Eliminating Synchronization Bottlenecks in Object-Based Programs Using Adaptive Replication

Martin Rinard
Massachusetts Institute of Technology
Laboratory for Computer Science
rinard@lcs.mit.edu
Pedro Diniz
University of Southern California
Information Sciences Institute
pedro@isi.edu

Abstract

This paper presents a 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 measuring the amount of contention at each object to detect 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. Our experimental results show that for our set of benchmark programs, adaptive replication can improve the overall performance by up to a factor of three over versions that use no replication, and can reduce the memory consumption by up to a factor of four over versions that fully replicate updated objects. Furthermore, adaptive replication never significantly harms the performance and never increases the memory usage without a corresponding increase in the performance.

1 Introduction

The characteristics of future computational environments ensure that multithreaded software will play an increasingly important role in many areas of computer science. Multithreading is a key structuring technique for programs that manipulate information in modern distributed computing environments. Important examples of multithreaded software include sophisticated user interface systems [13, 4, 26] and responsive multithreaded servers [23]. Programmers also use multithreaded software to exploit the capabilities of small-scale shared-memory multiprocessors, a major source of computational power for scientific, engineering and information-intensive computing.

Unfortunately, problems such as nondeterministic behavior and deadlock complicate the development of multithreaded software. Programmers have responded to these problems by adopting a programming methodology (inspired by concepts from object-oriented programming) in which each thread is structured as a sequence of atomic operations on objects. The result is a more structured, tractable programming model. The advantages of this model have led to its adoption in programming languages such as Mesa and Java [18, 1].

The research presented in this paper 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 [18, 29].

In many programs it is possible to eliminate synchronization bottlenecks by replicating frequently accessed objects that cause synchronization 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.

We have implemented adaptive replication in the context of a parallelizing compiler for object-based languages [28]. Given a serial program written in a subset of C++, the compiler automatically generates parallel code that uses adaptive replication to efficiently eliminate synchronization bottlenecks. This paper presents experimental results that characterize the impact of adaptive replication 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, adaptive replication can improve the overall performance by up to a factor of three over versions that use no replication, and can reduce the memory consumption by up to a factor of four over versions that fully replicate updated objects. Furthermore, adaptive replication never significantly harms the performance and never increases the memory usage without a corresponding increase in the performance.

This paper makes the following contributions:

- It presents a static program analysis algorithm that determines when it is legal to replicate objects to eliminate synchronization bottlenecks.
- It presents a program transformation algorithm that enables the generated code to correctly replicate objects.
- It presents a adaptive replication algorithm, which dynamically adapts to the access pattern of each execution of the program to choose which objects to replicate.
- It presents experimental results that characterize the performance impact of adaptive replication on three benchmark applications. These results show that adaptive replication can significantly improve the overall performance while avoiding unnecessary replication.

The remainder of the paper is structured as follows. Section 2 presents an example that illustrates the issues associated with adaptive replication. Section 3 presents the replication analysis and code generation algorithms. Section 4 presents experimental results that characterize the impact of 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 adaptive replication. The program in Figure 1 implements a serial 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 in parallel. Our compiler is able to use commutativity analysis to statically detect this source of concurrency [28]. Because the data structure may be a graph, the parallel traversal may visit the same node multiple times. The generated code must therefore contain synchronization constructs that make each `visit` operation execute atomically with respect to all other operations that access the same object.

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

Figure 1: Serial Graph Traversal

Figure 2 presents the code that the compiler generates when it does not use adaptive replication. The compiler augments each `node` object with a mutual exclusion lock **mutex**. The automatically generated `parallel_visit` operation, which performs the parallel traversal, uses this lock to ensure that it executes atomically. It acquires the lock before it updates the `sum` instance variable, then releases the lock after the update. The lock synchronization makes multiple updates to the same object execute sequentially.

The transitions from serial to parallel execution and from parallel back to serial execution take place inside the `visit` operation. This operation first invokes the `parallel_visit` operation, then invokes the **wait** construct, which blocks until all parallel tasks created by the current task or its descendant tasks finish. The `parallel_visit` 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 [22].

