Effective Fine-Grain Synchronization For Automatically Parallelized Programs Using Optimistic Synchronization Primitives

MARTIN C. RINARD
Massachusetts Institute of Technology

This paper presents our experience using optimistic synchronization to implement fine-grain atomic operations in the context of a parallelizing compiler for irregular, object-based computations. Our experience shows that the synchronization requirements of these programs differ significantly from those of traditional parallel computations, which use loop nests to access dense matrices using affine access functions. In addition to coarse-grain barrier synchronization, our irregular computations require synchronization primitives that support efficient fine-grain atomic operations.

The standard implementation mechanism for atomic operations uses mutual exclusion locks. But the overhead of acquiring and releasing locks can reduce the performance. Locks can also consume significant amounts of memory. Optimisitic synchronization primitives such as *load linked/store conditional* are an attractive alternative. They require no additional memory and eliminate the use of heavyweight blocking synchronization constructs.

We evaluate the effectiveness of optimistic synchronization by comparing experimental results from two versions of a parallelizing compiler for irregular, object-based computations. One version generates code that uses mutual exclusion locks to make operations execute atomically. The other version uses optimistic synchronization. We used this compiler to automatically parallelize three irregular, object-based benchmark applications of interest to the scientific and engineering computation community. The presented experimental results indicate that the use of optimistic synchronization in this context can significantly reduce the memory consumption and improve the overall performance.

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, synchronization, optimistic synchronization, parallel computing, parallelizing compilers

1. INTRODUCTION

Parallelizing compilers have traditionally exploited a specific, restricted form of concurrency: the concurrency available in loops that access dense matrices using affine access functions [Bacon et al. 1994]. The generated parallel programs use a correspondingly restricted kind of synchronization: coarse-grain barrier synchronization at the end of each parallel loop.

As shared-memory multiprocessors become the dominant commodity source of computation, parallelizing compilers must support a wider class of computations. It will be especially important to support irregular, object-based computations, including computa-

An earlier version of this paper appeared at the Sixth ACM Symposium on Principles and Practice of Parallel Programming (PPoPP), Las Vegas, NV, June, 1997.
Author's address: Martin C. Rinard, MIT Laboratory for Computer Science, 545 Technology Square, NE43-620A, Cambridge, MA 02139; email: rinard@lcs.mit.edu.
This research was supported in part by NSF grant CCR-9702297.

tions that access pointer-based data structures such as lists, trees and graphs. We have implemented a parallelizing compiler for this class of computations [Rinard and Diniz 1997]. Our experience using this compiler shows that the automatically generated parallel computations exhibit a very different kind of concurrency. Instead of parallel loops with coarse-grain barrier synchronization, the computations are structured as a set of lightweight threads that synchronize using fine-grained atomic operations. The implementation of these atomic operations has a critical impact on the performance and resource consumption of the generated parallel code.

1.1 Mutual Exclusion Locks

The standard implementation mechanism for atomic operations uses *mutual exclusion locks*. In this approach, each piece of data has an associated lock. To update a piece of data, an operation first acquires the corresponding lock. It performs the update, then releases the lock. Any other operation that attempts to acquire the lock blocks until the first operation releases the lock.

Despite their popularity, mutual exclusion locks are far from an optimal synchronization mechanism. One drawback is the memory required to hold the state of the locks. Locks increase the amount of memory that the program consumes, and can degrade the performance of the memory hierarchy by occupying space in the caches. The overhead of executing the acquire and release constructs may also reduce the performance.

Locking the computation at a coarse granularity (by mapping multiple pieces of data to the same lock) addresses these problems. It reduces the impact on the memory system and may allow the program to amortize the lock overhead over many updates to different pieces of data. Unfortunately, a coarse lock granularity may also introduce *false exclusion*. False exclusion occurs when multiple operations that access different pieces of data attempt to acquire the same lock. The operations execute serially even though they are independent and could, in principle, execute in parallel. False exclusion can significantly degrade the performance by reducing the amount of available parallelism.

1.2 Optimistic Synchronization

Optimistic synchronization is an attractive alternative to locks. Atomic operations that use optimistic synchronization use a *load linked* primitive to retrieve the initial value in an updated memory location. They compute the new value, then use a *store conditional* primitive to attempt to write the new value back into the memory location. If no other operation wrote the location between the load linked and the store conditional, the store conditional succeeds. Otherwise, the store conditional fails, the new value is not written into the location, and the operation typically retries the computation. Operations that use optimistic synchronization never block — they avoid atomicity violations by nullifying and retrying computations. Optimistic synchronization primitives therefore avoid problems (such as poor responsiveness, lock convoys, priority inversions, deadlock, and the need to reclaim locks held by failed processes) that are associated with the use of locks in multiprogrammed systems [Herlihy 1993]. But they also have properties that make them especially appropriate for implementing the fine-grain atomic operations characteristic of irregular parallel computations. Specifically, the use of optimistic synchronization imposes no memory overhead and eliminates the use of heavyweight blocking synchronization primitives.

This paper describes our experience using optimistic synchronization to implement atomic operations in the context of a parallelizing compiler for object-based programs. The compiler accepts an unannotated serial program written in a subset of C++ and automatically generates a parallel program that performs the same computation [Rinard and Diniz 1997]. The compiler is designed to parallelize irregular computations, including computations that manipulate pointer-based data structures such as lists, trees and graphs. Because it uses commutativity analysis [Rinard and Diniz 1997] as its primary analysis technique, the compiler views the computation as consisting of a sequence of *operations* on *objects*. If all of the operations in a given computation commute (i.e. generate the same result regardless of the order in which they execute), the compiler can automatically generate parallel code. For the parallel computation to execute correctly, each operation must execute atomically.

We have implemented two versions of the compiler — one version generates code that uses mutual exclusion locks to make operations execute atomically; the other generates code that uses optimistic synchronization. A comparison characterizes the impact of using optimistic synchronization instead of mutual exclusion locks. Our results show that using optimistic synchronization instead of locks can significantly improve the performance and reduce the amount of memory required to execute the computation.

This paper provides the following contributions:

—**Optimistic Synchronization:** It identifies optimistic synchronization as an effective synchronization mechanism for atomic operations in the context of a parallelizing compiler for object-based programs.

—**Analysis Algorithms:** It presents novel analysis and transformation algorithms that enable a compiler to automatically generate optimistically synchronized parallel code.

—**Experimental Results:** It presents a complete set of experimental results that characterize the overall impact of using optimistic synchronization instead of mutual exclusion locks. These results show that optimistic synchronization can substantially reduce the amount of memory that the program consumes. Furthermore, the optimistically synchronized versions never perform significantly worse than the best lock synchronized versions, and perform significantly better than versions that lock the computation at an inappropriate granularity.

To our knowledge, our compiler is the first compiler to automatically generate optimistically synchronized code.

The remainder of the paper is structured as follows. Section 2 presents the optimistic synchronization primitives that the compiler uses to implement fine-grain atomic operations. Section 3 presents an example that illustrates the use of optimistic synchronization. Section 4 discusses how the differences between optimistic and lock synchronization affect the computation. In Section 5 we present an overview of the technique, commutativity analysis, that our prototype compiler uses to parallelize the applications. In Section 6 we present the synchronization selection algorithm, which enables the compiler to generate optimistically synchronized code. Section 7 presents the experimental results. Section 8 discusses related work. We conclude in Section 9.

2. OPTIMISTIC SYNCHRONIZATION PRIMITIVES

Only recently have modern RISC processors provided efficient implementations of the hardware primitives required for optimistic synchronization. Examples include the load linked/store conditional instructions in the MIPS RISC [Heinrich 1993], PowerPC [Motorola, Incorporated 1993], and DEC Alpha architectures [Digital Equipment Corporation

```
retry: ll    $2,0($4)    # load value
       addiu $3,$2,1     # increment value
       sc    $3,0($4)    # attempt to store new value
       beq   $3,0,retry  # retry if failure
```

Fig. 1. Atomic Increment Using `ll` and `sc`

1992], and the compare-and-swap instructions in the SPARC architecture [Weaver and Germond 1994]. In this section we focus on hardware primitives available on MIPS processors such as the MIPS R4400 and R10000. The two key instructions are the load linked (`ll`) and store conditional (`sc`) instructions [Jensen et al. 1987], which can be used together to atomically update a memory location as follows.

The program first uses a load linked instruction to load the original value from the memory location into a register. It computes the new value into another register, then uses a store conditional instruction to attempt to store the new value back into the memory location. If the memory location was not written between the load linked and store conditional, the store conditional succeeds and writes the new value into the memory location. If the memory location was written between the load linked and the store conditional, the store conditional fails and the new value is not written into the memory location. A flag indicating the success or failure of the store conditional is written into the register in the store conditional instruction that held the new value. The computation typically retries the atomic update until it succeeds. Figure 1 presents an assembly language sequence that uses the `ll` and `sc` instructions to atomically increment an integer variable. Register 4 (`$4`) contains a pointer to the variable to increment.

The standard implementation mechanism for load linked and store conditional uses a reservation address register that holds the address from the last load linked instruction and a reservation bit that is invalidated when the address is written [Michael and Scott 1995].

