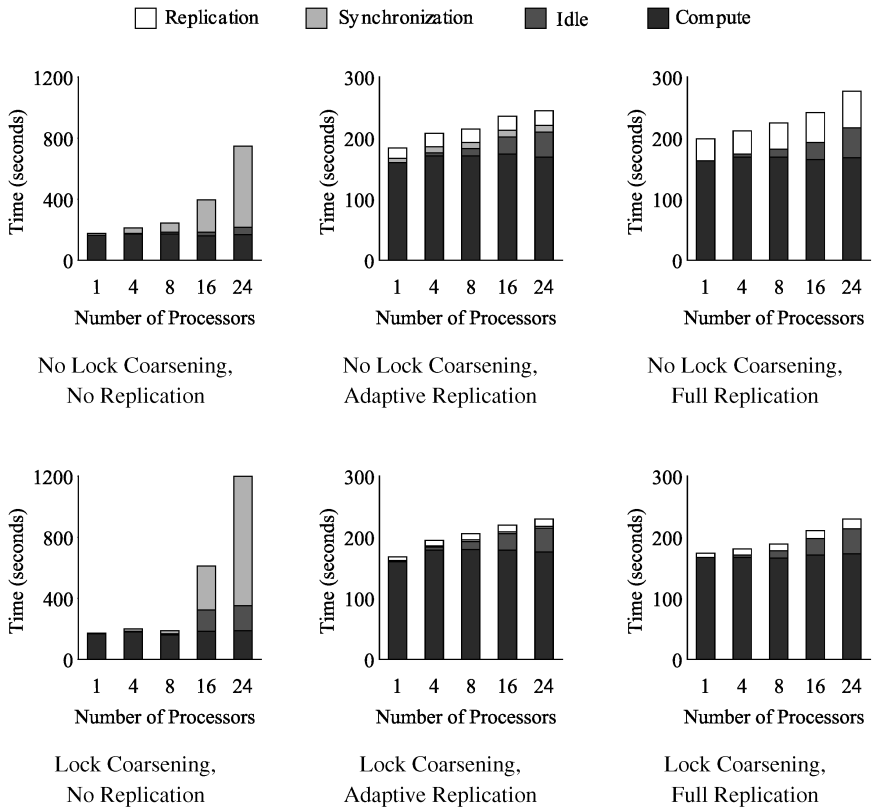


Fig. 12. Time breakdowns for Water (Note change of axis on No Replication versions).

at the end of parallel phases, including the time spent idle waiting for other processors to finish combining replicas.

—*Synchronization*: Time spent acquiring and releasing locks, including time spent waiting to acquire locks held by other threads. Synchronization bottlenecks show up as a large amount of time spent in this component.

—*Idle*: Time spent idle. All but one processor is idle during sequential phases of the computation; processors may also be idle during parallel phases if the program has poor load balancing. In all of our applications, idle time during sequential phases accounts for the vast majority of the total idle time.

—*Compute*: Time spent performing useful computation from the application.

Figure 12 presents the time breakdowns for Water.[7] These breakdowns clearly show the synchronization bottleneck in the versions without replication and the elimination of the bottleneck in the versions with replication. Lock coarsening has different effects depending on the presence or absence

[7]For each component, the size of the part of the bar dedicated to that component corresponds to the sum over all processors of the amount of time the processor spends in that component. The total height of the bar divided by the number of processors is therefore the running time of the application on that number of processors.

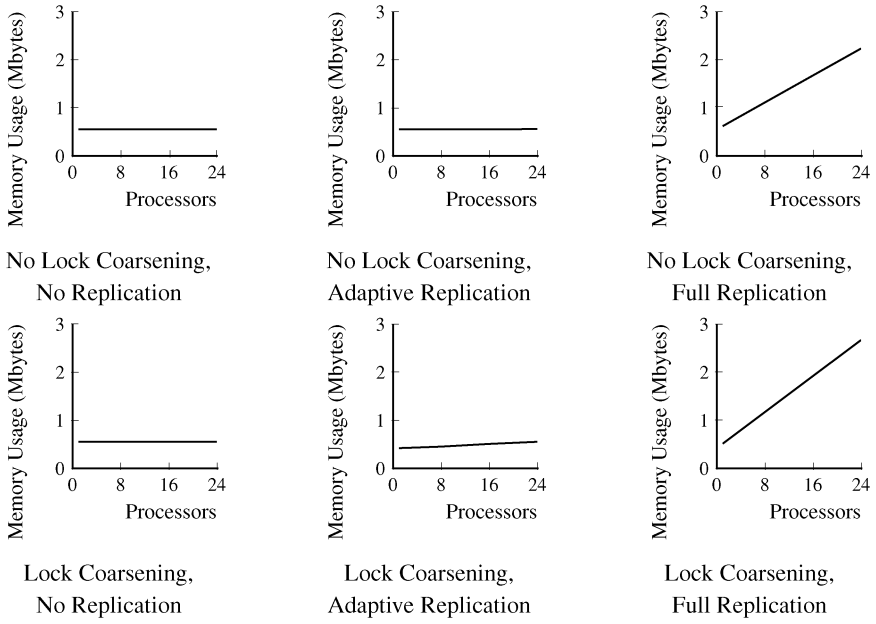


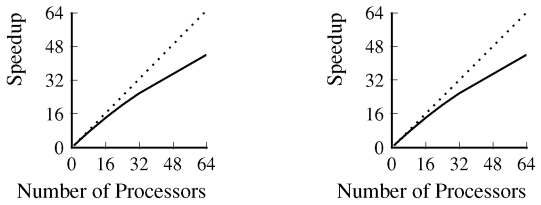
Fig. 13. Peak memory usage for Water.

of adaptive replication. Without replication, lock coarsening makes the synchronization bottleneck worse. With adaptive replication or full replication, it reduces the replication overhead.