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

Figure 2: Parallel Traversal Without Adaptive Replication

A problem with the code in Figure 2 is that it may suffer from synchronization bottlenecks if many of the nodes in the graph all

point to the same node. Figure 3 presents such a graph.[1] Because all of the updates use an operator (the + operator)[2] that is associative, commutative, and has a zero, it is possible to eliminate the synchronization bottlenecks by having each thread allocate its own local replica of each node that caused a bottleneck. Each thread would then update its own replica without synchronization and without interacting with other threads. At the end of the parallel phase, the program would compute the total sum of the partial sums stored in the replicas, and store this total sum back into the original object. It would then deallocate the replicas. Figure 4 presents a high level version of the code that the compiler generates when it uses this approach.

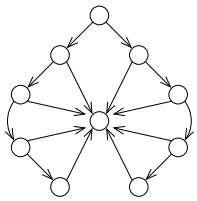

Figure 3: Example Graph

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 a construct, the **try_acquire** construct, that attempts to acquire a lock. The construct returns false if it is unable to immediately acquire the lock (this happens if another thread already holds the lock), and true if it did acquire the lock. If a thread is unable to acquire an object's lock, it assumes that the object may cause a synchronization bottleneck. It obtains a local replica of the object and performs the update locally on the replica. The replication policy therefore adapts to the dynamic characteristics of each program execution — the program uses the **try_acquire** construct to dynamically detect and replicate only those objects that would otherwise cause synchronization bottlenecks.

- **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 4 eliminates this check.

- **Retrieving Replicas:** Each thread 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

[1] This specific example is intended primarily for the purpose of presenting the technique. Section 4 describes several applications whose performance benefits from the use of adaptive replication.

[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 1.

```
class node {
  private:
    lock mutex;
    int value, sum;
    node *left, *right;
  public:
    void visit(int);
    void parallel_visit(int);
    void replica_visit(int);
    node *replicate();
    friend void combine_node_replicas();
};
void node::visit(int p) {
  this->parallel_visit(p);
  wait();
  combine_node_replicas();
}
void node::parallel_visit(int p) {
  node *replica = lookup(this);
  if (replica == NULL) {
    if (mutex.try_acquire()) {
      sum = sum + p;
      mutex.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 node::replica_visit(int p) {
  sum = sum + p;
  if (left != NULL)
    spawn(left->parallel_visit(value));
  if (right != NULL)
    spawn(right->parallel_visit(value));
}
node *node::replicate() {
  node *replica = new node;
  replica->sum = 0;
  replica->value = value;
  replica->left = left;
  replica->right = right;
  insert(this, replica);
  return(replica);
}
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;
    }
  }
}
```

Figure 4: Parallel Traversal With Adaptive Replication

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.

- **Updating Replicas:** All updates to replicas are performed by synchronization-free versions of operations that the compiler generates for that purpose. In the example in Figure 4, the `replica_visit` operation performs all updates to replicated graph nodes.

- **Combining Values:** At the end of the parallel phase, the generated code traverses the hash tables (recall that there is one hash table per thread) 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 in parallel. For clarity, the code in Figure 4 performs the traversals serially.

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. Our current compiler generates code that first checks for an existing replica. If the replica exists, the update is performed on the replica. If the replica does not exist, it attempts to acquire the lock in the object. It creates a replica only if it is unable to acquire the lock.

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 lock primitives, the resulting attempts to acquire the object's lock will often generate expensive accesses to remote data, *even if the lock acquisition attempts do not succeed.* [3] On current architectures, we expect such lock 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 will 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 generated code that first attempted to acquire the lock, then checked for a local replica suffered from these two effects.

The potential drawback of checking for an existing replica first is the overhead associated with unsuccesful hash table lookups, especially for applications with few objects that multiple processors attempt to update. As noted above, we expect such lookups to execute quickly. And our experimental results show that, for the programs in our benchmark set, unsuccessful cache lookups have a negligible effect on the overall performance.

[3] 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 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.

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, 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 [28]. Commutativity analysis is capable of paralleliz-ing 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. Replication analysis is therefore designed to handle a general class of parallel computations that perform atomic operations on objects and consist of an alternating sequence of parallel and serial phases.

This section presents the replication analysis and code generation algorithms, starting with the model of computation for programs that the replication analysis algorithm is designed to analyze and the program representation that the algorithms use.

3.1 Model of Computation

The replication analysis algorithm is designed to analyze pure 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 pointer 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 a receiver object and several parameters. When an operation executes, it can read and write the instance variables of the receiver object, access the parameters, or invoke other operations. Well structured object-based programs conform to this model of computation; pure object-based languages such as Smalltalk enforce it explicitly [9].