As implemented in the MIPS processor family, `ll` and `sc` directly support atomic operations only on 32 bit and 64 bit data items. Although it is possible to use the instructions to synthesize atomic operations on larger data items, the transformations may impose substantial data structure modifications [Herlihy 1993]. Because these modifications may degrade the performance, the current compiler uses optimistic synchronization only for updates to 32 or 64 bit data items. Transactional memory [Herlihy and Moss 1993] and Oklahoma update [Stone et al. 1993] would support optimistically synchronized atomic operations on larger objects, but no hardware implementation of these mechanisms currently exists, and it is unclear how efficient any such mechanism would be.

3. AN EXAMPLE

We next provide an example that illustrates how the compiler can use optimistic synchronization to implement atomic operations. The program in Figure 2 implements a 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 [Rinard and Diniz 1997]. But 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

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

Fig. 2. Serial Graph Traversal

make each operation execute atomically with respect to all other operations that access the same object.

Figure 3 presents the code that the compiler generates when it uses mutual exclusion locks to make operations execute atomically. The compiler augments each `graph` 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 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 finishes. 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 [Mohr et al. 1990] can increase the granularity of the resulting parallel computation.

Figure 4 presents a high-level version of the code that the compiler generates when it uses optimistic synchronization. Unlike the version that uses locks, there is no change to the `graph` objects. Instead of acquiring and releasing locks, the `parallel_visit` operation uses a load linked instruction to fetch the value of `sum`. It computes the new value, then uses a store conditional instruction to attempt to store the new value back into `sum`. A loop retries the update until it succeeds.

4. ISSUES

We next discuss how the differences between optimistic and lock synchronization affect the computation.

4.1 Memory System Effects

To generate code that uses mutual exclusion locks, the compiler must augment the data structures with locks. Our experimental results indicate that using locks can significantly increase the amount of memory required to run the program. The memory overhead associated with using locks is therefore an important potential problem. In many cases, it is desirable to run as large a problem as possible, and the amount of available memory is the key factor that limits the runnable problem size.

```
class graph {
  private:
    lock mutex;
    int value, sum;
    graph *left; graph *right;
  public:
    void visit(int);
    void parallel_visit(int);
};
void graph::visit(int p) {
  this->parallel_visit(p);
  wait();
}
void graph::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));
}
```

Fig. 3. Parallel Traversal With Locks

```
class graph {
  private:
    int value, sum;
    graph *left; graph *right;
  public:
    void visit(int);
    void parallel_visit(int);
};
void graph::visit(int p) {
  this->parallel_visit(p);
  wait();
}
void graph::parallel_visit(int p) {
  register int new_sum;
  do {
    new_sum = ll(sum);
    new_sum = new_sum + p;
  } while (!sc(new_sum,sum));
  if (left != NULL) spawn(left->parallel_visit(value));
  if (right != NULL) spawn(right->parallel_visit(value));
}
```

Fig. 4. Parallel Traversal With Optimistic Synchronization

Locks can also have a negative impact on the performance of the memory hierarchy. They occupy space in the data caches, which reduces the amount of useful application data that the caches can hold. They also increase the amount of space between useful application data, which may reduce the spatial locality. If the lock and its associated data are stored on different cache lines, a locked update may generate extra cache misses — operations may incur not only the cache misses required to perform the update, but also an extra cache miss to acquire the lock.

The use of locks may also affect the performance of the instruction cache. The additional instructions required to address, acquire, and release the lock increase the size of the generated code, effectively reducing the amount of application code that the instruction cache can hold. For optimistic synchronization, the conditional branches that retry failed updates are the only additional instructions.

Lock synchronization may also affect the performance of the memory consistency protocol. To execute the program correctly, the machine must globally order the execution of the locking constructs with respect to the memory accesses performed as part of the atomic operation. Modern machines that implement relaxed memory consistency models typically enforce this order by waiting for all outstanding writes to complete globally before releasing a lock [Adve and Gharachorloo 1996; Gharachorloo 1996]. But unlike lock synchronization, optimistic synchronization imposes no additional order between accesses to different memory locations. Although interactions between the caches, optimistic synchronization and out of order execution complicate the implementation on modern processors, in principle the machine can use the same relaxed ordering constraints for optimistically synchronized updates as it does for unsynchronized updates. The fundamental performance differences between optimistically synchronized and unsynchronized updates come primarily from the need to acquire exclusive access to the memory location before a store conditional can succeed and control dependences from the branch that retries the update if it fails.

A significant advantage of optimistic synchronization is that it imposes no memory, data cache or memory consistency overheads — the serial and parallel programs have identical data memory layouts, identical data memory access patterns and identical ordering constraints between accesses to different memory locations.

4.2 Synchronization Granularity

As described in Section 2, existing optimistic synchronization primitives support atomic operations only on individual data items. The advantage of synchronizing at this fine granularity is that it minimizes the possibility of false exclusion and maximizes the exposed concurrency. But it may not always be desirable or even possible to synchronize at this granularity. If an operation updates many data items, it may be more efficient to acquire a lock, perform all of the updates without additional synchronization, then release the lock. Furthermore, atomic operations that perform multiple interdependent updates to multiple data items cannot synchronize at the granularity of individual items — all of the updates must execute atomically together as a group. In this case, the compiler must generate code that uses lock synchronization.

4.3 Impact on Applications

The extent to which an application can use optimistic synchronization depends on the characteristics of its atomic operations. The parallel phases of many applications compute a set of contributions that are atomically accumulated into shared objects using simple binary operators such as, max, min, addition, or multiplication. The basic atomic operations in these applications update a single word of memory, and all of these atomic operations can use optimistic synchronization instead of mutual exclusion locks. All of our benchmark applications use this kind of atomic operation exclusively, as do many of the applications in parallel benchmark suites such as the SPLASH and SPLASH2 benchmarks [Singh et al. 1992; Woo et al. 1995].

Problematic cases tend to arise when applications use atomic operations to construct or update linked data structures in parallel, as opposed to simply traversing the data structures in parallel. The version of Barnes-Hut in the SPLASH2 benchmark suite, for example, builds an space-subdivision tree in parallel. The atomic operations used to build or update linked data structures tend to update multiple words of data atomically, which makes them unsuitable for the simple atomic synchronization primitives that are currently available on RISC microprocessors. Experience with operating systems kernels, however, suggests that more advanced primitives such as double compare and swap may make it possible to recode these computations to use optimistic synchronization instead of mutual exclusion locks [Greenwald and Cheriton 1996; Massalin and Pu 1989].

A final question is whether it may be more efficient to synchronize at the coarser granularity of objects instead of using optimistic synchronization. For our set of benchmark applications, synchronizing at the granularity of objects never significantly improves the performance. We expect, however, that applications with coarser granularity updates than those in our set of benchmark applications may execute more efficiently with lock synchronization than with optimistic synchronization. Given the relatively efficient current implementations of optimistic synchronization primitives, we would not expect to see dramatic performance differences from this effect.

5. COMMUTATIVITY ANALYSIS

We next provide an overview of *commutativity analysis*, the technique that our compiler uses to automatically parallelize our set of applications [Rinard and Diniz 1997]. Commutativity analysis is designed to parallelize 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.

The commutativity analysis algorithm analyzes the program at the granularity of *operations* on *objects* to determine if the operations commute, i.e., 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 must consider 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 receiver objects of A and B 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 have different receiver 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 reasons about the values computed in the two execution orders.

The compiler uses symbolic execution [King 1976; Kemmerer and Eckmann 1985] to extract expressions that denote the new values of instance variables and the multiset of invoked operations. Symbolic execution simply executes the methods, computing with expressions instead of values. It maintains a set of bindings that map variables to expressions that denote their values and updates the bindings as it executes the methods. The compiler uses the extracted expressions from the symbolic execution to apply the commutativity testing conditions presented above. If the compiler cannot determine that corresponding expressions are equivalent, it must conservatively assume that the two operations do not commute.

5.1 Automatic Parallelization and Parallel Phases

The compiler uses commutativity analysis as outlined above to parallelize the program. It recursively traverses the call graph of the program visiting operation invocation sites as follows. At each site, it first computes the set of operations executed by the computation rooted at that site. It then uses commutativity analysis to determine if all pairs of operations in this set commute. If so, the compiler generates code that executes the operations in parallel. We refer to such parallelized computations as *parallel phases*. The parallel phase therefore corresponds to the entire computation rooted at the invocation site.

As soon as the traversal detects a parallel phase, it does not recursively visit the operation invoked at the invocation site. It instead skips to the next invocation site after the parallel phase. If the compiler was unable to parallelize the computation rooted at the call site, the traversal recursively visits the invocation sites in the invoked operation.

5.2 Synchronization in the Generated Code

Commutativity analysis assumes that the operations in the parallel phases execute atomically. When the compiler generates the parallel code, it must therefore choose a synchronization mechanism and insert the corresponding synchronization constructs into operations in parallel phases that access potentially updated objects. These constructs ensure that the operation executes atomically with respect to all other operations in the parallel phase that access the same object. A standard mechanism is to augment each potentially updated object with a *mutual exclusion lock*. The generated code for each operation would first acquire the object's lock, access the object, then release the lock. This paper presents the results of a compiler that, when possible, uses optimistic synchronization instead of the standard lock synchronization.