Figure 13 presents the *peak memory usage* for Water. The peak memory usage measures the maximum amount of memory allocated to original objects or replicas during the computation. The peak memory usage for the versions without replication does not vary with the number of processors. With adaptive replication, the peak memory usage increases slightly with the number of processors. Lock coarsening slightly increases the amount of replication in the adaptive replication version. We attribute this slight increase to the increase in size of the critical sections. With full replication, the peak memory usage increases significantly with the number of processors.

The combination of lock coarsening and adaptive replication works well for Water. It eliminates the synchronization bottlenecks that degrade the performance of the versions without replication and it reduces the lock and replication overheads of the versions without lock coarsening. The peak memory usage of the adaptive replication versions is significantly less than that of the full replication versions, which indicates that it is possible to eliminate the synchronization bottleneck by replicating only a small amount of data.

We combine the experimental results for Water with an analysis of its behavior to obtain an analytic performance model for this application and applications with similar characteristics. To obtain this model we first subdivide the compute time into the *parallel time* (the time spent in parallel phases) and the *sequential time* (the time spent in sequential phases). We then characterize the effect of the replication on the performance. For both adaptive replication



Lock Coarsening, Adaptive Replication

Lock Coarsening, Full Replication

Fig. 14. Analytically generated speedups for Water.

and full replication, the replication overhead is dominated by the execution of the code that checks for the presence of replicas. We refer to this component as the *replica check time*. This component of the execution time is part of the normal execution of each parallel task and remains relatively constant as the number of processors increases. The time spent creating and combining replicas, on the other hand, grows (at most linearly) as the number of processors (and hence the contention) increases. We model this overhead as the *replica processing time*, which is the amount of time each processor spends creating and combining replicas. If each shared object is replicated by all processors, and there are enough shared objects to parallelize the combination computation, the replica processing time should (disregarding memory system effects) not vary as the number of processors increases. Note that in Water, all of these replication overheads are parallelized along with the computation in the parallel phases. In particular, the computation tends to replicate enough objects so that the replica combinations execute in parallel.

Given this analysis, we can model the execution time with the following expression:

$$\frac{\text{parallel time}}{\text{number of processors}} + \text{sequential time} + \frac{\text{replica check time}}{\text{number of processors}} + \text{replica processing time}$$

Using the profiling information, we chose the following representative values: a parallel time of 165 seconds, a sequential time of 1 second, a replica check time of 5 seconds, and replica processing times of .1 seconds for full replication and 0 seconds for adaptive replication. Figure 14 presents speedup curves from this model up to 64 processors. While this analysis fails to model important aspects of the parallel computation such as load imbalancing and memory system effects, it does provide some insight into the reasons for the performance of the Water application and similar applications.

4.5 Barnes-Hut

Table III presents the execution times for Barnes-Hut, Table IV presents the standard deviations of these execution times and Figure 15 presents the corresponding speedups. Figure 16 presents the time breakdowns. Without lock coarsening, the application has significant synchronization and/or replication

Table III. Execution Times for Barnes-Hut (seconds)

Version	Processors				
	1	4	8	16	24
Sequential	136.33	—	—	—	—
No Coarsening, No Replication	165.47	45.83	24.37	14.30	10.99
No Coarsening, Adaptive Replication	242.61	64.19	34.03	19.28	14.33
No Coarsening, Full Replication	246.70	58.60	30.60	17.36	13.07
Coarsening, No Replication	139.78	37.63	20.79	12.47	9.67
Coarsening, Adaptive Replication	139.85	38.02	21.16	12.67	9.94
Coarsening, Full Replication	164.28	41.02	22.26	13.00	10.21

Table IV. Standard Deviations of Execution Times for Barnes-Hut (seconds)

Version	Processors				
	1	4	8	16	24
Sequential	0.67	—	—	—	—
No Coarsening, No Replication	0.24	0.55	0.20	0.46	0.52
No Coarsening, Adaptive Replication	0.21	0.40	1.68	0.32	1.46
No Coarsening, Full Replication	1.14	0.27	0.21	2.06	1.22
Coarsening, No Replication	1.13	0.35	0.29	0.24	0.11
Coarsening, Adaptive Replication	0.10	0.31	16.53	5.88	5.10
Coarsening, Full Replication	0.44	0.75	0.18	0.12	6.07

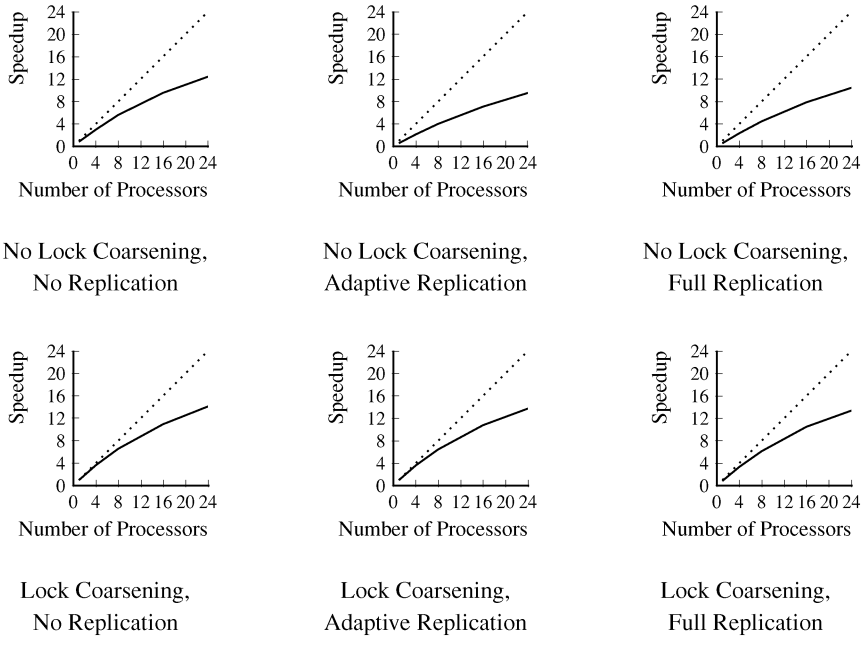


Fig. 15. Speedups for Barnes-Hut.

