Lock Coarsening: Eliminating Lock Overhead in Automatically Parallelized Object-Based Programs

Pedro Diniz* and Martin Rinard†
Department of Computer Science, Engr I Bldg, Room 2106
University of California, Santa Barbara
Santa Barbara, CA 93106-5110
{pedro,martin}@cs.ucsb.edu
Phone: (805) 893-2777
Fax: (805) 893-8553

Abstract

Atomic operations are a key primitive in parallel computing systems. The standard implementation mechanism for atomic operations uses mutual exclusion locks. In an object-based programming system the natural granularity is to give each object its own lock. Each operation can then make its execution atomic by acquiring and releasing the lock for the object that it accesses. But this fine lock granularity may have high synchronization overhead because it maximizes the number of executed acquire and release constructs. To achieve good performance it may be necessary to reduce the overhead by coarsening the granularity at which the computation locks objects.

In this paper we describe a static analysis technique — lock coarsening — designed to automatically increase the lock granularity in object-based programs with atomic operations. We have implemented this technique in the context of a parallelizing compiler for irregular, object-based programs and used it to improve the generated parallel code. Experiments with two automatically parallelized applications show these algorithms to be effective in reducing the lock overhead to negligible levels.

1 Introduction

Atomic operations are an important primitive in the design and implementation of parallel systems. Operations are typically made atomic by associating mutual exclusion locks with the data that they access. An atomic operation first acquires the lock for the data that it manipulates, accesses the data, then releases the lock.

We have implemented a compiler designed to automatically parallelize object-based computations that manipulate irregular, pointer-based data structures. This compiler uses commutativity analysis [12] as its primary analysis paradigm. For the generated program to execute correctly, each operation in the generated parallel code must execute atomically. The automatically generated code therefore contains mutual exclusion locks and constructs that acquire and release these locks.

We have found that the granularity at which the generated parallel computation locks objects can have a significant impact on the overall performance. The natural lock granularity is to give each object its own lock and generate code in which each operation acquires and releases the lock for

*Sponsored by the PRAXIS XXI program administrated by Portugal's JNICT – Junta Nacional de Investigação Científica e Tecnológica, and holds a Fulbright travel grant.
†Supported in part by an Alfred P. Sloan Research Fellowship.

the object that it accesses. Our experimental results indicate, however, that locking objects at this fine granularity may introduce enough overhead to significantly degrade the overall performance. To achieve good performance we have found it necessary to coarsen the lock granularity to reduce the amount of lock overhead.

This paper presents the analysis algorithms and program transformations that the compiler uses to automatically coarsen the lock granularity. We have implemented these algorithms in the context of a parallelizing compiler for object-based programs [12]; they are used to reduce lock overhead in the generated parallel code. This paper also presents experimental results that characterize the performance impact of using the lock coarsening algorithms in the compiler.

The results show the algorithms to be effective in reducing the lock overhead to negligible levels. They also show that an overly aggressive lock coarsening algorithm can actually impair the performance by artificially increasing lock contention (lock contention occurs when two processors attempt to acquire the same lock at the same time). Lock contention is a potentially serious problem because it can serialize large sections of the parallel computation. A successful compiler must therefore negotiate a trade off between reducing lock overhead and increasing lock contention.

Although we have developed our techniques in the context of a parallelizing compiler for object-based languages, we expect them to be useful in other contexts. Many explicitly parallel programming languages, for example, provide atomic operations as part of the programming model. Compilers for these languages should be able to use lock coarsening techniques to reduce the overhead of implementing the atomic operations. Many database systems use locks to implement atomic transactions; these systems may be able to use lock coarsening techniques to drive down the lock overhead associated with implementing atomic transactions.

This paper makes the following contributions:

- It introduces two techniques for reducing lock overhead: data lock coarsening and computation lock coarsening.
- It presents novel and practical lock coarsening algorithms that a compiler can use to reduce the lock overhead.
- It presents experimental results that characterize the performance impact of the lock coarsening algorithms on several automatically parallelized applications. These performance results show that, for these applications, the algorithms can effectively reduce the lock overhead to negligible levels.