6. THE SYNCHRONIZATION SELECTION ALGORITHM

Synchronization selection takes place after the commutativity analysis algorithm has successfully parallelized a phase of the computation as outlined above in Section 5. The synchronization selection algorithm chooses a synchronization mechanism for all variables updated by operations in the phase. The algorithm uses optimistic synchronization whenever possible. The synchronization mechanism for a given variable can differ across parallel phases — i.e., one phase may use lock synchronization, while another phase may use optimistic synchronization for the same variable.

Even though we implemented the synchronization selection algorithm in the context of our parallelizing compiler, the algorithm is independent of the specific means used to extract the concurrency, and can be used both for automatically and manually parallelized computations.

6.1 Model of Computation

We next outline the basic requirements for the application of the synchronization selection algorithm. The algorithm is designed to work on pure object-based programs with parallel phases and atomic operations. For each atomic operation, the algorithm must be able to determine the operations that may execute in parallel with the atomic operation and the instance variables that the operations update. To use optimistic synchronization, it must also be able to generate expressions that denote the new values of updated instance variables.

In our compiler, the commutativity analysis algorithm extracts some of the information that the synchronization selection algorithm uses. Specifically, it produces the set of operations that each parallel phase may invoke and the set of instance variables that these operation may update [Rinard and Diniz 1997]. For each operation, it 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; Kemmerer and Eckmann 1985; Rinard and Diniz 1997].

6.2 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** $(\texttt{i=exp}_1;\texttt{i<exp}_2;\texttt{i+=exp}_3)$ `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. 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 synchronization selection algorithm runs only after the commutativity analysis algorithm has successfully parallelized a phase, update expressions are available for all operations that the parallel phase many invoke.

6.3 Invocation Expressions

An invocation expression $\texttt{exp}_0\texttt{->op}(\texttt{exp}_1, \cdots, \texttt{exp}_n)$ represents an invocation of the operation op. The symbolic expression $\texttt{exp}_0$ denotes the receiver object of the operation and the symbolic expressions $\texttt{exp}_1, \cdots, \texttt{exp}_n$ denote the parameters. An invocation expression of the form **for** $(\texttt{i=exp}_1;\texttt{i<exp}_2;\texttt{i+=exp}_3)$ 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. 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 synchronization selection algorithm runs only after the commutativity analysis algorithm has successfully parallelized a phase, invocation expressions are available for all operations that the parallel phase many invoke.

6.4 Synchronization Selection Requirements

To execute correctly, all accesses to a potentially updated variable must use the same synchronization mechanism. The synchronization selection algorithm therefore classifies each potentially updated variable as either an *optimistically synchronized variable*, (a variable whose updates can use optimistic synchronization or no synchronization) or a *lock synchronized variable* (a variable whose accesses must use lock synchronization).

The commutativity analysis algorithm assumes that each operation executes atomically with respect to other operations that access the same object. The analysis takes place at the granularity of instance variables and multisets of invoked operations — two operations commute if the instance variables and multisets of invoked operations are the same in both execution orders [Rinard and Diniz 1997]. But optimistically synchronized updates execute atomically only at the granularity of individual updates, not at the coarser granularity of complete operations (each operation may perform multiple updates). If the synchronization selection algorithm chooses to optimistically synchronize a set of updates, it must ensure that the generated parallel program always produces the same result as the corresponding program in which all operations execute atomically.

The atomicity requirements may force the synchronization selection algorithm to use lock synchronization. Consider, for example, a computation that contains an operation that updates multiple variables and an update in a different operation that reads all of the variables. To satisfy the atomicity requirements, the updates in the first operation must execute atomically as a group with respect to the update that reads the variables. Because optimistically synchronized updates are atomic only at the granularity of individual updates, the synchronization selection algorithm cannot optimistically synchronize the updates in the first operation. All of the updated variables must be classified as lock synchronized variables.

The current algorithm applies two constraints to enforce the atomicity requirements.

—First, each update to an optimistically synchronized variable may access only the updated variable and variables that are not updated during the parallel phase.

—Second, all accesses to an optimistically synchronized variable may occur only in updates to that variable — no other update reads the variable and no operation invocation site depends on the variable.

It is possible to relax these constraints. For example, if an operation accessed only one updated variable, the compiler could allow its operation invocation sites to depend on the variable even if the variable were optimistically synchronized. To ensure that updates to the variable would execute atomically with respect to the accesses in the operation, the generated code would read the value of the variable into a local variable. It would then access the value from that local variable for the remainder of the operation.

It would also be possible to integrate the commutativity testing and synchronization selection more closely. In such a scenario, the synchronization selection algorithm would propose a set of optimistically synchronized variables, and the commutativity testing would take place at the finer granularity of individual updates to those variables.

The practical impact of these generalizations would depend on the characteristics of the applications. In general, we expect the current algorithm to recognize most of the opportunities to use optimistic synchronization that occur in practice. But experience with a broader range of applications would shed additional light on the situation.

6.5 The Algorithm

We next present some notation that we will use when we present the synchronization selection algorithm. The program defines a set of classes $\mathtt{cl} \in \mathtt{CL}$ and a set of operations $\mathtt{op} \in \mathtt{OP}$. Given an operation op, the function receiverClass($\mathtt{op}$) returns the class of the receiver objects of op. The program also defines a set of instance variables $\mathtt{v} \in \mathtt{V}$. The function instanceVariables($\mathtt{cl}$) returns the instance variables of the class cl. No two classes share an instance variable — i.e., $\text{instanceVariables}(\mathtt{cl}_1) \cap \text{instanceVariables}(\mathtt{cl}_2) = \emptyset$ if $\mathtt{cl}_1 \neq \mathtt{cl}_2$.

Figure 5 presents the algorithm. It takes as parameters a set of invoked operations, a set of updated variables, a function updates($\mathtt{op}$), which returns the set of update expressions that represent the updates that the operation op performs, and a function invocations($\mathtt{op}$), 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($\mathtt{exp}$) returns the set of variables in the symbolic expression exp, variables($\mathtt{upd}$) returns the set of free variables in the update expression upd, and variables($\mathtt{inv}$) returns the set of free variables in the invocation expression inv. 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. The algorithm produces a set of variables whose updates may be optimistically synchronized, a set of classes that must be augmented with locks, and a set of operations that must use lock synchronization.

The algorithm determines the kind of synchronization it must use by processing all of the updates and invocations. For each update, the choices are to use lock synchronization, optimistic synchronization, or no synchronization at all. For simplicity, the algorithm classifies each variable as either requiring lock synchronization or able to use optimistic synchronization. Some updates to variables classified as able to use optimistic synchronization may require no synchronization at all. When the compiler generates the code for such updates, it simply omits the synchronization. The decision to generate or omit synchronization is based on the form of the update as discussed below.

```
synchronizationSelection(invokedOperations, updatedVariables, updates, invocations)
  lockedClasses = ∅;
  lockedOperations = ∅;
  optimisticVariables = updatedVariables;
  for all op ∈ invokedOperations
    for all u ∈ updates(op)
      if (requiresLockSynchronization(u, updatedVariables))
        optimisticVariables = optimisticVariables − variables(u);
        lockedClasses = lockedClasses ∪ {receiverClass(op)};
    for all i ∈ invocations(op)
      if (variables(i) ∩ updatedVariables ≠ ∅)
        optimisticVariables = optimisticVariables − variables(i);
        lockedClasses = lockedClasses ∪ {receiverClass(op)};
  lockedVariables = updatedVariables − optimisticVariables;
  for all op ∈ invokedOperations
    for all u ∈ updates(op)
      if (variables(u) ∩ lockedVariables ≠ ∅)
        lockedOperations = lockedOperations ∪ {op};
    for all i ∈ invocations(op)
      if (variables(i) ∩ lockedVariables ≠ ∅)
        lockedOperations = lockedOperations ∪ {op};
  return ⟨optimisticVariables, lockedClasses, lockedOperations⟩;


requiresLockSynchronization(u, updatedVariables)
  if (u is of the form v = exp and variables(exp) ∩ updatedVariables = ∅)
    return false;
  if (u is of the form v[exp′] = exp and (variables(exp) ∪ variables(exp′)) ∩ updatedVariables = ∅)
    return false;
  if (u is of the form v = v ⊕ exp and variables(exp) ∩ updatedVariables = ∅)
    return false;
  if (u is of the form v[exp′] = v[exp′] ⊕ exp and
     (variables(exp) ∪ variables(exp′)) ∩ updatedVariables = ∅)
    return false;
  if (u is of the form if (exp) upd and variables(exp) ∩ updatedVariables = ∅)
    return requiresLockSynchronization(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 requiresLockSynchronization(upd, updatedVariables);
  return true;
```

Fig. 5. Synchronization Selection Algorithm

Three factors contribute to the synchronization choice for a given variable: the requirement of each update to the variable, the way that other updates use the variable, and the way that invocations use the variable. Each update imposes the following requirement on the updated variable:

—**No Synchronization:** If an update computes a new value that does not depend on any updated variable, then writes the value into a variable, it is a candidate for using no synchronization at all. The sole responsibility of the synchronization selection algorithm is to ensure that the operations execute atomically. Because store instructions execute atomically, there may be no need to explicitly synchronize an update that simply writes a value into an instance variable if the value does not depend on a variable that some other operation may update.