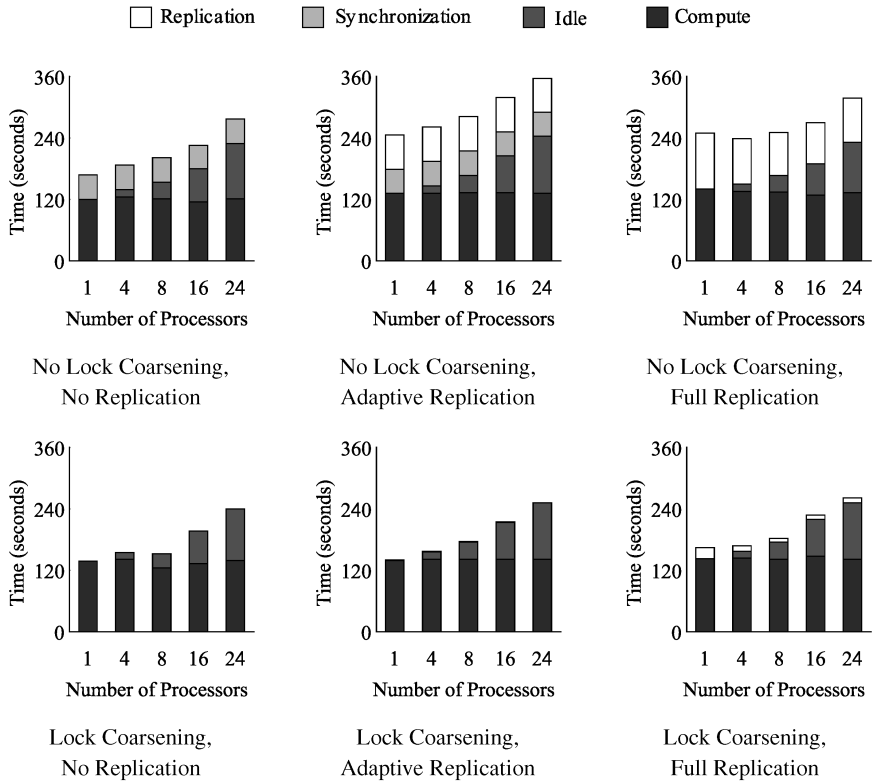


Fig. 16. Time breakdowns for Barnes-Hut.

overhead. With lock coarsening, this overhead drops to negligible levels. The peak memory usage results in Figure 17 show that the versions with adaptive replication use the same amount of memory as the versions without replication—for this application, the adaptive replication versions never replicate an object. Note that the generated code for the adaptive replication versions looks up a potential replica before attempting to acquire the lock on a replicatable object. The adaptive replication versions therefore have replication overhead even though they never replicate an object. The full replication version has the same space overhead for all numbers of processors. This computation simulates the interactions of number of bodies. It stores the bodies in the leaves of a space subdivision tree with the parallelized versions executing one task for each leaf of the tree. That task updates the leaf and reads other data structures. With this computation pattern, each leaf object is updated by one and only one processor, so the full replication version replicates each leaf object once regardless of the number of processors executing the computation. The resulting space overhead does not vary with the number of processors.

For Barnes-Hut, lock coarsening by itself generates the best performance, with the combination of lock coarsening and adaptive replication coming in a close second. Without lock coarsening, there is significant synchronization and/or replication overhead; with full replication, the application uses more

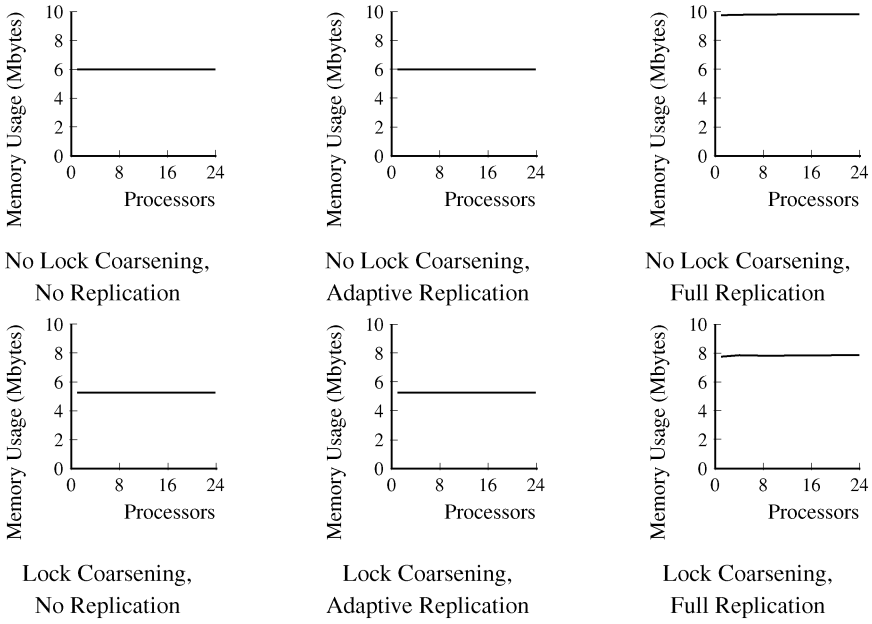


Fig. 17. Peak memory usage for Barnes-Hut.

memory than it needs to. Lock coarsening also drives down the peak memory usage for the versions with replication. We attribute this reduction to the fact that lock coarsening replaces multiple locks in one of the classes of objects with a single lock, driving down the size of the objects.