The rest of this paper is structured as follows. In Section 2 we discuss the basic issues associated with using mutual exclusion locks to implement atomic operations. Section 3 presents an example that illustrates how coarsening the lock granularity can reduce the lock overhead. Section 4 describes the kinds of programs that the lock coarsening algorithms are designed to optimize. In Sections 5 and 6 we present the lock coarsening algorithms and transformations. In Section 7 we present experimental results that characterize the impact of the lock coarsening algorithms on the overall performance of two automatically parallelized applications.

2 Basic Issues in Lock Coarsening

The lock coarsening algorithms deal with two basic sources of performance loss: lock overhead and lock contention.

- **Lock Overhead:** Acquiring or releasing a lock generates overhead; the goal of the algorithms is to reduce this overhead by applying transformations that make the computation execute fewer acquire and release constructs.

- **Lock Contention:** Lock contention occurs whenever one processor attempts to acquire a lock held by another processor. In this case the first processor must wait until the second processor releases the lock; the first processor performs no useful computation during time it spends waiting for the lock to be released. Increased lock contention therefore reduces the amount of available parallelism.

All of the transformations that the algorithms apply to reduce the lock overhead have the potential to increase the lock contention. The algorithms must therefore negotiate a tradeoff between the lock overhead and the lock contention.

The algorithms apply two lock coarsening techniques: data lock coarsening and computation lock coarsening.

- **Data Lock Coarsening:** Data lock coarsening is a technique in which the compiler associates one lock with multiple objects that tend to be accessed together. The compiler then transforms computations that manipulate one or more of the objects. Each transformed computation acquires the lock, performs the manipulations, then releases the lock. The original computation, of course, acquired and released a lock every time it manipulated any one of the objects.

  Data lock coarsening may improve the computation in two ways. First, it may reduce the number of executed acquire and release constructs — it enables computations to access multiple objects with the overhead of only a single acquire construct and a single release construct. Second, it may reduce the number of locks that the computation requires to execute successfully — giving multiple objects the same lock may reduce the number of allocated locks.

  An overly aggressive data lock coarsening algorithm may introduce *false contention*. False contention occurs when two operations attempt to acquire the same lock even though they access different objects.

- **Computation Lock Coarsening:** Consider a computation that repeatedly acquires and releases the same lock. This may happen, for example, if a computation performs multiple operations on the same object or on objects that have all been given the same lock by the data lock coasrsening algorithm. The computation lock coarsening algorithm analyzes the program to find such computations. It then transforms the computations to acquire the lock, perform the operations without synchronization, then release the lock. This transformation may significantly reduce the number of executed acquire and release constructs.

  An overly aggressive computation lock coarsening algorithm may introduce *false exclusion*. False exclusion may occur when a computation holds a lock for an extended period of time during which it does not access one of the lock's objects. If another computation attempts to acquire the lock (so that it may access one of the lock's objects), it must wait for the first computation to release the lock even though the first computation is not actively accessing any of the lock's objects. False exclusion may therefore reduce the performance by decreasing the amount of available concurrency.

There is a potential interaction between lock coarsening and concurrency generation. Each acquire and release pair defines a mutual exclusion region — the region between the acquire and release constructs. The lock coarsening transformations replace multiple mutual exclusion regions with a single coarsened mutual exclusion region. Note that the coarsened mutual exclusion region may contain the execution of multiple operations. Furthermore, the transformations eliminate all of the synchronization constructs in these operations.

To ensure that all of the operations within a given coarsened mutual exclusion region execute atomically with respect to each other, the algorithms require that the entire computation within the coarsened region execute sequentially. There are two options: refusing to apply the lock coarsening transformation if the coarsened region would contain a concurrency generation construct, or removing all of the concurrency generation constructs within the coarsened region. The current transformations apply the first option.

There may be a concern that the transformations will introduce deadlock. As explained in Sections 5 and 6, the lock coarsening transformations never cause a program to deadlock.