—**Optimistic Synchronization:** In principle, if an update reads an instance variable, computes a new value that does not depend on a different variable that another operation might update, then writes the new value back into the variable, the update should be a candidate for optimistic synchronization. In the MIPS R4400, however, it is illegal to perform a memory access between the `ll` and `sc` instructions. The compiler could still use optimistic synchronization whenever it is possible to compute the new value without accessing memory after the initial read of the instance variable. To simplify the implementation of the compiler, however, the algorithm uses optimistic synchronization only when the new value can be expressed in the form $\mathtt{v} \oplus \mathtt{exp}$ or $\mathtt{v}[\mathtt{exp}'] \oplus \mathtt{exp}$, where `v` or $\mathtt{v}[\mathtt{exp}']$ denotes the original value of the variable, $\oplus$ is an arbitrary binary operator, and `exp` and $\mathtt{exp}'$ do not contain an updated variable. In this case, the generated code can compute the value of `exp` into a register, then use an instruction sequence similar to that in Figure 1 to atomically perform the computation and update the variable.

—**Lock Synchronization:** The algorithm classifies all other updates as requiring lock synchronization.

In a given parallel phase, all accesses to an updated variable must use the same synchronization mechanism. If even one update that accesses the variable (either by reading the variable or writing the variable) requires lock synchronization, then all updates that access the variable must use lock synchronization.

The algorithm starts by assuming that all variables can use either optimistic synchronization or no synchronization at all. It then scans the updates to find updates that require lock synchronization. Any variable involved in such an update is removed from the set of optimistically synchronized variables. The compiler will augment the class of the receiver object of the variable with a lock and insert constructs that acquire and release the lock into all operations that access the variable. The invocation expressions capture the remainder of the operation's accesses to updated variables. If an invocation expression accesses an updated variable, the algorithm deletes the variable from the set of optimistically synchronized variables. All accesses to the variable will use lock synchronization. The algorithm computes the set of operations that must use lock synchronization by scanning the operations to find all operations that access a lock synchronized variable.

6.6 Multiple Updates In Loops

There is a technical detail associated with loops that update the same variable or array element more than once. If the updates are optimistically synchronized, they are atomic only at the granularity of the individual loop iterations, not at the granularity of the entire loop. With lock synchronization, they would be atomic at the granularity of the entire loop. Optimistically synchronizing variables that may be updated multiple times in a loop makes the granularity of the enforced atomicity finer. Because the commutativity analysis for such variables takes place at the granularity of individual loop iterations instead of at the granularity of the entire loop, this change in the granularity does not affect the correctness of the optimistically synchronized parallel program.

6.7 Code Generation

If an operation must use lock synchronization, the generated code acquires the lock in the receiver object before it accesses any potentially updated instance variables; it does not release the lock until it has completed all of its accesses to potentially updated variables. The commutativity analysis algorithm ensures that all invocations of operations that may access potentially updated variables occur after the invoking operation has completed its last access to a potentially updated variable [Rinard and Diniz 1997]. This *separability* property ensures that the generated code never attempts to acquire more than one lock, which in turn ensures that it never deadlocks.

It is possible for an operation to synchronize some of its updates using locks and other updates using optimistic synchronization. The nonblocking nature of optimistic synchronization ensures that the combination of the two different synchronization mechanisms never causes deadlock. Finally, note that the code generation algorithm augments a class with a lock only if one of the phases uses lock synchronization when it updates an object of that class. So if the compiler is able to use optimistic synchronization for all updates in parallel phases to objects of a given class, the object layout is the same in the parallel and serial versions and there is no memory overhead for locks in objects of that class.

7. EXPERIMENTAL RESULTS

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

7.1 The Compilation System

We implemented a prototype parallelizing compiler that uses commutativity analysis as its basic analysis paradigm. Compiler flags determine whether it generates code that uses mutual exclusion locks or (when possible) optimistic synchronization.

The compiler is structured as a source-to-source translator that takes a serial program written in a subset of C++ and generates an explicitly parallel C++ program that performs the same computation. We use Sage++ [Bodin et al. 1994] as a front end. The analysis and code generation phases consist of approximately 21,000 lines of C++ code. This count includes no code from the Sage++ system. The generated parallel code contains calls to a run-time library that provides the basic concurrency management and synchronization functionality. The library consists of approximately 6000 lines of C code.

The current version of the compiler imposes several restrictions on the dialect of C++ that it can analyze. The major restrictions include the following:

—The program does not use operator or method overloading.
—The program uses neither multiple inheritance nor templates.
—The program contains no `typedef`, `union`, `struct`, or `enum` types.
—Global variables cannot be primitive data types; they must be class types.
—The program has no static members.
—The program contains no casts between base types such as `int`, `float`, and `double` that are used to represent numbers. The program may contain casts between pointer types; the compiler assumes that the casts do not cause the program to violate its type declarations.
—The program contains no default arguments or methods with variable numbers of arguments.
—No operation accesses an instance variable of a nested object of the receiver or an instance variable declared in a class from which the receiver's class inherits.
—The program has no virtual methods or function pointers.
—The program does not use exceptions.
—The program does not use pointers to members.

In addition to these restrictions, the compiler assumes that the program has been type checked and does not violate its type declarations. The goal of these restrictions is to simplify the implementation of the compiler while providing enough expressive power to allow the programmer to develop clean object-based programs. Most of the restrictions are designed simply to reduce the number of cases that the compiler must handle, and impose no conceptual limitation on the expressive power of the language. But the restrictions against virtual methods, function pointers, exceptions, and pointers to members do significantly limit the expressive power of the language. Supporting these constructs in our compiler would require the presence of additional analysis algorithms. The current call graph construction algorithm, for example, would have to be extended to determine which methods could be invoked at virtual function call sites and call sites that use function pointers. The symbolic execution algorithm would have to be extended to support the additional control flow associated with exceptions and to operate conservatively in the presence of pointers to members. While none of these extensions would fundamentally affect the commutativity analysis or synchronization selection algorithms discussed in this paper, they would require a significant engineering effort to integrate into the compiler.

7.2 Methodology

We collected experimental results for the applications running on an SGI Challenge XL multiprocessor with 24 100 MHz R4400 processors running IRIX version 6.2. We compiled the generated parallel programs using version 7.1 of the MipsPro compiler from Silicon Graphics. We implemented two versions of the lock primitives: a spin lock [Heinrich 1993], and a Mellor-Crummey Scott (MCS) lock [Mellor-Crummey and Scott 1991]. The spin lock acquire is implemented using a compiler intrinsic that uses `ll` and `sc` to atomically test and set a value that indicates whether the lock is free or not. The release simply clears the value. Whenever an attempt to acquire a lock fails, the processor immediately reexecutes the instruction sequence that attempts to acquire the lock: there is no backoff. Spin locks perform well if there is little lock contention. If there is substantial lock contention, spin locks typically perform poorly because they generate a large amount of invalidation traffic [Mellor-Crummey and Scott 1991].

MCS locks are designed to perform well across a range of locking conditions. Processors waiting to acquire a lock are arranged in a linked list; each processor spins on a separate memory location. MCS locks therefore avoid significant invalidation traffic even for heavily contended locks. The drawback is that the MCS lock implementation executes a compare-and-swap operation both when the lock is acquired and when the lock is released. As Table I in Section 7.3 illustrates, it therefore takes significantly longer to acquire and release an available MCS lock than it does to acquire and release an available spin lock.

We used the compiler to obtain the following versions of each application:

—**Optimistic:** When possible, the compiler generates code that uses optimistic synchronization. For our three applications, the atomic operations use optimistic synchronization exclusively — the generated code contains no mutual exclusion locks. The serial and Optimistic versions therefore have identical memory layouts.

—**Item Lock:** If a primitive data item (such as an `int`, `float` or `double`) in an object may be updated in a parallel phase, there is a spin lock associated with that item. If an operation in a parallel phase updates an item, it acquires the corresponding lock, performs the update, then releases the lock.

—**Object Lock:** If an object may be updated in a parallel phase, there is a spin lock associated with that object. When an operation in a parallel phase updates an object, it acquires the object's lock, performs the update, then releases the lock. Each nested object has the same lock as its enclosing object.

—**Coarse Lock:** Like the Object Lock version, there is a spin lock associated with each object and each operation that updates an object holds that object's lock. But the compiler analyzes the program to detect sequences of operations that acquire and release the same lock. It then transforms the sequence so that it acquires the lock once, executes the operations without synchronization, then releases the lock [Diniz and Rinard 1998; 1997].

—**Item MCS Lock:** The same as Item Lock listed above, except that the generated code uses MCS locks instead of spin locks.

—**Object MCS Lock:** The same as Object Lock listed above, except that the generated code uses MCS locks instead of spin locks.

—**Coarse MCS Lock:** The same as Coarse Lock listed above, except that the generated code uses MCS locks instead of spin locks.

The Optimistic versions synchronize at the granularity of individual data items. The Item Lock versions also synchronize at this granularity, but use locks instead of optimistic synchronization. Because the Object Lock versions synchronize at the coarser granularity of objects, they allocate fewer locks and execute fewer lock constructs than the Item Lock versions. The tradeoff, of course, is that the Object Lock versions may suffer from false exclusion.