The analytical performance model for Barnes-Hut differs slightly from that of Water. In Barnes-Hut, each updated object is accessed by only one processor and there is no contention. Thus, with adaptive replication, there is no replica processing time at all; with full replication, the replica processing time decreases as the number of processors increases because each processor is responsible for combining the replicas of fewer objects. The total amount of time that the application spends on replication overhead (the *total replica time*) does not vary as the number of processors increases. Given this analysis, we can model the execution time with the following expression:

$$\frac{\text{parallel time}}{\text{number of processors}} + \text{sequential time} + \frac{\text{total replica time}}{\text{number of processors}}.$$

Using the profiling information, we chose the following representative values: a parallel time of 135 seconds, a sequential time of 4 seconds, and a total replica time of .5 seconds for adaptive replication and 7.5 seconds for full replication. Figure 18 presents speedup curves from this model up to 64 processors. The primary reason the application does not scale is the sequential time, which becomes an increasingly large percentage of the wall-clock execution time of the application as the number of processors increases. The primary sequential bottleneck is the subcomputation that constructs the space subdivision tree. It is possible to parallelize this subcomputation, and hand-parallelized versions

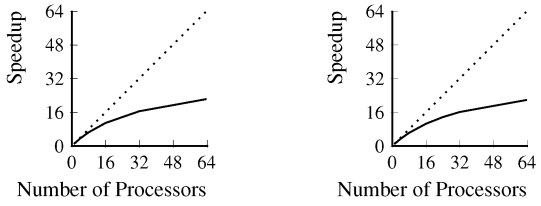


Fig. 18. Analytically generated speedups for Barnes-Hut.

Table V. Execution times for String (seconds)

	Processors				
Version	1	4	8	16	24
Sequential	881.27	—	—	—	—
No Coarsening, No Replication	896.99	236.36	126.86	78.76	89.60
No Coarsening, Adaptive Replication	921.86	233.49	116.10	59.54	42.18
No Coarsening, Full Replication	912.85	229.95	117.10	59.74	42.32
Coarsening, Adaptive Replication	892.70	224.45	113.78	60.38	45.06
Coarsening, Full Replication	897.98	223.73	113.89	60.61	44.91

Table VI. Standard Deviations of Execution Times for String (seconds)

	Processors				
Version	1	4	8	16	24
Sequential	3.45	—	—	—	—
No Coarsening, No Replication	2.71	1.96	0.35	2.25	27.99
No Coarsening, Adaptive Replication	22.57	4.10	9.59	0.59	0.60
No Coarsening, Full Replication	0.42	1.87	0.47	0.33	0.39
Coarsening, Adaptive Replication	4.29	3.63	0.64	0.20	0.18
Coarsening, Full Replication	6.85	1.58	0.76	0.11	0.55

do so, but the resulting parallel computation is nondeterministic in the sense that the detailed structure of the new tree may vary depending on the execution speed of the parallel tasks. Our parallelizing compiler therefore does not exploit the concurrency available in this subcomputation.

4.6 String

Table V presents the execution times for String. Table VI presents the standard deviations for these execution times. Figure 19 presents the corresponding speedup curves. We omit performance results for the version with lock coarsening and no replication—this combination completely serializes the execution and the program does not scale at all. With replication, lock coarsening has a negligible effect on the performance. The time breakdowns in Figure 20 show that the version without replication suffers from a serious synchronization bottleneck at 24 processors. Replication completely eliminates this bottleneck, at the cost of a modest amount of replication overhead. Without replication, the performance peaks at 16 processors, then rapidly falls off. With replication, the application scales almost linearly with the number of processors. The peak

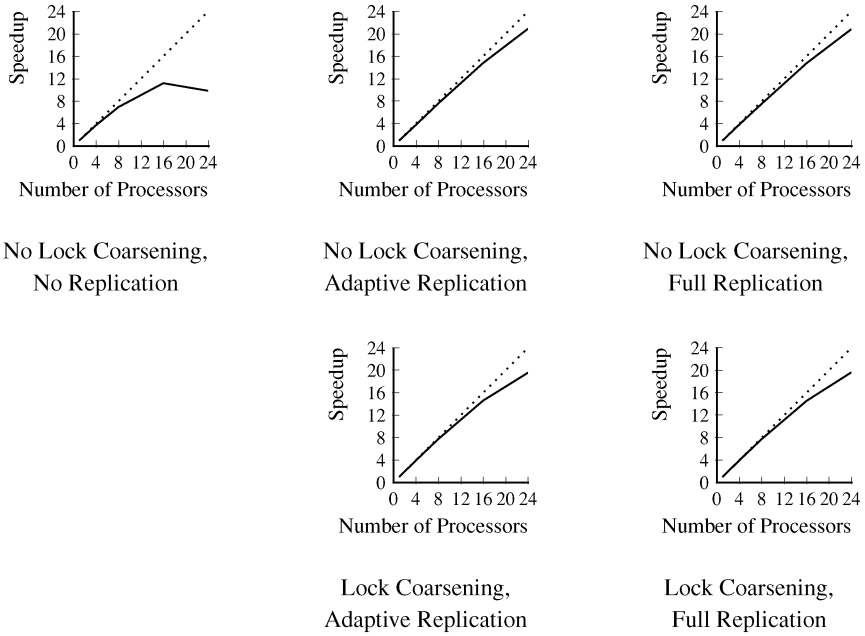


Fig. 19. Speedups for String.

memory usage graphs in Figure 21 show that both adaptive replication and full replication significantly increase the memory usage—both versions completely replicate a large object. The overall memory usage is still acceptable, however.

