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

Martin Rinard
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
martin@cs.ucsb.edu
http://www.cs.ucsb.edu/∼martin

Abstract

As shared-memory multiprocessors become the dominant commodity source of computation, parallelizing compilers must support mainstream computations that manipulate irregular, pointer-based data structures such as lists, trees and graphs. Our experience with a parallelizing compiler for this class of applications shows 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.

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.

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 programs. We have implemented two versions of the compiler. One version generates code that uses mutual exclusion locks to make operations execute atomically. The other version uses optimistic synchronization. This paper presents the first published algorithm that enables compilers to automatically generate optimistically synchronized parallel code. The presented experimental results indicate that optimistic synchronization is clearly the superior choice for our set of applications. Our results show that it can significantly reduce the memory consumption and improve the overall performance.

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 [3]. 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 that access dynamic, pointer-based data structures such as lists, trees and graphs. We have implemented a parallelizing compiler for this class of computations [25]. 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.

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 [7]. 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.

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 imposes no memory overhead and eliminates the use of heavyweight blocking synchronization primitives. It is therefore especially appropriate for implementing the fine-grain atomic operations characteristic of irregular parallel computations.

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 [25]. The compiler is designed to parallelize irregular computations that manipulate dynamic, pointer-based data structures such as lists, trees and graphs. Because it uses commutativity analysis [25] 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 and improve the performance of the automatically generated parallel code.

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

Although the algorithms presented in this paper are designed for commodity multiprocessors, they are also useful for more aggressive machines, such as Tera [1] or Monsoon [14], that 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 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). The analysis algorithms presented in this paper would enable the automatic generation of code that uses this efficient fine-grain synchronization mechanism for irregular computations that access dynamic, pointer-based data structures.

```
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
```

Figure 1: Atomic Increment Using `ll` and `sc`

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 the synchronization selection algorithm, which enables the compiler to generate optimistically synchronized code. Section 6 presents the experimental results. Section 7 discusses related work. We conclude in Section 8.

2 Optimistic Synchronization Primitives

Only recently have modern RISC processors provided efficient implementations of the hardware primitives required for optimistic synchronization. In this section we focus on hardware primitives available on MIPS processors such as the MIPS R4400 and R10000, although primitives that provide similar functionality are available in most modern processors [6, 15]. The two key instructions are the load linked (`ll`) and store conditional (`sc`) instructions [16], 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 [22].

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 [12]. 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 [13] would support optimistically synchronized atomic operations on larger objects, but no hardware implementation of this mechanism currently exists.

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

```
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);
}
```

Figure 2: Serial Graph Traversal

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 [24] 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.

```
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));
}
```

Figure 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));
}
```

Figure 4: Parallel Traversal With Optimistic Synchronization

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.

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. [1] 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.

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 [8]. 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 memory layouts, identical 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.

The experimental results in Section 6 suggest that neither of these two potential problems may occur frequently in practice. The compiler is able to correctly synchronize all of the benchmark applications using only optimistic synchronization. Furthermore, synchronizing at the coarser granularity of objects instead of using optimistic synchronization does not significantly improve the overall performance of any of our benchmark applications.

[1] In principle, the additional instructions required to address, acquire and release the lock may have a similar effect on the instruction cache. For optimistic synchronization, the conditional branches that retry failed updates are the only additional instructions.

5 The Synchronization Selection Algorithm

This section presents the synchronization selection algorithm. This algorithm chooses a synchronization mechanism for each operation, using optimistic synchronization whenever possible. We describe the model of computation for programs that the algorithm is designed to analyze, present the program representation that the synchronization selection algorithm uses, and discuss the requirements that the synchronization selection algorithm must satisfy to ensure that the generated parallel program executes correctly. We then present the algorithm in detail.

5.1 Model of Computation

The 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.

We next present some notation that we will use when we present the algorithms. 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 $\mathtt{op}$, the function receiverClass($\mathtt{op}$) returns the class of the receiver objects of $\mathtt{op}$. 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 $\mathtt{cl}$. No two classes share an instance variable — i.e., instanceVariables($\mathtt{cl}_1$) $\cap$ instanceVariables($\mathtt{cl}_2$) $= \emptyset$ if $\mathtt{cl}_1 \neq \mathtt{cl}_2$.

5.2 Program Representation

Synchronization selection takes place after the commutativity analysis algorithm has successfully parallelized a phase of the computation. The commutativity analysis algorithm extracts some information that the synchronization selection algorithm uses. Specifically, it produces the set of operations that the parallel phase may invoke and the set of instance variables that the phase may update [25]. 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 [18, 25].

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. [2]

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. [3]

5.3 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 [25]. 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 can not 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.

5.4 The Algorithm

Figure 5 presents the synchronization selection algorithm. It 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`.[4] 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, it first checks if it needs to synchronize the update at all. Because store instructions execute atomically, there is 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. Such updates often occur in code that initializes objects.[5]

If the update requires some form of synchronization, the algorithm checks if it can synchronize the update using optimistic synchronization. In principle, it should be possible to use optimistic synchronization for any update that reads an instance variable, computes a new value that does not depend on a different vari-

---

[2] 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.

[3] 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.

[4] 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.

[5] The algorithm allows two operations to update the same variable without synchronization as long as the new values do not depend on updated variables. Before the synchronization selection takes place, the commutativity testing algorithm has verified that the operations generate the same final result regardless of the order in which the updates execute. The sole responsibility of the synchronization selection algorithm is to ensure that the operations execute atomically.

able that another operation might update, then writes the new value back into the variable. In the MIPS R4400, however, it is illegal to perform a memory access between the `ll` and `sc` instructions. The algorithm therefore 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 $\mathtt{v}$ or $\mathtt{v}[\mathtt{exp}']$ denotes the original value of the variable, $\oplus$ is an arbitrary binary operator and $\mathtt{exp}$ 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.