The Coarse Lock versions allocate the same number of locks as the Object Lock versions, but execute fewer lock constructs. The tradeoff is that the Coarse Lock versions may suffer from *false contention*. False contention occurs when a processor attempts to acquire a lock, but the processor that currently holds the lock is executing code that was originally in no atomic operation. The first processor must wait until the lock is released, even though the computations are independent and should, in principle, be able to execute concurrently. False contention can significantly degrade the performance by reducing the amount of available parallelism.

The compiler is structured as a source to source translator from serial C++ to parallel C++ containing synchronization primitives and calls to procedures in our run time system. Starting with an unannotated, serial C++ program, the compiler generates the Object Lock and Coarse Lock versions automatically with no programmer intervention. Although it would be possible to generate the Item Lock versions automatically, we generated these versions by hand starting from the Object Lock version. Ideally, the generated code for the Optimistic versions would use compiler intrinsics to implement the atomic updates. But the 7.1 version of the MipsPro compiler does not support such compiler intrinsics for floating point numbers, and all of our applications need to perform atomic updates on floating point numbers. The Optimistic versions therefore contain calls to assembly language routines that implement the atomic updates using optimistic synchronization primitives.

7.3 Cost Of Basic Operations

Table I presents the execution times for a single update implemented using different synchronization mechanisms. Each update reads an array element, adds a constant to the element, then stores the new value back into the array. 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 times vary significantly for the cached versions, with the optimistically synchronized update executing faster than the lock synchronized updates. The execution times of the uncached versions are dominated by the cache miss time and are roughly comparable for the different synchronization mechanisms.

Version	Execution Time For One Cached Update (microseconds)	Execution Time For One Uncached Update (microseconds)
No Synchronization	0.049	3.34
Optimistic Synchronization	0.20	3.48
Lock Synchronization	0.34	3.69
MCS Lock Synchronization	0.56	3.93

Table I. Measured Execution Times for One Update

7.4 Barnes-Hut

We start our discussion of Barnes-Hut by analyzing the memory overhead of using locks. Table II presents the *memory usage* for each version — the amount of memory used to store objects during the execution of the program. The applications store all of their data except local variables in objects. The memory usage therefore indicates the total heap data usage of the program.* The *lock overhead* is the percentage of memory used to hold locks.

*The reader should bear in mind that the numbers in Table II are for a realistic but small data set — 16,384 particles. Larger data sets would increase the memory usage.

Version	Memory Usage (Mbytes)	Lock Overhead
Serial	5.37	-
Optimistic	5.37	-
Item Lock	7.08	24%
Object Lock	6.30	15%
Coarse Lock	5.51	2.5%
Item MCS Lock	7.08	24%
Object MCS Lock	6.30	15%
Coarse MCS Lock	5.51	2.5%

Table II. Lock Memory Overheads for Barnes-Hut

Version	Number of Lock Acquire/Release Pairs	Number of Optimistically Synchronized Updates
Optimistic	-	108,641,972
Item Lock	108,641,972	-
Object Lock	54,271,834	-
Coarse Lock	212,992	-
Item MCS Lock	108,641,972	-
Object MCS Lock	54,271,834	-
Coarse Lock	212,992	-

Table III. Number of Synchronization Operations for Barnes-Hut

Locks impose a significant memory overhead, with the Item Lock version allocating twice as much memory for locks as the Object Lock and Coarse Lock versions.

Table III presents the number of synchronization operations performed by the different versions. The Optimistic and Item Lock versions synchronize each update separately from all other updates. The Item Lock version therefore executes as many acquire/release pairs as the Optimistic version executes optimistically synchronized updates. The Object Lock version executes half as many acquire and release pairs as the Item Lock version — on average, the computation performs two updates every time it acquires and releases a lock. The Coarse Lock version executes dramatically fewer acquire/release pairs than the Item Lock and Object lock versions. On average, the Coarse Lock version performs approximately 510 updates every time it acquires and releases a lock.

Table IV presents the running times for Barnes-Hut as a function of the number of processors executing the computation. We evaluate how well the application scales by computing the *speedup*, which is the running time of the serial version divided by the running time of the parallel version. The serial version executes with no parallelization overhead. The serial versions of Barnes-Hut and Water perform slightly better than a highly optimized version written in C [Rinard and Diniz 1997]. The serial version of String performs slightly worse than the corresponding C version. Figure 6 presents the speedups using speedup curves, which plot the speedup as a function of the number of processors executing the computation. These curves show that Barnes-Hut scales reasonably well — the speedup over the Serial version is above 12 out of 24 processors for all versions.

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:

Version	Processors 1	4	8	12	16	20	24
Serial	136.34	-	-	-	-	-	-
Optimistic	147.10	41.05	22.06	16.55	13.00	11.41	10.08
Item Lock	169.20	46.84	25.56	18.58	14.79	12.87	11.62
Object Lock	150.04	41.74	22.71	16.84	13.43	11.79	10.37
Coarse Lock	135.09	38.88	20.90	15.83	12.38	11.10	9.63
Item MCS Lock	211.65	55.98	30.18	22.00	17.34	14.95	13.24
Object MCS Lock	173.55	47.70	25.69	18.76	14.92	12.97	11.67
Coarse MCS Lock	139.12	39.15	20.80	15.79	12.49	11.13	9.71

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

—**Atomic Operation:** The amount of time spent executing operations that require synchronization to execute atomically or operations that contain lock constructs. Performance problems caused by contention for locks or retried updates show up as increases in this component of the execution time.

For the Optimistic, Item Lock and Object Lock versions, all application-level synchronization takes place inside operations that require synchronization to execute atomically. For the Coarse Lock versions, the compiler occasionally lifts lock constructs out of operations that require synchronization into operations that would otherwise not contain lock constructs. Our applications spend very little time executing such operations. Barnes-Hut always spends less than 0.5% of its execution time executing operations that contain lifted lock constructs; Water always spends less than 1.8% of its execution time executing operations that contain lifted lock constructs. As discussed below in Section 7.5, the Coarse Lock versions of String execute sequentially because of false contention, so we do not present performance results for these versions.

—**Idle:** The amount of 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.

—**Serial Compute:** Time spent computing in serial phases of the computation.

—**Parallel Compute:** Time spent computing in parallel phases of the computation.

Figure 7 graphically presents the time breakdowns for the different versions of Barnes-Hut. For each category, the size of the part of the bar dedicated to that category corresponds to the sum over all processors of the amount of time the processor spends in that category. Ideally, the total height of the bar divided by the number of processors would therefore be the execution time of the application on that number of processors. In practice, there are several sources of instrumentation error that introduce discrepancies between the two measurements. First, the time breakdown numbers are from runs with program counter sampling turned on, while the execution times are from runs without this instrumentation. Second, the execution times reflect the absolute running times, while the time breakdown numbers reflect the amount of time spent executing on the processor. Third, there is a delay between the time when the application completes its execution and when the program counter sampling is turned off on all processors and the results collected. This delay can artificially increase the time breakdown numbers relative to the execution time numbers. Given these sources of instrumentation error, the time breakdown numbers should be viewed primarily as giving qualitative insight into the application's overall performance.

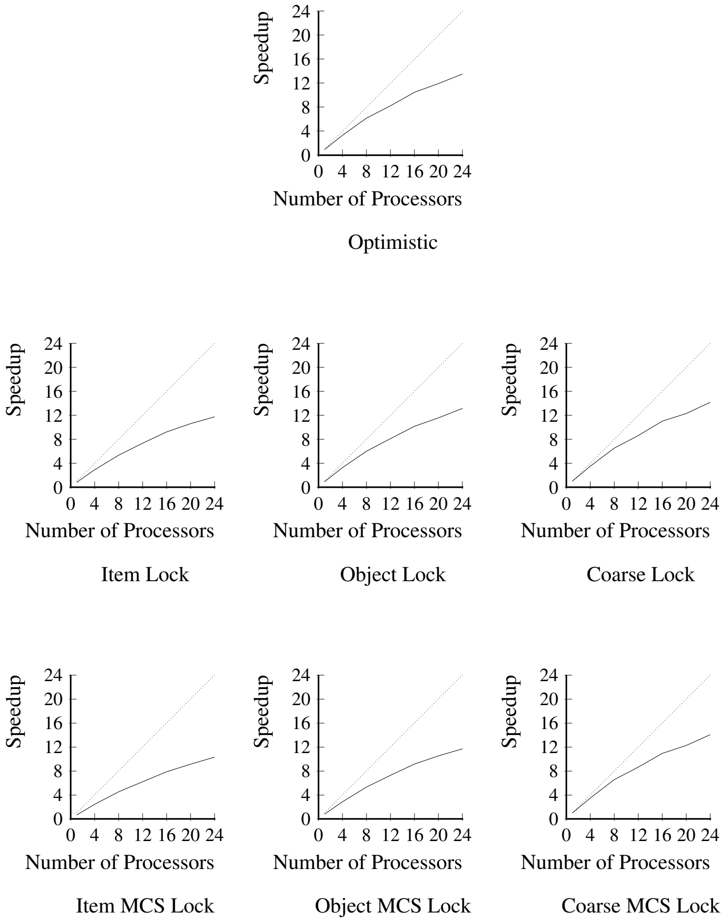

Fig. 6. Speedups for Barnes-Hut

For Barnes-Hut, the time breakdown numbers show that 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 that limits the performance.