3 Example

In this section we provide an example, inspired by the Barnes-Hut application in Section 7, that illustrates both kinds of lock coarsening. The example computation manipulates an array of pointers to nodes. Each node represents its state with a vector and a count. The example also stores a set of values in a binary search tree. The point of the computation is to scale every node by all of the values in the tree that fall within a certain range. The computation finds all of the values that fall within the range by traversing the binary search tree that stores the values.

Figure 1 contains the parallel C++ code for this example. Each class is augmented with a mutual exclusion lock; the parallel code uses this lock to make operations on objects of that class atomic. If an operation modifies its receiver object[1], it first acquires the receiver's lock, performs the modification, then releases the lock.

The computation starts at the nodeSet::scaleNodeSet method. This method invokes the node::traverse method in parallel for each node in the array; the parallel for loop makes the loop iterations execute concurrently. Note that all of the invocations of the node::traverse method may not be independent — if two array elements point to the same node, the corresponding loop iterations will modify the same node. The operations in the loop iterations must therefore execute atomically for the computation to execute correctly.

The node::traverse method traverses the binary search tree to find all of the values in the range [min,max]. Whenever it finds a value inside the range, it invokes the node::scaleNode method to scale the node by the value. The node::scaleNode method scales a node by incrementing the count of applied scale operations, then invoking the vector::scaleVector method to scale the vector stored in the node.

3.1 Data Lock Coarsening in the Example

An examination of the parallel code in Figure 1 reveals that the computation acquires and releases two locks every time it scales a node: the lock in the node object (the node::scaleNode method acquires and releases this lock) and the lock in the nested vector object inside the node object (the vector::scaleVector method acquires and releases this lock).

It is possible to eliminate the acquisition and release of the lock in the nested vector object by coarsening the lock granularity as follows. Instead of giving each nested vector object its own lock, the compiler can use the lock in the enclosing node object to make operations on the nested vector object atomic. Figure 2 contains the transformed code that locks the objects at this granularity.

[1] Programs that use the object-based programming paradigm structure the computation as operations on objects. Each operation has a single receiver object; as described in Section 4 this object is the object that the operation manipulates.

```
const int NDIM 3;
class vector {
  lock mutex;
  double value[NDIM];
public:
  void scaleVector(double s){
    mutex.acquire();
    for(int i=0; i < NDIM; i++) value[i] *= s;
    mutex.release();
  }
};

class tree {
public:
  double x;
  tree *left;
  tree *right;
};

class node {
  lock mutex;
public:
  int count;
  vector value;
  void scaleNode(double s);
  void traverse(tree *t, double min, double max);
};

class nodeSet {
  int size;
  node **elements;
public:
  void scaleNodeSet(tree *t, double min, double max);
};

void node::scaleNode(double s){
  mutex.acquire();
  count++;
  mutex.release();
  value.scaleVector(s);
}

void node::traverse(tree *t, double min, double max){
  if((min ≤ t->x) && (t->x < max)) scaleNode(t->x);
  if(min ≤ t->x) traverse(t->left,min,max);
  if(t->x ≤ max) traverse(t->right,min,max);
}

void nodeSet::scaleNodeSet(tree *t, double min, double max){
  scaleNodeSet(t, min, max);
  wait();
}

void nodeSet::scaleNodeSet(tree *t, double min, double max){
  parallel for(int i = 0; i < size; i++){
    elements[i]->traverse(t, min, max);
  }
}
```

Figure 1: Parallel Node Scaling Example

The compiler generates a new version of the vector::scaleVector method (this new version is called vector::syncFree_scaleVector) that does not acquire the lock. It invokes this new version from within the node::scaleNode method and transforms the code so that it holds the node's lock during the execution of the vector::syncFree_scaleVector method.

```
void vector::syncFree_scaleVector(double s){
  for(int i=0; i < NDIM; i++) value[i] *= s;
}
void node::scaleNode(double s){
  mutex.acquire();
  count++;
  vector.syncFree_scaleVector(s);
  mutex.release();
}
```

Figure 2: Data Lock Coarsening Example

To legally perform this transformation, the compiler must ensure that every thread that executes a vector operation acquires the corresponding node lock before it executes the operation. An examination of the code shows that it satisfies this constraint.