3.2 Program Representation

The commutativity analysis algorithm extracts some information that the replication analysis and code generation algorithms use. The algorithm executes on one phase of the program at a time. The phase selection is driven by an algorithm in the compiler that traverses the static call graph of the program to find subgraphs whose execution it can parallelize. Each such subgraph corresponds to a parallel phase of the program.

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 [28]. It determines each of these sets by traversing the call subgraph of the phase. The set of operations is simply the set of operations in the call subgraph; 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 [16, 28]. Appendix A contains a description of the exact format of update and invocation expressions.

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. 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 `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 serially 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 receiver, 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 5 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 generalizes in a straightforward way to handle computations that contain multiple such operators. The version implemented in our prototype compiler can apply adaptive replication to computations that contain multiple commutative, associative operators with a zero.

The algorithm takes as parameters the set of invoked operations, 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 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 non-replicatable operation on a replica. If the operation passes both of these tests, it is classified as a replicatable operation.

3.5 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 parallel version and the replicated version.

The parallel version executes on original objects, not replicas. If it may update the receiver, it first checks to see if a replica has already been created. If so, it invokes the replicated version to perform the update on the replica. If not, it uses a **try_acquire** construct to attempt to acquire the lock in the receiver. If the lock is acquired, the parallel version performs the updates as in the original program, then releases the lock and completes its execution by invoking the parallel version of each operation that it should invoke. If the lock is not acquired, the parallel 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 parallel 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 parallel version invokes the replicated version to perform the update on the replica.

The replicated version executes on replicas, not original objects. It performs all of its updates to the replica without synchronization. It invokes the parallel version of each operation that it should invoke unless the receiver of the operation is `this` (the object that the replicated version executes on). If the receiver is `this`, it invokes the replicated version of the invoked operation. This invocation strategy reduces the overhead by eliminating the trip through the parallel version for sequences of operations on the same replicated object.

The code generation algorithm also produces a replica creation operation. The receiver of this operation is the original object. The 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