The time breakdowns for the different versions are approximately equivalent except for the atomic operation times, which directly reflect the efficiency differences between the different synchronization mechanisms. For each version, the atomic operation times stay roughly constant as the number of processors increases. This independence of the number of processors indicates that contention is not a problem in any of the versions, that the Object Lock and Coarse Lock versions do not incur false exclusion, and that the Coarse Lock version does not incur false contention. Note that contention occurs when two operations attempt to update the same object or data item. In the Optimistic version, contention causes the store conditional instruction to fail, which in turn causes the operation to retry the update. In the versions that use locks, contention causes multiple operations to attempt

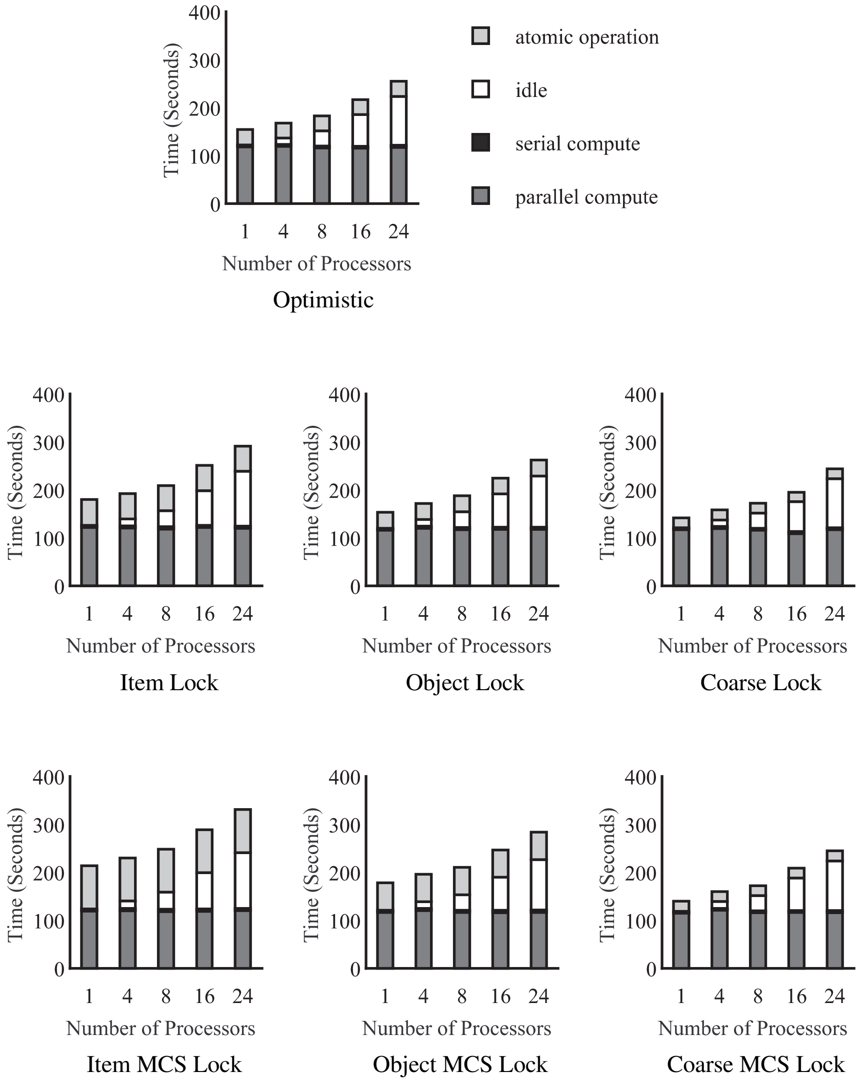

Fig. 7. Time Breakdowns for Barnes-Hut

to acquire the same lock at the same time. One of the operations acquires the lock, and the other operations must wait until that operation finishes the update and releases the lock. Contention shows up in the time breakdowns as an increase in the atomic operation time. Given the lack of false exclusion and false contention, the Coarse Lock version performs best, although the difference between all of the versions is quite small as the computation scales.

7.5 String

For String, false contention in the Coarse Lock version completely serializes the computation. We therefore present results for only the Optimistic, Item Lock, and Object Lock versions. Table V presents the lock memory overheads for String. For the Object Lock version, the overhead is negligible. For the Item Lock version, allocating one lock for each data item significantly increases the overhead. Table VI presents the number of synchronization operations for the different versions. The Item Lock and Object Lock versions both execute one lock acquire/release pair per update — the increased lock granularity in the Object Lock version does not decrease the number of executed lock constructs.

Table VII presents the execution times; Figure 8 presents the corresponding speedup curves. The Optimistic and Item Lock versions scale almost perfectly up to 24 processors. The Object Lock version, however, stops scaling at 16 processors. An examination of the time breakdowns in Figure 9 shows that the atomic operation times for the Object Lock versions grow dramatically as the number of processors increases, while the Optimistic and Item Lock versions spend almost no time executing atomic operations regardless of the number of processors executing the computation. This difference in the atomic operation times indicates that false exclusion causes the poor performance in the Object Lock version.

An examination of the application helps to explain the performance differences. String repeatedly updates individual elements of a large aggregate data structure stored in a single object. Because the update pattern is very irregular and is determined in part by the input data, it is impractical to lock the data structure at any granularity other than the object or item granularities.

In the Object Lock version, there is one lock for the entire aggregate. Operations that attempt to concurrently update any item in the aggregate must contend for that one lock even if they update different items. The results indicate that this coarse synchronization granularity generates a significant amount of contention. Because the Optimistic and Item Lock versions synchronize at the granularity of individual items, operations suffer from contention only if they attempt to concurrently update the same item. The results indicate that this fine synchronization granularity almost completely eliminates contention.

7.6 Water

Table VIII presents the lock memory overheads for Water. The use of locks significantly increases the amount of memory that the program consumes. Table IX presents the number of synchronization operations. The Object Lock version executes approximately 2.75 updates per lock acquire/release pair; the Coarse Lock version executes approximately 5.5 updates per acquire/release pair.

Table X presents the execution times; Figure 10 presents the corresponding speedup curves. The application scales reasonably well to 8 or 12 processors (depending on the specific version), then the performance levels off. The time breakdowns in Figure 11 show

Version	Memory Usage (Mbytes)	Lock Overhead
Serial	3.58	-
Optimistic	3.58	-
Item Lock	5.98	40%
Object Lock	3.57	0%
Item MCS Lock	5.98	40%
Object MCS Lock	3.57	0%

Table V. Lock Memory Overheads for String

Version	Number of Lock Acquire/Release Pairs	Number of Optimistically Synchronized Updates
Optimistic	-	30,036,938
Item Lock	30,036,938	
Object Lock	30,036,938	-
Item MCS Lock	30,036,938	
Object MCS Lock	30,036,938	-

Table VI. Number of Synchronization Operations for String

	Processors						
Version	1	4	8	12	16	20	24
Serial	886.25	-	-	-	-	-	-
Optimistic	899.24	227.07	113.17	76.75	56.05	46.80	39.59
Item Lock	885.16	223.41	113.50	76.62	55.91	49.53	38.78
Object Lock	894.85	235.37	125.78	93.23	78.31	80.14	86.29
Item MCS Lock	889.31	229.16	114.70	77.37	57.80	45.95	38.76
Object MCS Lock	913.04	245.51	135.56	110.00	99.45	94.99	89.75

Table VII. Execution Times for String (seconds)

that increased atomic operation times cause the poor performance. In one of the parallel phases, there is a single one-word object that all of the parallel threads repeatedly update. As the number of processors increases, these updates generate a synchronization bottleneck in all of the versions.

7.7 Discussion

The experimental results demonstrate that optimistic synchronization is the superior choice for this set of applications. It imposes no memory overhead at all, and, for all of the benchmark applications, the fastest lock synchronized version never performs significantly better than the optimistically synchronized version. This robustness enables a compiler to automatically generate optimistically synchronized code without risking a significant degradation in the performance. Based on these results, we believe that, whenever possible, compilers should generate code that uses optimistic synchronization instead of locks.

The results also demonstrate the simplicity of automatically applying optimistic synchronization in a parallelizing compiler instead of lock synchronization. For these applications, one of the lock synchronized versions always performs at least as well as the optimistically synchronized version. The problem is that the best lock synchronized version is different for different applications.