This transformation illustrates the utility of data lock coarsening. It reduces the number of executed locking constructs by a factor of two because it eliminates the acquire/release pair in the vector::scaleVector method. The compiler can also omit the mutual exclusion lock declaration in the vector class because none of the methods in the parallel computation acquire or release the lock.

3.2 Computation Lock Coarsening in the Example

The example also contains an opportunity for computation lock coarsening. Consider the subcomputation generated as a result of executing a node::traverse method. This subcomputation periodically executes node::scaleNode methods, which acquire and release the node's mutual exclusion lock. All of these executions acquire and release the same mutual exclusion lock. In fact, all of the operations in the entire subcomputation that acquire any lock acquire the same lock: the lock in the receiver object of the original node::traverse operation. It is therefore possible to coarsen the lock granularity by acquiring the lock once at the beginning of the subcomputation, then holding it until the subcomputation finishes. This transformation eliminates all of the lock constructs except the initial acquire and the final release. Figure 3 shows the transformed code.

This example also illustrates the potential for false exclusion. Consider the original program in Figure 1. This program only holds the node's lock when it is actually updating the node. The transformed code in Figure 3 holds the lock for the entire traversal. If two traversals on the same node seldom update the node, they can execute mostly in parallel in the original version of the code. In the coarsened version they will execute serially. As we will see in Section 7, this kind of serialization may significantly impair the performance of the parallel computation. The compiler must therefore ensure that its lock coarsening policy does not introduce a significant amount of false exclusion.

```
void node::syncFree_scaleNode(double s){
  count++;
  vector.syncFree_scaleVector(s);
}
void node::syncFree_traverse(tree *t, double min, double max){
  if((min ≤ t->x) && (t->x < max)) syncFree_scaleNode(t->x);
  if(min ≤ t->x)
      syncFree_traverse(t->left,min,max);
  if(t->x ≤ max)
      syncFree_traverse(t->right,min,max);
}
void node::traverse(tree *t, double min, double max){
  mutex.acquire();
  syncFree_traverse(t, min, max);
  mutex.release();
}
```

Figure 3: Computation Lock Coarsening Example

4 Model of Computation

Before presenting the lock coarsening algorithms, we discuss the kinds of programs that they are designed to optimize. First, the algorithms are designed for pure object-based programs. Such programs structure the computation as operations on objects. Each object implements its state using a set of instance variables. Each instance variable can be either a nested object or a primitive type from the underlying language such as an integer, a double, or a pointer to an object. Each object has a mutual exclusion lock that exports an acquire construct and a release construct. Once a processor has successfully executed an acquire construct on a given lock, all other processors that attempt to acquire that lock block until the first processor executes a release construct. Operations on the object use its lock to ensure that they execute atomically.

Programmers define operations by writing methods. Each operation corresponds to a method invocation: to execute an operation, the machine executes the code in the corresponding method. Each operation has a receiver object and several parameters. When an operation executes it can access the parameters, invoke other operations or access the instance variables of the receiver. There are several restrictions on instance variable access. An operation cannot directly access an instance variable of a nested object — it can only access the variable indirectly by invoking an operation that has the nested object as the receiver. If an instance variable is declared in a parent class from which the receiver's class inherits, the operation can not directly access the instance variable — it can only access the variable indirectly by invoking an operation whose receiver's class is the parent class.

The computation consists of a sequence of alternating serial and parallel phases. Within a parallel phase the computation uses constructs such as parallel loops to create operations that execute concurrently. The only synchronization consists of the mutual exclusion synchronization required to make the operations atomic and the barrier synchronization at the end of a parallel phase.

If an operation accesses an instance variable that may be modified during the parallel phase, it uses the lock in the instance variable's object to make its access atomic. Before the operation executes its first access to the instance variable, it acquires the object's lock. It releases the lock after it completes the last access.