[4] The restrictions on replicatable variables and replicatable operations ensure that pointers 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 as the receiver expression at the operation invocation site. Any other expression used to identify the receiver 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).
         receiver(i) = this 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 = exp1; i < exp2; i+ = exp3) upd
      and ∀1≤j≤3 variables(expj) ∩ 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 = exp1; i < exp2; i+ = exp3) upd)
    return updatedVariable(upd);
```

Figure 5: Replication Analysis Algorithm

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 parallel version waits until it acquires the lock on the original object and performs the operation on that object. In the example in Figure 4, the `replicate` operation is the replica creation operation. For clarity, we eliminate 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.

3.6 Interaction With Lock Coarsening

Our compiler contains an analysis algorithm and a sequence of transformations that increase the granularity at which the computation locks objects [6, 25]. The default granularity is for each operation that updates a lock to acquire and release the lock in the updated object. The analysis algorithm finds sequences of operations that acquire and release the same lock. The transformed program acquires the lock once, performs the operations without additional synchronization, then releases the lock. The advantage is a reduction in the lock overhead (the amount of time spent acquiring and releasing locks). The disadvantage is a potentially crippling decrease in the amount of available concurrency. To achieve good performance for a range of applications, we have developed sophisticated dynamic feedback algorithms that adjust the amount of coarsening to the dynamic characteristics of the computation [5].

There is a synergistic interaction between lock coarsening and adaptive replication. Consider a maximally lock coarsened program. If all of the operations that execute between a lock acquire and release pair are replicatable operations, the compiler can apply adaptive replication at the coarsened granularity. The compiler replaces the lock acquire site with a code sequence that checks if a local replica already exists. If so, the generated code executes the corresponding sequence of replicated operations on the replica. If not, it uses a **try_acquire** construct to attempt to acquire the lock in the original object. If the attempt succeeds, the program executes the sequence of operations on the original object. If the attempt fails, it creates a replica (subject, of course, to the memory consumption limits) and executes the sequence of replicated operations on the replica.

The resulting program combines the advantages of 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 serialization that the lock coarsening transformation may have introduced. Our current compiler first applies the maximal 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.

4 Experimental Results

We next present experimental results that characterize the performance and memory impact of using adaptive replication. We present results for three automatically parallelized applications: Water [30], which simulates water molecules in the liquid state, Barnes-Hut [2], a hierarchical N-body solver, and String [12], 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 serial C++ code, Barnes-Hut consists of approximately 1500 lines of serial C++ code, and String consists of approximately 2050 lines of serial C++ code.

4.1 Methodology

We implemented a prototype parallelizing compiler that uses commutativity analysis as its basic analysis paradigm. The compiler includes an analysis algorithm that determines when it is legal to replicate updated objects. Flags determine the replication policy that the generated code uses. We used the compiler to obtain the following versions of each application:

- **No Replication:** There is no replication of updated objects. There is one lock per object; the generated code acquires the lock before it performs any updates to the object. For each application we use the lock coarsening policy (see Section 3.6) that performs best for that application. For String and Water there is no lock coarsening. For Barnes-Hut the generated code uses the maximal lock coarsening policy.