For Barnes-Hut, the Coarse Lock version performs better than both the Object Lock and Item Lock versions and consumes less memory than the Item Lock version. For String,

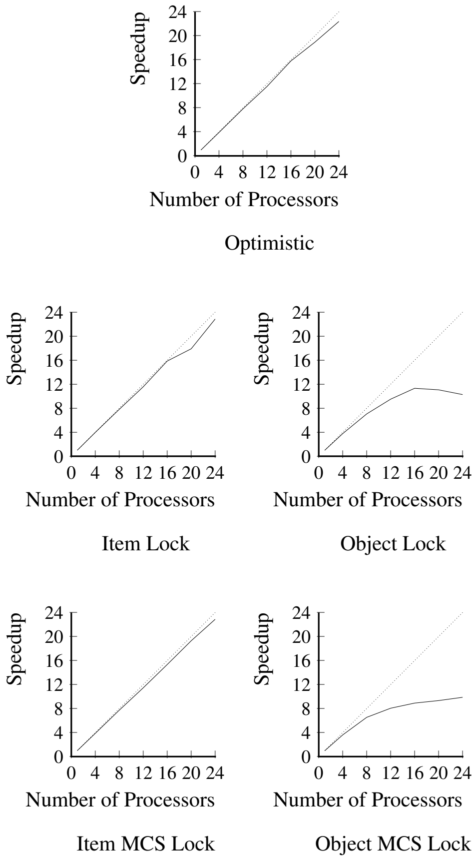

Fig. 8. Speedups for String

the Item Lock version consumes more memory than the Object Lock and Coarse Lock versions, but it eliminates the false exclusion and false contention problems that limit the performance of the other two versions. For Water, all versions incur significant memory and synchronization overheads. These results show that the compiler must manage a complex tradeoff between memory usage, lock overhead, false exclusion, and false contention if it is to choose the best lock granularity. We have been able to use dynamic feedback to successfully manage the tradeoff between the Object Lock and different Coarse Lock versions [Diniz and Rinard 1999]. The basic approach is to measure the performance of each version during a short sampling interval, then use the best version during a longer production interval. The generated code periodically resamples to dynamically adapt to changes in the best version. This approach requires the code to switch between the different versions dynamically. This dynamic switching is feasible for the Object Lock and Coarse Lock versions, in part because these versions all have the same association of locks with data and therefore the same memory layout. The fact that the Item Lock version has a different memory layout than the Object Lock and Coarse Lock versions would significantly complicate the extension of dynamic feedback to include the Item Lock version.

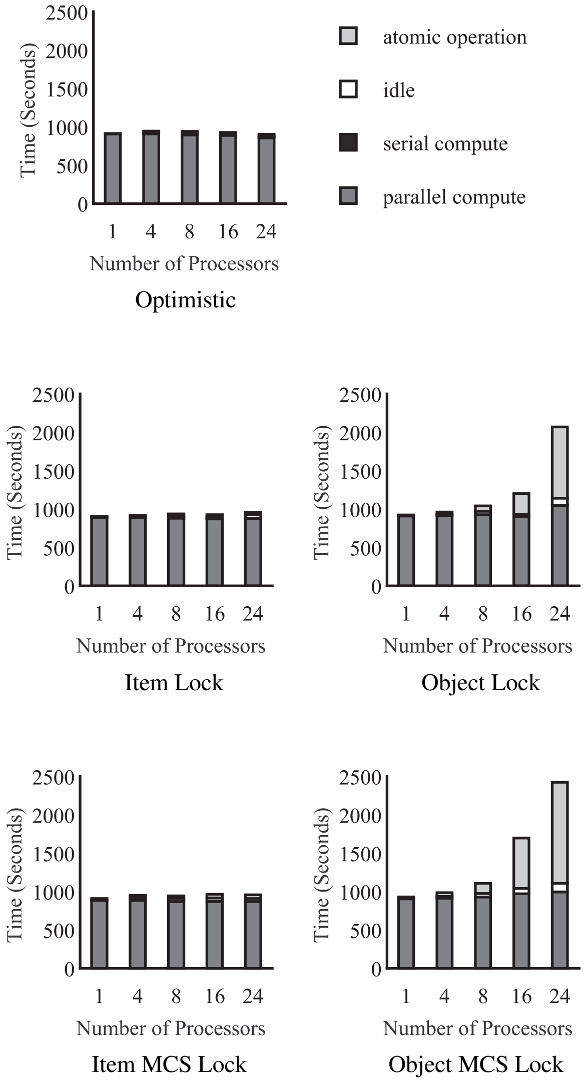

Fig. 9. Time Breakdowns for String

Version	Memory Usage (Mbytes)	Lock Overhead
Serial	0.43	-
Optimistic	0.43	-
Item Lock	0.72	40%
Object Lock	0.57	25%
Coarse Lock	0.58	25%
Item MCS Lock	0.72	40%
Object MCS Lock	0.57	25%
Coarse MCS Lock	0.58	25%

Table VIII. Lock Memory Overheads for Water

Version	Number of Lock Acquire/Release Pairs	Number of Optimistically Synchronized Updates
Optimistic	-	34,652,160
Item Lock	34,652,160	-
Object Lock	12,601,344	-
Coarse Lock	6,297,600	-
Item MCS Lock	34,652,160	-
Object MCS Lock	12,601,344	-
Coarse MCS Lock	6,297,600	-

Table IX. Number of Synchronization Operations for Water

	Processors						
Version	1	4	8	12	16	20	24
Serial	164.83	-	-	-	-	-	-
Optimistic	170.93	49.14	25.32	18.35	20.15	21.22	22.09
Item Lock	176.60	52.70	28.11	37.71	47.83	52.66	54.30
Object Lock	172.23	49.16	25.84	20.02	28.48	31.66	33.95
Coarse Lock	170.66	49.63	26.27	31.18	38.57	43.46	49.56
Item MCS Lock	190.10	56.68	34.38	40.75	47.65	50.12	52.67
Object MCS Lock	172.65	51.48	28.06	29.54	31.93	31.44	31.62
Coarse MCS Lock	169.77	51.30	28.44	29.58	32.49	32.30	32.75

Table X. Execution Times for Water (seconds)

Another drawback of using lock synchronization is that, as String illustrates, the version with the best performance may have significant memory overhead.

Based on these experimental results, we believe the major reasons for using optimistic synchronization in automatically parallelized applications are the absence of memory overhead and the elimination of the need to choose a lock granularity that works well for the application at hand. In general, we would not expect the best optimistically synchronized version of a parallel application to perform significantly better than the best lock synchronized version in a standard parallel computing environment.

8. RELATED WORK

The majority of existing research in optimistic synchronization has addressed problems associated with using blocking synchronization primitives such as locks in a multiprogrammed system. These problems include poor responsiveness, lock convoys, priority inversions, deadlock, and the need to reclaim locks held by failed processes [Herlihy 1993]. Optimistically synchronized data structures such as atomic queues allow programmers to avoid these problems. Such data structures were a key component of the extremely efficient

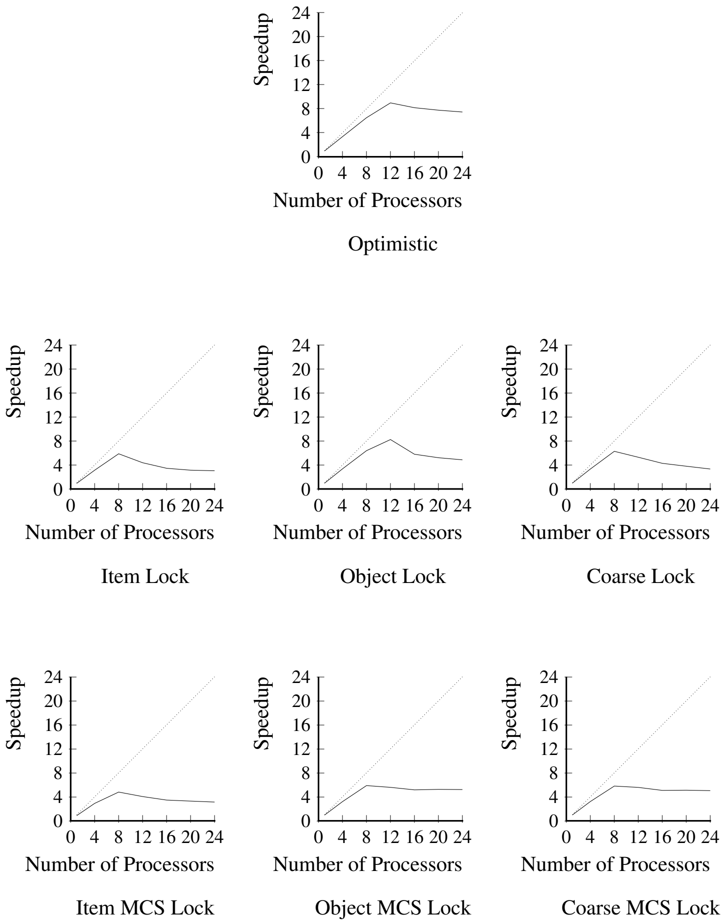

Fig. 10. Speedups for Water

Synthesis kernel [Massalin and Pu 1989], a complete operating system kernel built without blocking synchronization. Herlihy has developed a general methodology for implementing optimistically synchronized data structures [Herlihy 1993], and other researchers have implemented and measured the performance of several such data structures [Michael and Scott 1996]. Our research explores the use of optimistic synchronization in a different context (a parallelizing compiler for irregular, object-based computations) and for a different reason (to enable efficient fine-grain synchronization).