```
synchronizationSelection(invokedOperations, updatedVariables,
                         updates, invocations)
  lockedClasses = ∅;
  lockedOperations = ∅;
  optimisticVariables = updatedVariables;
  for all op ∈ invokedOperations
    for all u ∈ updates(op)
      if (requiresSynchronization(u, updatedVariables) and
          not canUseOptimisticSynchronization(u, updatedVariables))
        optimisticVariables = optimisticVariables − variables(u);
        lockedClasses = lockedClasses ∪ {receiverClass(op)};
        lockedOperations = lockedOperations ∪ {op};
    for all i ∈ invocations(op)
      if (variables(i) ∩ updatedVariables ≠ ∅)
        optimisticVariables = optimisticVariables − variables(i);
        lockedClasses = lockedClasses ∪ {receiverClass(op)};
        lockedOperations = lockedOperations ∪ {op};
  return ⟨optimisticVariables, lockedClasses, lockedOperations⟩;

canUseOptimisticSynchronization(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 canUseOptimisticSynchronization(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 canUseOptimisticSynchronization(upd, updatedVariables);
  return false;

requiresSynchronization(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 if (exp) upd and
      variables(exp) ∩ updatedVariables = ∅)
    return requiresSynchronization(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 requiresSynchronization(upd, updatedVariables);
  return true;
```

Figure 5: Synchronization Selection Algorithm

To execute correctly, all accesses to a potentially updated variable must use the same synchronization mechanism. If any of the updates to a given variable must use lock synchronization, the algorithm deletes the variable from the set of optimistically synchronized variables. The compiler will augment the class of the receiver object with a lock and insert constructs that acquire and release the lock in all operations that access the variable.

The invocation expressions capture the remainder of the operation's accesses to updated variables. If an invocation expression contains 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 end result is that all accesses to a given optimistically synchronized variable occur only in optimistically synchronized updates to that variable.

5.5 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.

5.6 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 [25]. 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.

6 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 [4], a hierarchical N-body solver, String [10], which builds a velocity model of the geology between two oil wells, and Water [26], 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.

6.1 Methodology

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. 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 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 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 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 [7].

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 trade off, 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 trade off 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 Optimistic, 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.

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.

The Item Lock, Object Lock, and Coarse Lock versions all use 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 [11]. The release simply clears the value. When the implementation attempts to acquire a lock that is not free or the instruction sequence that implements the test and set fails, it must retry until it acquires the lock. The implementation uses exponential backoff to eliminate immediate, repeated attempts to acquire an unavailable lock [2, 21]. 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 [17]. The presented results therefore overstate the efficiency of the versions that use locks.