- **Adaptive Replication:** The generated code uses adaptive replication applied to the version with the maximal lock coarsening policy.
- **Full Replication:** Whenever possible, the generated code performs updates on replicas. For our set of applications, all of the operations in parallel phases are replicatable operations. All of the updates are therefore performed on replicas.

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 [14]. The release simply clears the value. In practice, we expect programs compiled for multiprogrammed machines to use less efficient primitives that invoke the operating system to suspend a thread if it is unable to acquire a lock [15]. The presented results therefore overstate the efficiency of the versions that use locks.

4.2 Cost Of Basic Operations

We measured the cost of the lock primitives by creating a benchmark program to measure the time required to perform a single update with and without lock synchronization. Each update reads an array element, adds a constant to the element, then stores the new value back into the array. Table 1 presents the execution times. We present times for cached updates, in which all of the accessed data are present in the first level processor cache, and for uncached updates, in which all of the data are present in the first level processor cache of another processor. The execution times of the uncached versions are dominated by the cache miss time and are therefore significantly larger than for the cached versions.

4.3 Water

Table 2 presents the execution times for Water. Figure 6 presents the corresponding speedup curves. The speedup curves plot the running time of the original serial 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 serial version is a standard sequential C++ program that executes with no parallelization or synchronization overhead.

Version	Execution Time For One Cached Update (microseconds)	Execution Time For One Uncached Update (microseconds)
No Synchronization	0.049	3.251
Lock Synchronization	0.278	3.456

Table 1: Measured Execution Times for One Update

Version	Processors				
	1	4	8	16	24
Serial	164.26	-	-	-	-
No Replication	174.65	52.43	29.80	25.62	29.25
Adaptive Replication	169.27	46.99	24.75	13.08	9.54
Full Replication	166.57	45.05	22.97	12.40	8.98

Table 2: Execution Times for Water (seconds)

The performance of both the Adaptive Replication and Full Replication versions scales well with the number of processors. The performance of the No Replication version, on the other hand, fails to scale beyond eight processors.

We used program counter sampling [10, 17] 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 at the end of parallel phases.
- **Synchronization:** Time spent acquiring and releasing locks, including time spent waiting to acquire locks held by other threads. Synchronization bottlenecks show up as large amount of time spent in this component.
- **Idle:** Time spent idle. All but one processor is idle during serial 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 serial phases accounts for the vast majority of the total idle time.
- **Compute:** Time spent performing useful computation from the application.

Figure 7 presents the time breakdowns for Water.[6] These breakdowns clearly show that the No Replication version suffers from a synchronization bottleneck, and that the bottleneck becomes more severe as the number of processors increases. The time breakdowns also show that replication completely eliminates the synchronization bottleneck. The versions that use replication have negligible synchronization overhead.

[6] 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.

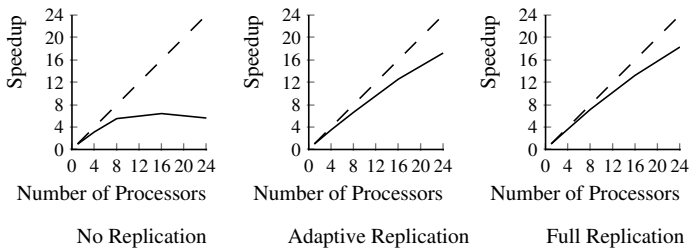

Figure 6: Speedups for Water

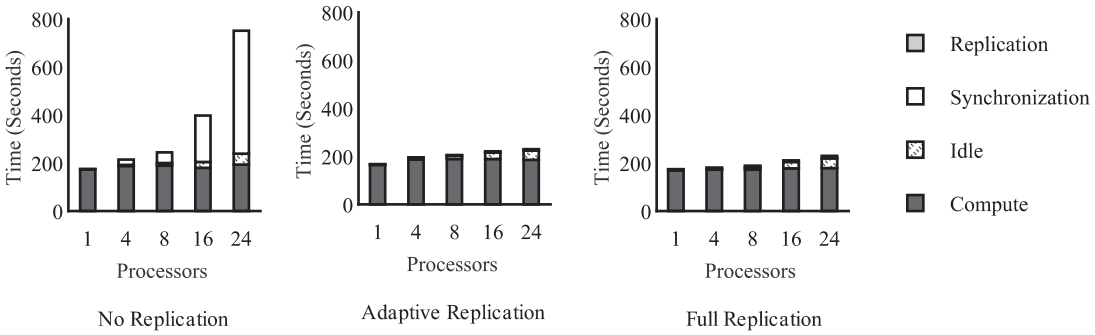

Figure 7: Time Breakdowns for Water

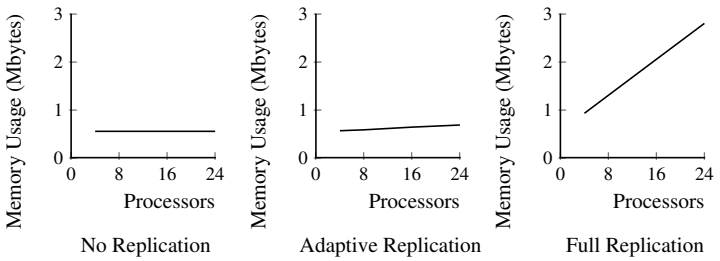

Figure 8: Peak Memory Usage for Water

Figure 8 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 No Replication version does not vary with the number of processors. The peak memory usage increases slightly with the number of processors for the Adaptive Replication version. For the Full Replication version, the peak memory usage increases significantly with the number of processors.