We extend the model for read-only data as follows. If no operation in the parallel phase modifies an instance variable, any operation in the phase (including operations whose receiver is not the object containing the instance variable) can access the variable without synchronization.

5 Data Lock Coarsening

The data lock coarsening algorithm starts with a computation in which each object has its own mutual exclusion lock. The basic idea of the data lock coarsening algorithm is to increase the lock granularity by giving multiple objects the same lock. Before the computation accesses any one of these objects, it first acquires the lock. Once it has acquired the lock it can access any other object that has the same lock with no further synchronization.

The data lock coarsening algorithm must first choose which objects should have the same lock. The current policy attempts to give nested objects the same lock as their enclosing object. The algorithm must then determine if it can transform the entire parallel computation to lock objects at the new granularity. The key issues are to determine statically the new lock that each operation must hold, to make sure that in the generated code each operation actually holds that lock, and to ensure that no part of the computation locks objects at the old granularity. The algorithm uses two key data structures: the class nesting graph and the call graph.

5.1 The Class Nesting Graph

The class nesting graph represents the nesting structure of the classes. The nodes of the graph are the classes used in the computation. There is a directed edge from a class c_1 to a class c_2 if c_2 inherits from c_1 or if c_2 has a nested object of class c_1. The class nesting graph is acyclic — a cyclic class nesting graph would define an object of either infinite or zero size. The algorithm constructs the class nesting graph by traversing the class declarations. Figure 4 shows the class nesting graph for the example code in Figure 1.

5.2 The Call Graph

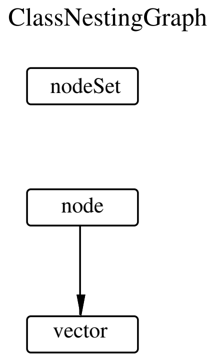
The nodes of the call graph are the methods that the computation may execute. There is a directed edge from method m_1 to method m_2 in the call graph if m_1 invokes m_2. To simplify the construction of the call graph the current version of the compiler does not support function pointers or dynamic method dispatch. Figure 5 shows the call graph for the example code in Figure 1.

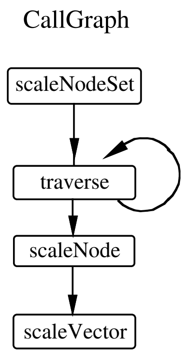
5.3 Connecting the Class Nesting Graph and the Call Graph

The compiler connects the nodes in the class nesting graph to the nodes in the call graph as follows: there is an undirected edge between a class c and a method m if the receiver of m is an object of class c. These edges allow the compiler to quickly compute which methods access objects of a given class. Figure 6 shows the class nesting graph and the call graph corresponding to the example code in Figure 1. We have represented the undirected edges between the two graphs by dashed lines.

5.4 Primitives

The lock coarsening algorithms use the following primitives:





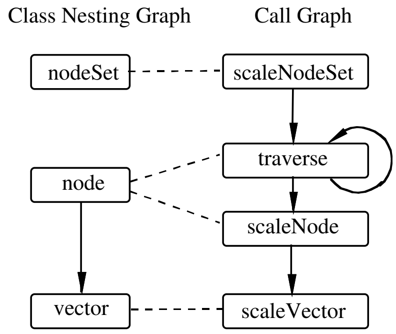


Figure 4: Class Nesting Graph for the Example Code

Figure 5: Call Graph for the Example Code

Figure 6: Relationship Between the Class Nesting Graph and Call Graph for the Example Code

Definition 1 *Given a class c,* `methods`(c) *is the set of methods defined in c — i.e. the set of methods whose receivers are objects of class c. The compiler computes* `methods`(c) *by following the edges from c in the class nesting graph to the methods in the call graph.*

Definition 2 *Given a method m,* `receiverClass`(m) *is the class that m is defined in — i.e. the class of the receiver objects of m. The compiler computes* `receiverClass`(m) *by following the edge from m in the call graph back to the class nesting graph.*