6.2 Cost Of Basic Operations

Table 1 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 the cached versions, with the optimistically synchronized update executing significantly faster than the lock synchronized update. 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	3.251
Optimistic Synchronization	0.187	3.352
Lock Synchronization	0.278	3.456

Table 1: Measured Execution Times for One Update

6.3 Barnes-Hut

We start our discussion of Barnes-Hut by analyzing the memory overhead of using locks. Table 2 presents the *memory usage* for each version — the amount of memory used to store objects during the execution of the program.[6] The *lock overhead* is the percentage of memory used to hold locks. 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.

Version	Memory Usage (Mbytes)	Lock Overhead
Serial	40	-
Optimistic	40	-
Item Lock	49	18%
Object Lock	45	11%
Coarse Lock	45	11%

Table 2: 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	-

Table 3: Number of Synchronization Operations for Barnes-Hut

Table 3 presents the number of synchronization operations performed by the different versions. The Optimistic and Item Lock

[6] 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 reader should bear in mind that the numbers in Table 2 are for a realistic but small data set — 16,384 particles. Larger data sets would dramatically increase the memory usage.

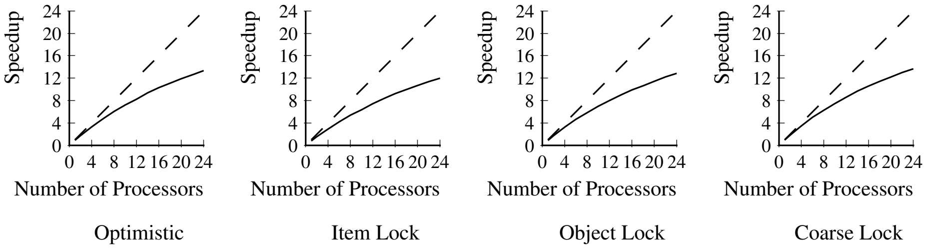

Figure 6: Speedups for Barnes-Hut

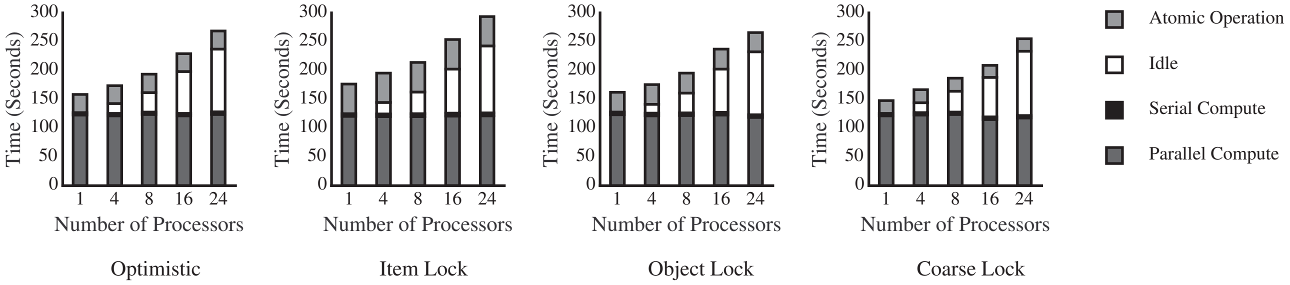

Figure 7: Time Breakdowns for Barnes-Hut

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 4 presents the running times for Barnes-Hut as a function of the number of processors executing the computation. Figure 6 presents the corresponding speedup curves.[7] The application scales reasonably well — the speedup over the Serial version is above 12 out of 24 processors for all versions.

	Processors						
Version	1	4	8	12	16	20	24
Serial	140.75	-	-	-	-	-	-
Optimistic	154.49	42.51	23.19	17.07	13.64	11.82	10.54
Item Lock	175.38	48.25	26.05	18.89	15.22	13.16	11.73
Object Lock	159.54	43.10	24.12	17.57	14.21	12.30	10.94
Coarse Lock	147.35	40.32	22.44	16.36	13.28	11.52	10.29

Table 4: Execution Times for Barnes-Hut (seconds)

We used program counter sampling [9, 19] 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:

- **Atomic Operation:** The amount of time spent executing operations that require synchronization to execute atomically or operations that contain lock constructs. [8] Performance problems caused by contention for locks or retried updates show up as increases in this component of the execution time.
- **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.[9] 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

[7] The speedup 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, and, like the serial version of Water, performs slightly better than a highly optimized version written in C [25]. The serial version of String performs slightly worse than the corresponding C version.

[8] 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 0.3% of its execution time executing operations that contain lifted lock constructs; Water always spends less than 1.3% of its execution time executing operations that contain lifted lock constructs.

[9] 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. 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.

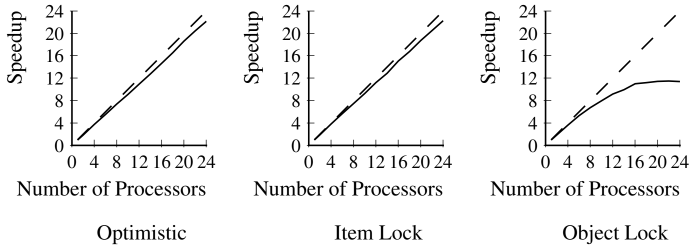

Figure 8: Speedups for String

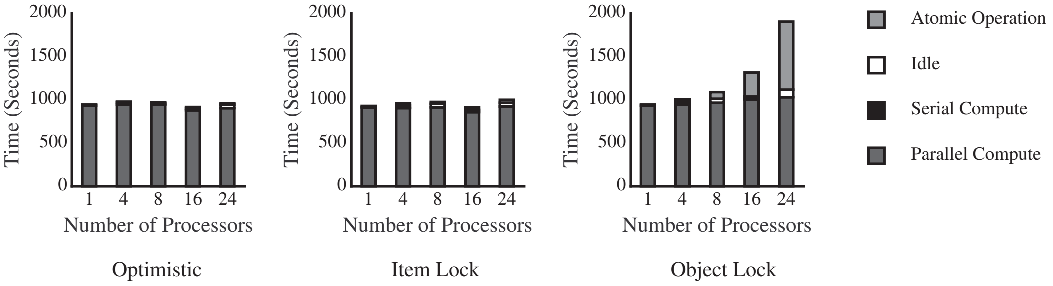

Figure 9: Time Breakdowns for String

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.[10] 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.