Adaptive replication works well for Water. It eliminates the synchronization bottlenecks that degrade the performance of the No Replication version. Its peak memory usage is also significantly less than the Full Replication version, which indicates that it is possible to eliminate the synchronization bottlenecks by replicating only a small amount of data.

4.4 Barnes-Hut

Table 3 presents the execution times for Barnes-Hut; Figure 9 presents the corresponding speedup curves. All applications exhibit close to identical performance. The time breakdowns in Figure 10 illustrate why: there are no synchronization bottlenecks in the No Replication version and very little replication overhead in the Adaptive Replication and Full Replication versions. As the number of processors increases, the primary limiting factor on the performance is the idle time. One of the phases of the computation (the tree construction phase) executes sequentially. As the number of processors increases, this serial phase becomes a bottleneck.

Figure 11 shows that the Adaptive Replication version uses the same amount of memory as the No Replication version — for this application, the Adaptive Replication version never replicates an object.

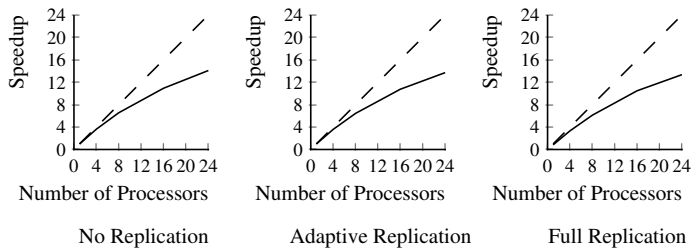

Figure 9: Speedups for Barnes-Hut

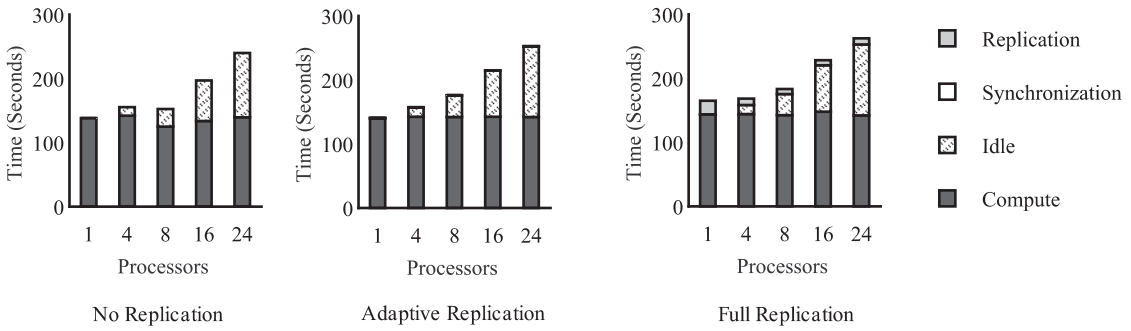

Figure 10: Time Breakdowns for Barnes-Hut

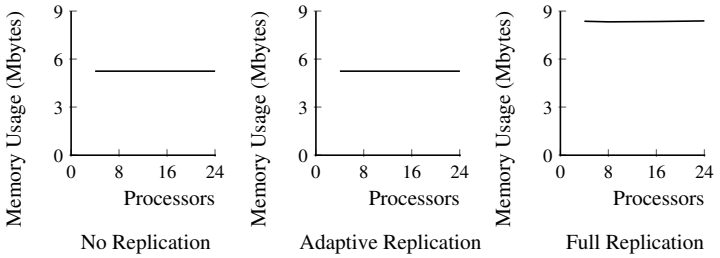

Figure 11: Peak Memory Usage for Barnes-Hut

Version	Processors				
	1	4	8	16	24
Serial	136.33	-	-	-	-
No Replication	139.78	37.63	20.79	12.47	9.67
Adaptive Replication	139.85	38.02	21.16	12.67	9.94
Full Replication	164.28	41.02	22.26	13.00	10.21

Table 3: Execution Times for Barnes-Hut (seconds)

4.5 String

Table 4 presents the execution times for String. Figure 12 presents the corresponding speedup curves. The performance for the No Replication version peaks at 16 processors, then rapidly falls off. The Adaptive Replication and Full Replication versions, on the other hand, perform well for all numbers of processors. The time breakdowns in Figure 13 show that the No Replication version suffers from a serious synchronization bottleneck at 24 processors. Replication completely eliminates this bottleneck, at the cost of a modest amount of replication overhead. The end result is that, with replication, the computation performs very well for all numbers of processors. The peak memory usage graphs in Figure 14 show that both the Adaptive Replication and Full Replication versions significantly increase the memory usage — both versions completely replicate a large object. The overall memory usage is still acceptable, however.

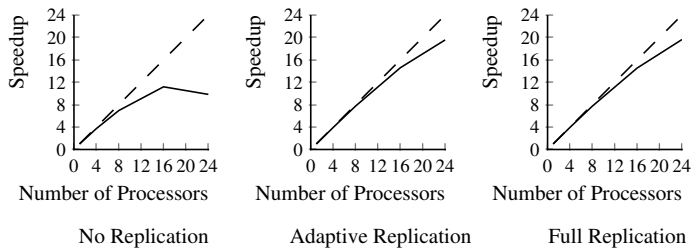

Figure 12: Speedups for String

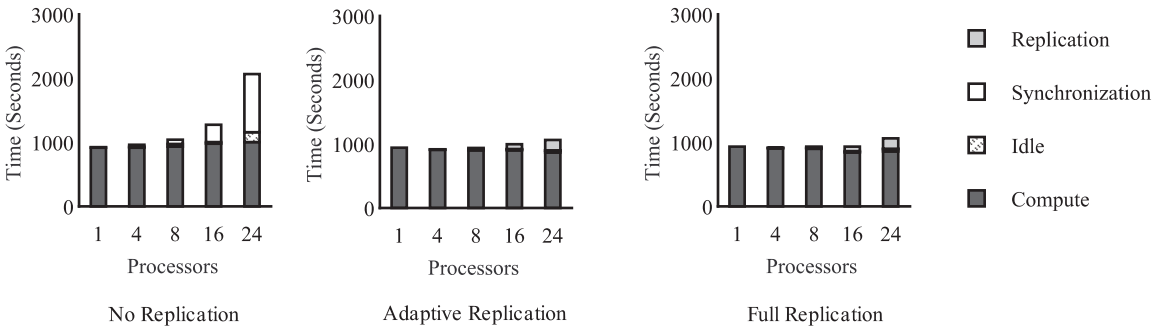

Figure 13: Time Breakdowns for String

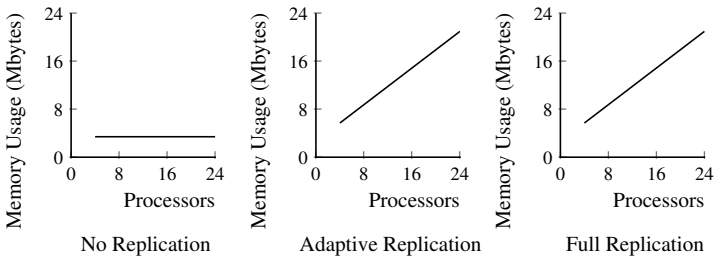

Figure 14: Peak Memory Usage for String

Version	Processors				
	1	4	8	16	24
Serial	881.27	-	-	-	-
No Replication	896.99	236.36	126.86	78.76	89.60
Adaptive Replication	892.70	224.45	113.78	60.38	45.06
Full Replication	897.98	223.73	113.89	60.61	44.91

Table 4: Execution Times for String (seconds)

4.6 Discussion

The results show that Adaptive Replication generates good performance for all of our benchmark applications. It effectively eliminates the synchronization bottlenecks that degrade the performance of Water and String, it imposes little performance overhead, and it minimizes the memory consumption by replicating only those objects that would otherwise cause synchronization bottlenecks. Furthermore, it is a robust technique suitable for inclusion in a compiler — as Barnes-Hut illustrates, it imposes very little performance overhead and no memory overhead if the application does not require replication for good performance.

We have also explored another approach for eliminating synchronization bottlenecks: implementing atomic operations using optimistic synchronization primitives such as load linked/store conditional instead of mutual exclusion locks [27]. Our results show 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 Related Work

In this section we discuss related work in the area of reduction analysis and replication for concurrent read access in shared memory systems.