Definition 3 *m is a c-method if $c =$* `receiverClass`(m)*.*

Definition 4 *Given a method m,* `directlyInvokedMethods`(m) *is the set of methods that m may directly invoke.* `invokedMethods`(m) *is the set of methods that may be directly or indirectly invoked as a result of executing m. Neither set includes m unless m may recursively invoke itself. The compiler computes* `directlyInvokedMethods`(m) *by traversing the call graph.*

Definition 5 *Given a class c,* `nestedClasses`(c) *is the set of classes that may be directly or indirectly nested inside c.* `nestedClasses`(c) *does not include c. The compiler computes this set by traversing the class nesting graph.*

Definition 6 *m is a nested c-method if* `receiverClass`$(m) \in$ `nestedClasses`(c)*.*

Definition 7 *Given a method m,* `closed`(m) *is true if the entire computation generated as a result of executing m only accesses the receiver object of m or nested objects of the receiver object of m. The compiler computes this function by traversing the call graph.*

Definition 8 *Given two sets of methods ms_1 and ms_2, ms_1* `dominates` *ms_2 if for every method $m \in ms_2$, every path from the root of the call graph to m contains a method in ms_1.*

Definition 9 *Given a method m,* `generatesConcurrency`(m) *is true if m contains any constructs that generate parallel execution.*

5.5 The Data Lock Coarsening Algorithm

The primary responsibility of the data lock coarsening algorithm is to ensure that every time the computation executes an operation on a nested object, it holds the lock in the nested object's enclosing object. The algorithm checks that the computation satisfies this constraint by computing the set of methods that may have a nested object as the receiver. It then verifies that all of these methods are invoked only from within methods that have the nested object's enclosing object as their receiver. In this case, the algorithm can generate code that holds the enclosing object's lock for the duration of all methods that execute on nested objects. Because the lock in the enclosing object ensures the atomic execution of all methods that execute with nested objects as the receiver, the compiler can eliminate the lock constructs in these methods.

```
global lockClass;
boolean DataLockCoarseningClass(class c){
    ms1 = {m : m ∈ methods(c) and closed(m)};
    ms2 = ∪{invokedMethods(m) : m ∈ ms1} − ms1;
    ms3 = {m : m ∈ ms1 and invokedMethods(m) ⊆ ms2}
    cs  = {receiverClass(m) : m ∈ ms2};
    ms4 = ∪{methods(c′) : c′ ∈ cs};
    if(ms3 dominates ms4) then
      for all methods m ∈ ms3 ∪ ms4 do
        if(generatesConcurrency(m) or (not CoarsenGranularity(m))) return false;
      for all methods m ∈ ms3 do
        Make the generated parallel version of m invoke the synchronization-free
        version of each method that it invokes. The synchronization-free version
        contains no lock constructs and invokes the synchronization-free version
        of all methods that it invokes.
        Also make the first statement of m acquire its receiver's lock
        and the last statement of m release the lock.
      for all methods m ∈ ms4 do
        lockClass[m] = c;
      return true;
    else return false;
}

void DataLockCoarsening(){
    for all methods m do lockClass[m] = receiverClass(m);
    for all classes c in the topological sort order of the Class Nesting Graph do
      if(c is not marked)
        if(DataLockCoarseningClass(c))
          for all c′ ∈ nestedClasses(c) do mark c′
}
```

Figure 7: Data Lock Coarsening Algorithm

Figure 7 presents the data lock coarsening algorithm. Figure 8 shows the values that the algorithm computes when it runs on the node class in the example program.

Given a class c, the goal of the algorithm is to use the lock in objects of class c for all objects nested inside objects of class c. The algorithm first computes ms_1, which is the set of all closed methods with receivers of class c. Because every method m in ms_1 is closed, it is possible for the compiler to find the enclosing object of the receiver of every operation that m invokes on nested objects — the enclosing object is simply the receiver of m. ms_2 is the set of invoked methods that are nested c-methods. If the algorithm succeeds, the compiler will generate code that eliminates all of the lock constructs in these methods. ms_3 is the set of c-methods that only invoke nested c-methods. If the algorithm succeeds, the methods in ms_3 and ms_2 together contain all call sites that invoke the nested c-methods in ms_2. ms_4 is simply the set of nested c-methods that may have the same receiver as one of the methods in ms_2. For the algorithm to succeed, all of the methods in ms_4 must be invoked by the execution of a method in ms_3 — in other words, ms_3 must dominate ms_4. Before the algorithm applies the lock coarsening transformation, it checks that none of the methods in ms_3 or ms_4 contain constructs that generate parallel execution. It also checks that all of the methods in ms_3 or ms_4 meet the conditions of the false exclusion policy in Section 6. This policy is designed to ensure that the lock coarsening transformations do not introduce false exclusion.