6.4 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 5 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 6 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 7 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.

[10] 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 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.

Version	Memory Usage (Mbytes)	Lock Overhead
Serial	3.6	-
Optimistic	3.6	-
Item Lock	4.9	27%
Object Lock	3.6	0%

Table 5: 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	-

Table 6: Number of Synchronization Operations for String

	Processors						
Version	1	4	8	12	16	20	24
Serial	891.74	-	-	-	-	-	-
Optimistic	936.21	237.44	120.78	81.47	60.92	47.90	40.24
Item Lock	917.57	232.61	119.37	79.51	59.17	47.62	40.06
Object Lock	933.06	245.79	132.16	97.29	81.02	78.02	78.24

Table 7: Execution Times for String (seconds)

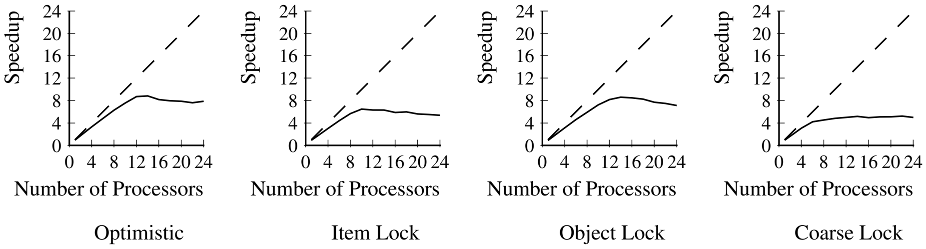

Figure 10: Speedups for Water

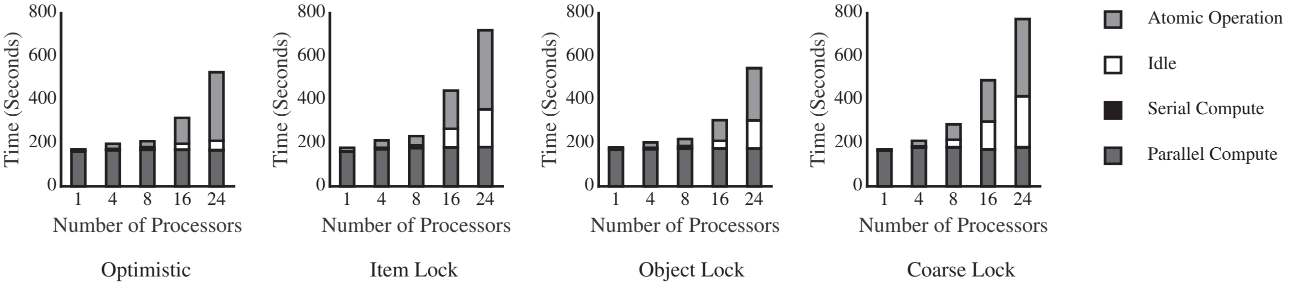

Figure 11: Time Breakdowns for Water

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.

6.5 Water

Table 8 presents the lock memory overheads for Water. The use of locks significantly increases the amount of memory that the program consumes. Table 9 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 10 presents the execution times; Figure 10 presents the corresponding speedup curves. The application scales reasonably well to 12 processors, then the performance levels off. The time breakdowns in Figure 11 show that increased Atomic Operation and Idle times cause the poor performance. Because the Item Lock, Object Lock and Coarse Lock versions repeatedly update the same objects, they repeatedly attempt to acquire the same locks. The resulting contention significantly reduces the performance. In the Optimistic version, this access pattern causes the computation to repeatedly retry failed optimistically synchronized updates.

Version	Memory Usage (Mbytes)	Lock Overhead
Serial	0.76	-
Optimistic	0.76	-
Item Lock	1.27	40%
Object Lock	1.00	24%
Coarse Lock	1.00	24%

Table 8: 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	-

Table 9: Number of Synchronization Operations for Water

	Processors						
Version	1	4	8	12	16	20	24
Serial	158.86	-	-	-	-	-	-
Optimistic	169.76	47.96	25.22	18.23	19.42	20.15	20.16
Item Lock	175.47	52.47	27.97	25.15	27.04	28.30	29.54
Object Lock	179.27	50.65	26.85	19.41	18.72	20.61	22.29
Coarse Lock	171.16	51.58	35.03	31.78	32.08	31.12	31.82

Table 10: Execution Times for Water (seconds)

The Optimistic version has a significantly smaller Idle time than the versions with lock synchronization. We attribute this difference to the fact that the Optimistic version immediately retries failed updates. The use of exponential backoff in the Optimistic version to increase the time between retried updates does not improve the overall performance but does shift time from the Atomic Operation component to the Idle component.

6.6 Discussion

The experimental results demonstrate that optimistic synchronization is clearly the superior choice. 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 instead of lock synchronization. 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, 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 trade off between memory usage, lock overhead, false exclusion, and false contention if it is to choose the best lock granularity. Given the dynamic nature of the factors that control this trade off, we believe that it will be difficult for compilers to successfully manage this trade off. And, as String illustrates, the version with the best performance may have significant memory overhead.