5.1 Reduction Analysis

Several existing compilers can recognize when a loop performs a reduction of many values into a single value [8, 7, 24, 3]. 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 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 [11].

The research presented in this paper 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 previous research, the restricted model of computation allows the compiler to statically recognize which variable or array must be replicated — the dynamic trade off between additional concurrency and replication overhead simply does not arise. But in object-based programs, it may be necessary to replicate only a subset of the objects — only those objects that would cause synchronization bottlenecks. Adaptive replication solves this new problem by dynamically measuring the contention at each object. It replicates an object only if there is significant contention for the object.

In previous research, the compiler only has to replicate a statically allocated variable or array. The code generation algorithm can therefore statically create an array of replicas, with each processor using a different element of the array. But in object-based programs, objects are allocated dynamically and accessed by navigating through references stored in other objects. It may therefore be necessary to replicate an unbounded number of dynamically allocated objects for which there are no statically available names. Adaptive replication solves this problem by dynamically creating replicas in response to contention and storing the replicas in a hash table.

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 [19, 20]. 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 [21]. 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.

6 Conclusion

Synchronization bottlenecks can significantly degrade the performance of multithreaded programs that perform atomic operations on objects. Our experimental results show that adaptive replication can effectively eliminate synchronization bottlenecks while avoiding unnecessary memory consumption.

To take a broader perspective, we view adaptive replication as part of a general trend towards adaptive computing. In the future we expect the complexity and heterogeneity of systems to increase, and we expect that this increase will be matched by an increase in the analysis and transformational capabilities of compilers. In such computing environments it will become increasingly important to develop flexible techniques that adapt to the dynamic characteristics of different program executions in a range of execution environments. Our experimental results clearly demonstrate the advantages of an adaptive approach to replication. In the future we expect researchers to develop effective adaptive techniques for a wide range of other problems.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, Mass., 1996.

[2] J. Barnes and P. Hut. A hierarchical O(NlogN) force calculation algorithm. *Nature*, 324(4):446–449, December 1986.

[3] D. Callahan. Recognizing and parallelizing bounded recurrences. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, pages 169–184, Santa Clara, CA, August 1991. Springer-Verlag.

[4] L. Cardelli and R. Pike. Squeak: a language for communicating with mice. In *Proceedings of SIGGRAPH '85*, San Francisco, CA, July 1985.

[5] P. Diniz and M. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Proceedings of the SIGPLAN '97 Conference on Program Language Design and Implementation*, Las Vegas, NV, June 1997.

[6] P. Diniz and M. Rinard. Synchronization transformations for parallel computing. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, pages 187–200, Paris, France, January 1997. ACM, New York.

[7] A. Fisher and A. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*, pages 135–144, Orlando, FL, June 1994. ACM, New York.

[8] A. Ghuloum and A. Fisher. Flattening and parallelizing irregular, recurrent loop nests. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 58–67, Santa Barbara, CA, July 1995. ACM, New York.

[9] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley, Reading, Mass., 1983.

[10] S. Graham, P. Kessler, and M. McKusick. gprof: a call graph execution profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, Boston, MA, June 1982. ACM, New York.