The analytical performance model for String differs slightly from those of Water and Barnes-Hut. In String, there is one large object that all threads update. Thus, with both adaptive replication and full replication, one processor is responsible for combining all of the replicas, and the other processors are idle because there is only one replicated object. Our model therefore uses the *replication combination time*, or the time required to combine one replica back into the original object. Given this analysis, we can model the execution time with the following expression:

$$\frac{\text{parallel time}}{\text{number of processors}} + \text{sequential time} + \text{replica combination time} \times \text{number of processors}$$

Using the profiling information, we chose the following representative values: a parallel time of 890 seconds, a sequential time of 5 seconds, and a replication combination time of .3 seconds. Figure 22 presents speedup curves from this model up to 64 processors. The primary reason the application does not scale is the replica combination time, which acts as a sequential bottleneck at large numbers of processors. The solution to this problem is to parallelize the combination of replicas into the single object. One reasonable approach would be to assign different array elements to different processors, with each processor combining the values in its array elements back into the original object.

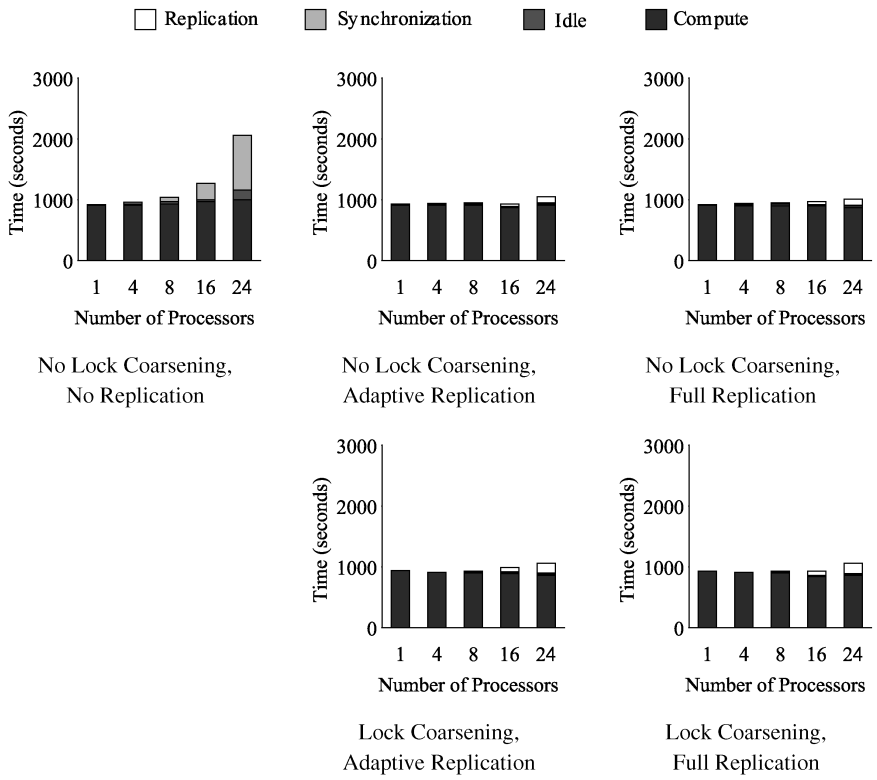


Fig. 20. Time breakdowns for string.

5. RELATED WORK

In this section we discuss related work in the area of reduction analysis, replication for concurrent read access in shared memory systems, using optimistic synchronization primitives to eliminate synchronization bottlenecks, using dynamic feedback to choose an appropriate lock coarsening policy, and replication in concurrent garbage collectors.

5.1 Reduction Analysis

Several existing compilers can recognize when a loop performs a reduction of many values into a single value [Ghuloum and Fisher 1995; Fisher and Ghuloum 1994; Pinter and Pinter 1991; Callahan 1991]. These compilers recognize when the reduction primitive (typically addition) is associative. They then exploit this algebraic property to eliminate the data dependence associated with the serial accumulation of values into the result. The generated program computes the reduction in parallel. Each processor has its own local replica of the variable used to hold the result; at the end of the parallel loop the partial contributions in the replicas are combined to produce the correct final result. Researchers have recently generalized the basic reduction

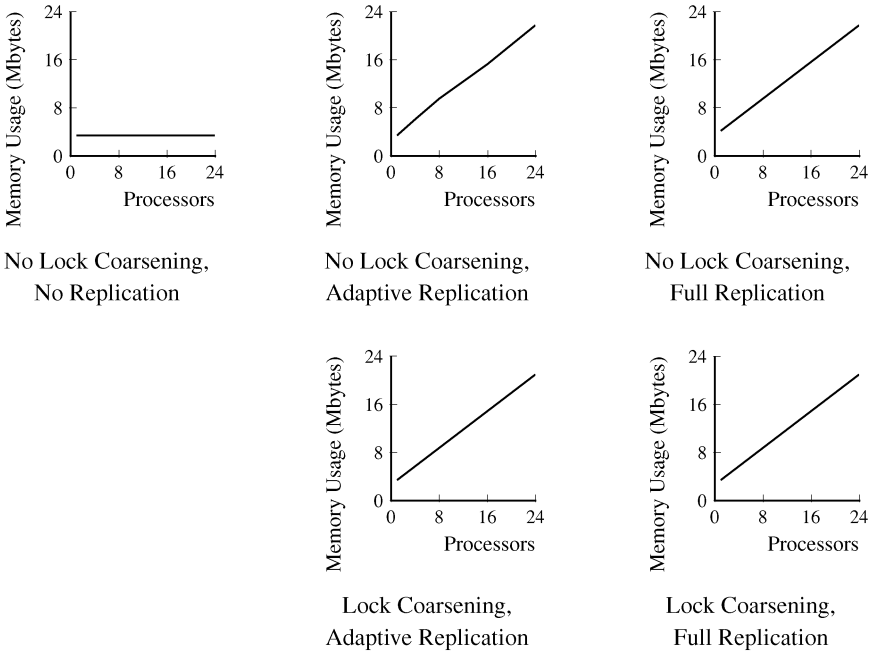


Fig. 21. Peak memory usage for String.