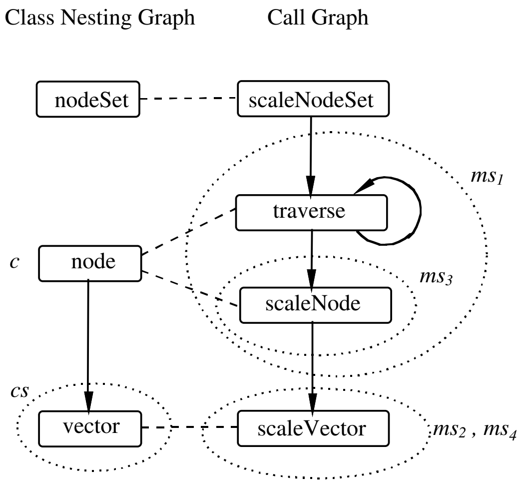


Figure 8: Computed Method and Class Sets for the Example Program

The only remaining potential problem is the possibility of false contention. We expect that locking nested objects at the granularity of the enclosing object will usually not increase the data lock granularity enough to generate a substantial amount of false contention. The data lock coarsening algorithm therefore always coarsens the granularity if it can legally do so.

The algorithm records the results of the data lock coarsening in the variable `lockClass`. Given a method m, `lockClass`$[m]$ is the class whose lock ensures the atomic execution of m. The computation lock coarsening algorithm presented below in Section 6 uses `lockClass` to determine if can legally apply the computation lock coarsening transformation.

To generate code for the transformed computation, the compiler generates a new *synchronization-free* version of all methods whose receivers are nested objects — in other words, all of the methods in the set ms_4. The synchronization-free version is the same as the original version except that it omits any synchronization constructs present in the original version and invokes the synchronization-free version of all of the methods that it executes. The compiler also modifes all of the call sites of the methods in the set ms_3 to ensure that they invoke the synchronization-free version of each invoked method.

If the data lock coarsening algorithm succeeds, it may be the case that none of the parallel phases in the transformed program acquires or releases the lock of any object of the same class as one of the nested objects. In this case, the generated code eliminates the lock from the declaration of the class and the computation never allocates the lock.

Finally, we briefly note that the data lock coarsening algorithm can never introduce deadlock. The model of computation in Section 4 ensures that the processor holds no locks when it enters the transformed version of one of the methods in ms_3. Because the entire computation of the transformed method only acquires and releases the lock in its receiver object, there is no possibility of deadlock.

6 Computation Lock Coarsening

Figure 9 presents the computation lock coarsening algorithm. This algorithm traverses the call graph, attempting to identify methods whose computation repeatedly acquires and releases the same lock. The `ComputationLockCoarseningMethod`(m) determines if it should coarsen the granularity at the execution of m. It first checks that m is closed. It then checks that none of the methods that m's computation may execute acquire and release different locks. It also checks to make sure that none of these methods contain any concurrency generation constructs. If m passes all of these tests, it is legal for the compiler to apply the computation lock coarsening transformation.

```
global lockClass;
boolean ComputationLockCoarseningMethod(method m){
  if(generatesConcurrency(m))
    return false;
  if(m is closed)
    c = receiverClass(m);
    for all m' ∈ invokedMethods(m) do
      if (lockClass[receiverClass(m)] ≠ c) or (generatesConcurrency(m)) return false;
    if(CoarsenGranularity(m))
        Make the generated parallel version of m invoke the synchronization-free
        version of each method that it invokes. This synchronization-free version
        contains no lock constructs and invokes the synchronization-free version
        of all methods that it invokes.
        Also make the first statement of m acquire its receiver's lock
        and the last statement of m release the lock.
        return true;
  return false;
}

void ComputationLockCoarsening(method m){
  if(m is not marked)
    mark m;
    if(not ComputationLockCoarseningMethod(m))
      for all m' ∈ directlyInvokedMethods(m) do
          ComputationLockCoarsening(m');
}
```