[11] M.W. Hall, S.P. Amarasinghe, B.R. Murphy, S. Liao, and M.S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press, Los Alamitos, Calif.

[12] J. Harris, S. Lazaratos, and R. Michelena. Tomographic string inversion. In *Proceedings of the 60th Annual International Meeting, Society of Exploration and Geophysics, Extended Abstracts*, pages 82–85, 1990.

[13] C. Hauser, C. Jacobi, M. Theimer, B. Welch, and M. Weiser. Using threads in interactive systems: A case study. In *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, Asheville, NC, December 1993.

[14] J. Heinrich. *MIPS R4000 Microprocessor User's Manual*. Prentice-Hall, Englewood Cliffs, N.J., 1993.

[15] A. Karlin, K. Li, M. Manasse, and S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, Pacific Grove, CA, October 1991.

[16] J. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[17] D. Knuth. An empirical study of FORTRAN programs. *Software—Practice and Experience*, 1:105–133, 1971.

[18] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[19] D. Lenoski. *The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor*. PhD thesis, Dept. of Electrical Engineering, Stanford Univ., Stanford, Calif., February 1992.

[20] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Dept. of Computer Science, Yale Univ., New Haven, Conn., September 1986.

[21] S. Lumetta, L. Murphy, X. Li, D. Culler, and I. Khalil. Decentralized optimal power pricing: the development of a parallel program. *IEEE Parallel and Distributed Technology*, 1(4):23–31, November 1993.

[22] E. Mohr, D. Kranz, and R. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 185–197. ACM, New York, June 1990.

[23] T. Pham and P. Garg. *Multithreaded Programming with Windows NT*. Prentice-Hall, Englewood Cliffs, N.J., 1995.

[24] S. Pinter and R. Pinter. Program optimization and parallelization using idioms. In *Proceedings of the 18th Annual ACM Symposium on the Principles of Programming Languages*, pages 79–92, Orlando, FL, January 1991. ACM, New York.

[25] J. Plevyak, X. Zhang, and A. Chien. Obtaining sequential efficiency for concurrent object-oriented languages. In *Proceedings of the 22nd Annual ACM Symposium on the Principles of Programming Languages*. ACM, January 1995.

[26] J. Reppy. *Higher–order Concurrency*. PhD thesis, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., June 1992.

[27] M. Rinard. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 112–123, Las Vegas, NV, June 1997. ACM, New York.

[28] M. Rinard and P. Diniz. Commutativity analysis: A new framework for parallelizing compilers. In *Proceedings of the SIGPLAN '96 Conference on Program Language Design and Implementation*, pages 54–67, Philadelphia, PA, May 1996. ACM, New York.

[29] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[30] J. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Comput. Arch. News*, 20(1):5–44, March 1992.

A Update Expressions

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`. An update expression `v[exp']=exp` represents an update to the array instance variable `v`. An update expression of the form **for** (`i=exp`$_1$; `i<exp`$_2$; `i+=exp`$_3$) `upd` 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`. An update expression of the form **if** (`exp`) `upd` represents an update `upd` that is executed only if `exp` is true. [7]

An invocation expression `exp`$_0$`->op(exp`$_1$, $\cdots$, `exp`$_n$`)` represents an invocation of the operation `op`. The symbolic expression `exp`$_0$ denotes the receiver object of the operation and the symbolic expressions `exp`$_1$, $\cdots$, `exp`$_n$ denote the parameters. An invocation expression of the form **for** (`i=exp`$_1$; `i<exp`$_2$; `i+=exp`$_3$) `inv` 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`. An invocation expression of the form **if** (`exp`) `inv` represents an operation `inv` that is invoked only if `exp` is true. [8]

[7] For some operations, the compiler may be unable to generate update expressions that accurately represent the new values of the instance variables. 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, update expressions are available for all operations that the parallel phase many invoke.

[8] For some operations, the compiler may be unable to generate 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.