Advanced parallel machines such as Tera [Alverson et al. 1990] or Monsoon [Hicks et al. 1993] augment each word of memory with state bits. These machines provide a synchronizing read instruction that suspends until a state bit is set, then atomically reads the value in the word and clears the bit. The corresponding synchronizing write instruction writes a value into the word, then sets the bit. These machines provide exceptional support for fine-grain atomic operations — each operation simply uses a synchronizing read to obtain the current value, computes the new value, then uses a synchronizing write to write

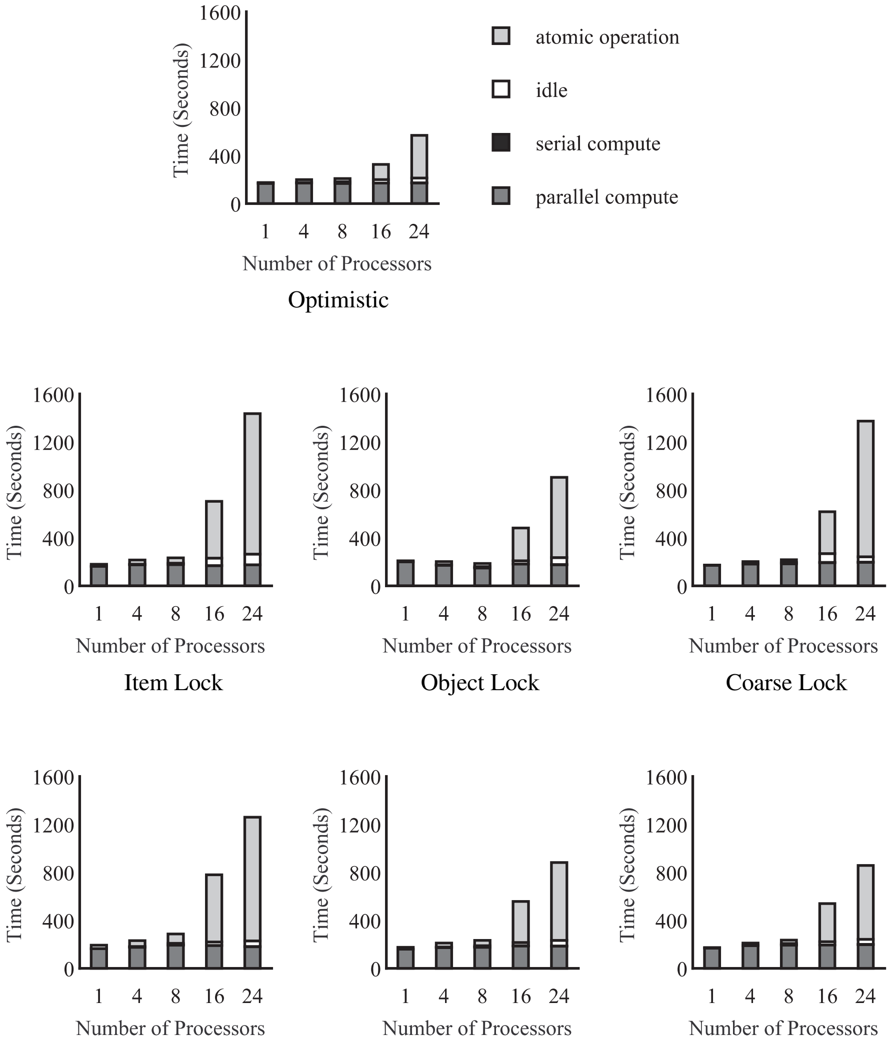

Fig. 11. Time Breakdowns for Water

the new value back into the word. There is no additional memory overhead (the lock bit for each word is already integrated into the machine) and no instruction overhead (the computation simply uses synchronizing read and write instructions in the place of normal read and write instructions).

Several software systems for these machines exploit this hardware support. The implementation of M-structures in the dataflow language Id uses the state bits to implement efficient, implicitly synchronized atomic operations [Barth et al. 1991]. The Tera compiler exploits the state bits to automatically parallelize loops that update indirectly accessed arrays. The generated code uses the bits to make the updates execute atomically. A simple modification to our code generation algorithm would enable our compiler to generate similarly synchronized code for computations that update dynamic, pointer-based data structures. Instead of an `ll` instruction, the generated code would use a synchronizing read. Instead of an `sc` instruction, the code would use a synchronizing write. Because the update would never fail, the code would omit the conditional branch that retries the operation in case of failure.

9. CONCLUSION

As shared-memory multiprocessors become the dominant commodity source of parallel computation, it will be important for parallelizing compilers to support irregular computations, including computations that access pointer-based data structures. Our experience with a parallelizing compiler for this class of applications indicates that their synchronization requirements differ significantly from those of traditional parallel computations. Instead of coarse-grain barrier synchronization, irregular computations require synchronization primitives that support efficient fine-grain atomic operations.

This paper presents our experience using optimistic synchronization to implement fine-grain atomic operations in automatically parallelized programs. For our set of benchmark applications, optimistic synchronization is clearly the superior choice. The optimistically synchronized versions have no memory overhead at all and the fastest lock synchronized version never performs significantly better than the optimistically synchronized version.

Acknowledgements

We would like to thank the anonymous referees for their many useful comments and suggestions. We would like to thank Silicon Graphics for the use of the Challenge XL multiprocessor. We would also like to thank Rohit Chandra, T.K. Lakshman, and Robert Kennedy for their help with the machines at Silicon Graphics. Pedro Diniz implemented the compilation system used to parallelize the applications.

REFERENCES

ADVE, S. AND GHARACHORLOO, K. 1996. Shared memory consistency models: a tutorial. *IEEE Comput. 29,* 12 (Dec.), 66–76.

ALVERSON, R., CALLAHAN, D., CUMMINGS, D., KOBLENZ, B., PORTERFIELD, A., AND SMITH, B. 1990. The Tera computer system. In *Proceedings of the 1990 ACM International Conference on Supercomputing*. Amsterdam, The Netherlands.

BACON, D., GRAHAM, S., AND SHARP, O. 1994. Compiler transformations for high-performance computing. *ACM Computing Surveys 26,* 4 (Dec.), 345–420.

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

BARTH, P., NIKHIL, R., AND ARVIND. 1991. M-structures: Extending a parallel, non-strict, functional language with state. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag, 538–568.

BODIN, F., BECKMAN, P., GANNON, D., GOTWALS, J., NARAYANA, S., SRINIVAS, S., AND WINNICKA, B. 1994. Sage++: An object-oriented toolkit and class library for building Fortran and C++ structuring tools. In *Proceedings of the Object-Oriented Numerics Conference*. Sunriver, Oregon.

DIGITAL EQUIPMENT CORPORATION. 1992. *Alpha Architecture Handbook*.

DINIZ, P. AND RINARD, M. 1997. Synchronization transformations for parallel computing. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*. ACM, New York, Paris, France, 187–200.

DINIZ, P. AND RINARD, M. 1998. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *Journal of Parallel and Distributed Computing 49,* 2 (Mar.), 2218–244.

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

GHARACHORLOO, K. 1996. Memory consistency models for shared memory multiprocessors. Ph.D. thesis, Dept. of Electrical Engineering, Stanford Univ., Stanford, Calif.

GRAHAM, S., KESSLER, P., AND MCKUSICK, M. 1982. gprof: a call graph execution profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*. ACM, New York, Boston, MA.

GREENWALD, M. AND CHERITON, D. 1996. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*.

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.

HERLIHY, M. 1993. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems 15,* 9 (Nov.), 745–770.

HERLIHY, M. AND MOSS, J. 1993. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*. San Diego, CA.

HICKS, J., CHIOU, D., ANG, B., AND ARVIND. 1993. Performance studies of Id on the Monsoon dataflow system. *Journal of Parallel and Distributed Computing 18,* 3 (July), 273–300.

JENSEN, E., HAGENSEN, G., AND BROUGHTON, J. 1987. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory. Nov.

KEMMERER, R. AND ECKMANN, S. 1985. UNISEX: a UNIx-based Symbolic EXecutor for Pascal. *Software—Practice and Experience 15,* 5 (May), 439–458.

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

KNUTH, D. 1971. An empirical study of FORTRAN programs. *Software—Practice and Experience 1*, 105–133.

MASSALIN, H. AND PU, C. 1989. Threads and input/output in the Synthesis kernel. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*. Litchfield Park, AZ.

MELLOR-CRUMMEY, J. AND SCOTT, M. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems 9,* 1 (Feb.), 21–65.

MICHAEL, M. AND SCOTT, M. 1995. Implementation of atomic primitives on distributed shared memory multiprocessors. In *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*. Raleigh, NC.

MICHAEL, M. AND SCOTT, M. 1996. Simple, fast and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on the Principles of Distributed Computing*. Philadelphia, PA.

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.

MOTOROLA, INCORPORATED. 1993. *PowerPC 601 RISC Microprocessor User's Manual*. IBM Microelectronics, Phoenix, AR.

RINARD, M. AND DINIZ, P. 1997. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems 19,* 6 (Nov.), 941–992.

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

STONE, J., STONE, H., HEIDELBERGHER, P., AND TUREK, J. 1993. Multiple reservations and the Oklahoma update. *IEEE Parallel and Distributed Technology 1,* 4 (Nov.), 58–71.

WEAVER, D. AND GERMOND, T. 1994. *The SPARC Architecture Manual*. SPARC International, Menlo Park, CA.

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.