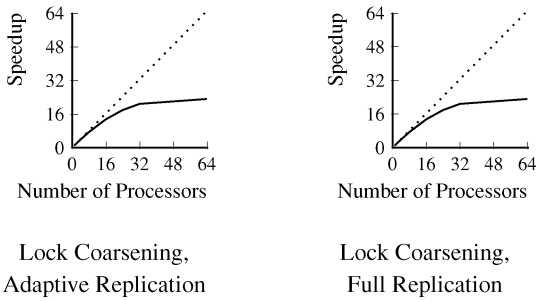


Fig. 22. Analytically Generated Speedups for String.

recognition algorithms to recognize when a loop performs a reduction of an array instead of a scalar. The reported results indicate that this optimization is crucial for obtaining good performance for the measured set of applications [Hall et al. 1995].

The research presented in this article applies a similar basic idea, but in the much less structured context of irregular object-based programs instead of regular loop nests in programs that access dense matrices using affine access functions. The generality of our target application set means that we must solve an additional set of problems that do not arise in the more restricted contexts that previous research in reduction analysis is designed to handle. In the next several paragraphs, we discuss how our adaptive replication algorithm solves these problems.

Object-based programs typically use references to access a statically unbounded number dynamically allocated objects. It is therefore difficult or impossible to statically identify and name the data that should be replicated to eliminate synchronization bottlenecks. Adaptive replication solves this problem by dynamically discovering the data that needs to be replicated to eliminate bottlenecks. The computations addressed in previous research access statically allocated variables and arrays. The compiler can therefore statically identify the variables or arrays which may cause synchronization bottlenecks, then statically allocate a replica for each processor.

In object-based programs, it may be possible to eliminate bottlenecks by replicating only a small subset of the updated data. In Water, for example, it is possible to eliminate the bottlenecks by replicating only a small portion of the updated objects. In Barnes-Hut, there is no bottleneck even without replication. To minimize memory usage, it is necessary to identify and replicate only those objects that would otherwise cause a bottleneck. The identification of these bottleneck objects is complicated by the fact that their identity may depend on the timing of the parallel computation, and therefore change from execution to execution. In traditional computations that access statically allocated arrays, a similar situation may occur if bottlenecks occur only at a small subset of the elements of a shared updated array. Nevertheless, previous research in this area did not attempt to address this issue at all, and simply replicated the entire array as a unit, a strategy that the researchers found acceptable for their set of applications.

Previous research in reduction analysis has demonstrated that accumulating partial contributions in replicated variables or arrays is often necessary to achieve good performance for loop nests in programs that access dense matrices using affine access functions. Our results show that accumulating partial contributions in replicated objects enables similar performance improvements in the more general context of parallel object-based programs; our techniques effectively solve the additional problems that arise in this more general context.

5.2 Replication in Shared Memory Systems

Many shared memory systems replicate data to enable concurrent read access [Lenoski 1992; Li 1986]. This optimization is clearly required to achieve any reasonable level of performance—in systems that do not implicitly replicate data for concurrent read access, programmers explicitly replicate the data [Lumetta et al. 1993]. Our hardware platform, the SGI Challenge XL multiprocessor, supports replication for concurrent read access via its cache coherence protocol.

One perspective on our research is that it replicates data to enable concurrent write access. Conceptually, each replica is a local proxy for the replicated object. The complications are that replication for concurrent write access may not always be legal (so our compiler must analyze the program to identify situations in which it is legal), it may not always improve performance (so our system only replicates objects that would otherwise cause synchronization bottlenecks), and there is a need to combine the partial contributions to generate

the correct final result (so our compiler generates code to perform the reduction at the end of the parallel phase). One reasonable way to view our research is that it generalizes the concept of replication for concurrent read access to provide, when legal and appropriate, replication for concurrent write access.

The coherence unit in most shared memory systems is larger than an individual memory word. Typical coherence units include cache lines in hardware shared memory systems and pages in software distributed shared memory systems [Lenoski 1992; Li 1986; Amza et al. 1996]. To avoid problems such as false sharing, these systems often include mechanisms such as relaxed memory consistency protocols that efficiently support concurrent writes to disjoint locations within the same coherence unit. Our research differs in that our goal is to enable concurrent writes to the same conceptual memory location, not concurrent writes to different memory locations within the same coherence unit.

5.3 Optimistic Synchronization Primitives

We have also explored another approach for eliminating synchronization bottlenecks: implementing atomic operations using optimistic synchronization primitives such as load linked and store conditional instead of mutual exclusion locks [Rinard 1999]. Our results show that optimistic synchronization can eliminate synchronization bottlenecks in applications (such as String) that use a single lock to synchronize concurrent updates to different instance variables of a large object. But optimistic synchronization is incapable of eliminating bottlenecks in applications (such as Water) that concurrently update the same instance variable. By contrast, the experimental results presented in this paper show that adaptive replication can eliminate bottlenecks in both kinds of applications. And it works well for applications (such as Barnes-Hut) that perform well without replication.

5.4 Dynamic Feedback

We have used dynamic feedback to find the best lock coarsening granularity [Diniz and Rinard 1999]. A compiler that uses dynamic feedback produces several different versions of the same source code; each version uses a different optimization policy. For lock coarsening, the different versions use different lock coarsening policies. The policies vary in how aggressively they apply the lock coarsening transformation. The generated code alternately performs sampling phases and production phases. Each sampling phase measures the overhead of each version in the current environment. For lock coarsening, the sources of overhead include *locking overhead*, which is time spent successfully acquiring and releasing locks, and *waiting overhead*, which is time spent at a lock acquire site waiting for another processor to release the lock. Each production phase uses the version with the least overhead in the previous sampling phase. The computation periodically resamples to adjust dynamically to changes in the environment.