7 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 and deadlock [12]. 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 Synthesis kernel [20], a complete operating system kernel built without blocking synchronization. Herlihy has developed a general methodology for implementing optimistically synchronized data structures [12], and other researchers have implemented and measured the performance of several such data structures [23]. 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).

Several software systems use the state bits and synchronizing read and write instructions on machines such as Tera and Monsoon to support efficient fine-grain synchronization. The implementation of M-structures in the dataflow language Id uses the state bits to support efficient, implicitly synchronized atomic operations [5]. 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 retried the operation in case of failure.

8 Conclusion

As shared-memory multiprocessors become the dominant commodity source of parallel computation, it will be important for parallelizing compilers to support irregular computations that access dynamic, 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. It presents the first published algorithm that enables compilers to automatically generate optimistically synchronized parallel code. The presented experimental results show that optimistic synchronization is clearly the superior choice for our set of benchmark applications. 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. These results indicate that optimistic synchronization may play a key enabling role in the successful automatic parallelization of irregular computations.

Acknowledgements

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

[1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[2] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[3] D. Bacon, S. Graham, and O. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4), December 1994.

[4] J. Barnes and P. Hut. A hierarchical O(NlogN) force-calculation algorithm. *Nature*, pages 446–449, December 1976.

[5] P. Barth, R. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *Proceedings of the Fifth ACM Conference on Functional Programming Languages and Computer Architecture*, pages 538–568. Springer-Verlag, August 1991.

[6] Digital Equipment Corporation. *Alpha Architecture Handbook*. 1992.

[7] P. Diniz and M. Rinard. Synchronization transformations for parallel computing. In *Proceedings of the Twenty-fourth Annual ACM Symposium on the Principles of Programming Languages*, Paris, France, January 1997.

[8] K. Gharachorloo. *Memory Consistency Models for Shared Memory Multiprocessors*. PhD thesis, Stanford, CA, 1996.

[9] 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.

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

[11] J. Heinrich. *MIPS R4000 Microprocessor User's Manual*. Prentice-Hall, 1993.

[12] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(9):745–770, November 1993.

[13] M. Herlihy and J. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[14] J. Hicks, D. Chiou, B. Ang, and Arvind. Performance studies of Id on the Monsoon dataflow system. *Journal of Parallel and Distributed Computing*, 18(3):273–300, July 1993.

[15] Motorola Incorporated. *PowerPC 601 RISC Microprocessor User's Manual*. IBM Microelectronics, Phoenix, AR, 1993.

[16] E. Jensen, G. Hagensen, and J. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, November 1987.

[17] 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.

[18] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

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

[20] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, Litchfield Park, AZ, December 1989.

[21] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[22] M. Michael and M. Scott. Implementation of atomic primitives on distributed shared memory multiprocessors. In *Proceedings of the First IEEE Symposium on High-Performance Computer Architecture*, Raleigh, NC, January 1995.