Dynamic feedback is quite effective at finding the best lock coarsening policy. But because it does not replicate data, it cannot eliminate synchronization bottlenecks that are inherently present in the computation. The Water application

discussed in this paper, for example, has such a synchronization bottleneck. Adaptive replication, like dynamic feeback, dynamically adapts the replication policy to the run-time behavior of the application. It is also capable of replicating objects to eliminate synchronization bottlenecks, even if the bottlenecks are present in the initial computation and are not introduced by the lock coarsening transformation.

5.5 Concurrent Garbage Collection

Some concurrent copying garbage collectors replicate data in the sense that both versions of an object (the versions in to space and from space) can be accessed by the mutator and/or the collector. One approach is to allow the concurrently executing mutator to access both the original version of immutable data in from space and the replica in to space [Huelsbergen and Larus 1993]. This approach allows the mutator to access immutable data freely during collection without checking to verify that it has obtained the most up-to-date version of the object. Accesses to mutable data require such a check and are therefore less efficient than accesses to immutable data. Another approach is to use separate allocation mechanisms for mutable and immutable data [Doligez and Leroy 1993]. Mutable data is allocated in a single shared heap which uses a mark-and-sweep collector; immutable data is allocated in private heaps which use a copying collector. When immutable data allocated in the private heap becomes accessible via objects allocated in the shared heap, the collector copies the immutable data into the shared heap. Until the next collection, the versions in the two heaps are both accessible. Finally, it is also possible to require the mutator to access only from-space versions of objects while the collector concurrently replicates objects into to space [O'Toole and Nettles 1994]. During collection, the mutator generates a list of updates to mutable objects, which the collector must apply to the replicas in to space before flipping the spaces.

It is possible to apply replication analysis to augment concurrent copying collectors for computations that use associative and commutative operators with a zero to combine a set of contributions into a single location in mutable data. Instead of requiring all updates to be performed on the most up-to-date copy, the system would let the mutator update replicas in both from space and to space. When the collector finished traversing from space, it would combine the value in from space into the location in to space, restoring the correct value. To apply this transformation, the collector would have to initialize the to space value to zero and ensure that the garbage collector finished combining the values before the computation accessed the location in any way except to update the value with another contribution.

6. CONCLUSION

This paper presents results that illustrate a synergistic interaction between two program transformations: lock coarsening and adaptive replication. Adaptive replication eliminates synchronization bottlenecks by replicating the frequently updated objects that cause the bottlenecks. Lock coarsening reduces synchronization overhead by reducing the frequency with which the

computation acquires and releases locks. But because it increases the size of the critical sections, it may introduce synchronization bottlenecks.

We have implemented a compiler that automatically applies lock coarsening and adaptive replication. We used the compiler to generate different versions of several benchmark programs; these versions use different combinations of the lock coarsening and replication transformations. Our results show that the combination of lock coarsening and adaptive replication can eliminate synchronization bottlenecks and significantly reduce the synchronization and replication overhead as compared to versions that use none or only one of the transformations.

ACKNOWLEDGMENTS

We would like to the anonymous referees of various versions of this article for their thoughtful and helpful comments.

REFERENCES

ALDRICH, J., CHAMBERS, C., SIRER, E., AND EGGERS, S. 1999. Static analyses for eliminating unnecessary synchronization from Java programs. In *Proceedings of the 6th International Static Analysis Symposium*.

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 Comput. 29*, 2 (June), 18–28.

ARNOLD, K. AND GOSLING, J. 1996. *The Java Programming Language*. Addison-Wesley, Reading, Mass.

BARNES, J. AND HUT, P. 1986. A hierarchical O(NlogN) force calculation algorithm. *Nature 324*, 4 (Dec.), 446–449.

BERRY, M., CHEN, D., KOSS, P., KUCK, D., LO, S., PANG, Y., POINTER, L., ROLOFF, R., SAMEH, A., CLEMENTI, E., CHIN, S., SCHNEIDER, D., FOX, G., MESSINA, P., WALKER, D., HSIUNG, C., SCHWARZMEIER, J., LUE, K., ORSZAG, S., SEIDL, F., JOHNSON, O., GOODRUM, R., AND MARTIN, J. 1989. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. ICASE Report 827, Center for Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, Urbana, IL. May.

BRINCH-HANSEN, P. 1972. Structured multiprogramming. *Commun. ACM 15*, 7 (July), 574–578.

BRINCH-HANSEN, P. 1975. The programming language Concurrent Pascal. *IEEE Trans. Softw. Eng. SE-1*, 2 (June), 199–207.

CALLAHAN, D. 1991. Recognizing and parallelizing bounded recurrences. In *Proceedings of the 4th Workshop on Languages and Compilers for Parallel Computing* (Santa Clara, Calif.). Springer-Verlag, New York, 169–184.

CHASE, D., WEGMAN, M., AND ZADEK, F. 1990. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation* (White Plains, N.Y.) ACM, New York, 296–310.

DINIZ, P. AND RINARD, M. 1998. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *J. Parall. Distrib. Comput. 49*, 2 (Mar.), 2218–244.

DINIZ, P. AND RINARD, M. 1999. Eliminating synchronization overhead in automatically parallelized programs using dynamic feedback. *ACM Trans. Comput. Syst. 17*, 2 (May), 89–132.

DOLBY, J. 1997. Automatic inline allocation of objects. In *Proceedings of the SIGPLAN '97 Conference on Program Language Design and Implementation* (Las Vegas, Nev.). ACM, New York.