[23] M. Michael and M. Scott. 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, May 1996.

[24] 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, June 1990.

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

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

A Other Lock Implementations

In addition to the inlined lock implementation used for the experimental results in Section 6 (we call this implementation the Inline implementation), we also ran the benchmark applications with three other lock implementations:

- **Eager:** The implementation does not use exponential backoff between attempts to acquire a lock. After every failed attempt to acquire a lock, the implementation immediately reexecutes the instruction sequence that attempts to acquire the lock.
- **Mapped:** The run time system provides a fixed number of locks (in our experiments there are 262,144 locks). Instead of augmenting updated objects with locks, the compiler generates code that maps the virtual address of each object or item to one of the locks in the run time system. Each acquire or release construct is implemented using the same inlined code sequence as in the Inline implementation. The advantage of this implementation is that there is a fixed amount of memory dedicated to locks regardless of how much data the program allocates. The primary disadvantages are that multiple objects or items may be mapped to the same lock, that locks are located on different cache lines than the associated objects or items, and that the mapping function may generate poor lock utilization for large objects.
- **Standard:** The compiler augments updated objects with the standard IRIX 6.2 locks; the generated code uses the standard primitives in the IRIX 6.2 libraries to acquire and release the locks. These primitives implement blocking locks — the acquire primitive temporarily relinquishes the processor if it cannot acquire the lock in a reasonable amount of time.[11]

Table 11 presents the measured execution times for a single atomic update with the different lock implementations.

Version	Execution Time For One Cached Update (microseconds)	Execution Time For One Uncached Update (microseconds)
Inline Lock	0.28	3.46
Eager Lock	0.28	3.46
Mapped Lock	0.33	6.71
Standard Lock	0.79	7.02

Table 11: Measured Execution Times for One Locked Update

We present the performance impact of the different lock implementations on our benchmark applications using the *normalized execution time*, which is the execution time of the Eager, Mapped, or Standard version divided by the execution time of the Inline version. Table 12 presents the mean (over all numbers of processors) normalized execution times for the different versions of the different applications.

Application	Version	Lock Implementation		
		Eager Lock	Mapped Lock	Standard Lock
Barnes-Hut	Item Lock	0.99	1.01	1.26
	Object Lock	1.00	1.01	1.13
	Coarse Lock	1.00	0.99	1.01
String	Item Lock	1.01	1.03	1.09
	Object Lock	1.02	1.01	1.16
Water	Item Lock	1.40	1.56	1.81
	Object Lock	1.19	1.94	1.74
	Coarse Lock	0.90	1.42	1.22

Table 12: Mean Normalized Execution Times

Finally, we ran a version of each benchmark application that uses optimistic synchronization with exponential backoff between retried updates. The use of exponential backoff slightly increases the execution time for one cached update (from 0.189 microseconds to 0.208 microseconds), but has little impact on the overall performance. The mean normalized execution times of the versions with exponential backoff relative to the versions that immediately retry failed updates are 1.01 for Barnes-Hut, 0.99 for String, and 0.99 for Water.

[11] The type of the lock is `ulock_t`, `ussetlock` acquires a lock and `usunsetlock` releases a lock