DOLIGEZ, D. AND LEROY, X. 1993. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proceedings of the 20th Annual ACM Symposium on the Principles of Programming Languages*. ACM, New York.

EMAMI, M., GHIYA, R., AND HENDREN, L. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation* (Orlando, Fl.) ACM, New York, 242–256.

FISHER, A. AND GHULOUM, A. 1994. Parallelizing complex scans and reductions. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation* (Orlando, Fla.) ACM, New York, 135–144.

GHIYA, R. AND HENDREN, L. 1996. Is it a tree, a DAG or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*. ACM, New York, 1–15.

GHULOUM, A. AND FISHER, A. 1995. Flattening and parallelizing irregular, recurrent loop nests. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Santa Barbara, Calif.). ACM, New York, 58–67.

GRAHAM, S., KESSLER, P., AND MCKUSICK, M. 1982. GPROF: A call graph execution profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction* (Boston, Mass.). ACM, New York.

GRAY, J. AND REUTER, A. 1993. *Transaction Processing: Concepts and Techniques*. Morgan-Kaufmann, San Francisco, CA.

HALL, M., AMARASINGHE, S., MURPHY, B., LIAO, S., AND LAM, M. 1995. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95* (San Diego, Calif.). IEEE Computer Society Press, Los Alamitos, Calif.

HARRIS, J., LAZARATOS, S., AND MICHELENA, R. 1990. Tomographic string inversion. In *Proceedings of the 60th Annual International Meeting, Society of Exploration and Geophysics, Extended Abstracts*. 82–85.

HEINRICH, J. 1993. *MIPS R4000 Microprocessor User's Manual*. Prentice-Hall, Englewood Cliffs, N.J.

HOARE, C. A. R. 1974. Monitors: An operating system concept. *Commun. ACM 17*, 10 (Oct.), 549–557.

HUELSBERGEN, L. AND LARUS, J. 1993. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, Calif.). ACM, New York.

KING, J. 1976. Symbolic execution and program testing. *Commun. ACM 19*, 7 (July), 385–394.

KNUTH, D. 1971. An empirical study of FORTRAN programs. *Softw.—Pract. Exper. 1*, 105–133.

LAMPSON, B. W. AND REDELL, D. D. 1980. Experience with processes and monitors in Mesa. *Commun. ACM 23*, 2 (Feb.), 105–117.

LENOSKI, D. 1992. The design and analysis of DASH: A scalable directory-based multiprocessor. Ph.D. thesis, Dept. of Electrical Engineering, Stanford Univ., Stanford, Calif.

LI, K. 1986. Shared virtual memory on loosely coupled multiprocessors. Ph.D. dissertation, Dept. of Computer Science, Yale Univ., New Haven, Conn.

LUMETTA, S., MURPHY, L., LI, X., CULLER, D., AND KHALIL, I. 1993. Decentralized optimal power pricing: The development of a parallel program. *IEEE Parall. Distrib. Tech. 1*, 4 (Nov.), 23–31.

MOHR, E., KRANZ, D., AND HALSTEAD, R. 1990. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*. ACM, New York, 185–197.

O'TOOLE, J. AND NETTLES, S. 1994. Concurrent replicating garbage collection. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming* (Orlando, Fla.). ACM, New York.

PINTER, S. AND PINTER, R. 1991. Program optimization and parallelization using idioms. In *Proceedings of the 18th Annual ACM Symposium on the Principles of Programming Languages* (Orlando, Fla.). ACM, New York, 79–92.

POLYCHRONOPOULOS, C. AND KUCK, D. 1987. Guided self-scheduling: A practical scheduling scheme for parallel computers. *IEEE Trans. Comput. 36*, 12 (Dec.), 1425–1439.

RINARD, M. 1999. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. *ACM Trans. Comput. Syst. 17*, 4 (Nov.), 337–371.

RINARD, M. AND DINIZ, P. 1997. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Trans. Prog. Lang. Syst. 19*, 6 (Nov.), 941–992.

RINARD, M. AND DINIZ, P. 1999. Eliminating synchronization bottlenecks in object-based programs using adaptive replication. In *Proceedings of the 1999 ACM International Conference on Supercomputing* (Rhodes, Greece). ACM, New York.

RINARD, M. AND LAM, M. 1998. The design, implementation, and evaluation of jade. *ACM Trans. Prog. Lang. Syst. 20*, 3 (May), 483–545.

ROVNER, P. 1986. Extending modula-2 to build large, integrated systems. *IEEE Softw. 3*, 6 (Nov.), 46–57.

RUGINA, R. AND RINARD, M. 1999. Pointer analysis for multithreaded programs. In *Proceedings of the SIGPLAN '99 Conference on Program Language Design and Implementation* (Atlanta, Ga.). ACM, New York.

SAGIV, M., REPS, T., AND WILHELM, R. 1998. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Prog. Lang. Syst. 20*, 1 (Jan.), 1–50.

SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. 1990. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput. 39*, 9 (Sept.), 1175–1185.

SINGH, J. 1993. Parallel hierarchical N-body methods and their implications for multiprocessors. Ph.D. dissertation, Dept. of Electrical Engineering, Stanford Univ., Stanford, Calif.

SINGH, J., WEBER, W., AND GUPTA, A. 1992. SPLASH: Stanford parallel applications for shared memory. *Comput. Arch. News 20*, 1 (Mar.), 5–44.

WILSON, R. AND LAM, M. 1995. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation* (La Jolla, Calif.). ACM, New York.

WOO, S., OHARA, M., TORRIE, E., SINGH, J., AND GUPTA, A. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*. ACM, New York.

Received June 2000; revised September 2001 and March 2002; accepted December 2002