Figure 9: Computation Lock Coarsening Algorithm

The remaining question is whether coarsening the granularity will generate an excessive amount of false exclusion. The compiler currently uses one of three policies to determine if it should apply the transformation:

- **Original:** Never apply the transformation — use the original lock granularity.

- **Bounded:** Increase the granularity only if the transformation will not cause the computation to hold a lock for a statically unbounded number of method executions. The compiler implements this policy by testing for cycles in the call graph of the set of methods that may execute while the computation holds the lock. It also checks to make sure that none of these methods contain loops that invoke methods. The idea is to limit the potential severity of any false exclusion by limiting the amount of time the computation holds any given lock.

- **Aggressive:** Always increase the granularity if it is legal to do so.

This policy choice is encapsulated inside the `CoarsenGranularity`(m) algorithm. If the algorithm determines that it should apply the transformation, the compiler generates code for m that acquires the lock, invokes the synchronization-free versions of all of the methods that it invokes, then releases the lock.

The computation lock coarsening algorithm can never introduce deadlock. It simply replaces computations that acquire and release the same lock with computations that acquire and release the lock only once. If the original version does not deadlock, the transformed version can not deadlock.

7 Experimental Results

We have implemented the lock coarsening algorithms described in Sections 5 and 6 and integrated them into a prototype compiler that uses commutativity analysis [12] as its primary analysis paradigm. In this section we present experimental results that characterize the performance impact of using the lock coarsening algorithms in a parallelizing compiler. We report performance results for two automatically parallelized scientific applications: the Barnes-Hut hierarchical N-body solver [1] and the Water [13] code.

7.1 The Compilation System

Although it is possible to apply the lock coarsening algorithms to any parallel program that conforms to the model of computation in Section 4, they were developed in the context of a parallelizing compiler for object-based languages [12]. This compiler uses a new analysis framework called commutativity analysis to automatically generate parallel code.

Commutativity analysis exploits the structure present in the object-based programming paradigm to view the computation as composed of a sequence of operations on objects. It then analyzes the program to determine if operations commute (two operations commute if they generate the same result regardless of the order in which they execute). If all of the operations in a given computation commute, the compiler can automatically generate parallel code. Even though the code may violate the data dependences of the original serial program, it is still guaranteed to generate the same result.

If the compiler determines that it can parallelize a computation, it generates code for the computation as follows. It first identifies the objects that the computation modifies, then augments the objects' class declarations so that each object has a mutual exclusion lock. It then inserts acquire and release constructs into operations that access the objects. These constructs make operations execute atomically in the context of the parallel computation. At each operation invocation site the compiler generates code that executes the operation in parallel. A straightforward application of lazy task creation techniques [11] can increase the granularity of the resulting parallel computation. The compiler also applies a heuristic designed to eliminate the exploitation of excess concurrency: the generated code executes each iteration of a parallel loop serially.

Commutativity analysis complements the standard analysis framework, data dependence analysis, used in traditional parallelizing compilers. Compilers that use data dependence analysis attempt to find independent computations (two computations are independent if neither writes a variable that the other accesses). The compiler then generates code that executes the independent pieces concurrently. If a traditional parallelizing compiler fails to determine that computations are independent, it must conservatively generate code that executes the computations sequentially. Commutativity analysis extends this approach. If all operations in a given computation commute, the compiler can automatically parallelize the computation regardless of any data dependences that may exist between operations.

The compiler itself is structured as a source-to-source translator that takes an unannotated serial program written in a subset of C++ and generates an explicitly parallel C++ program that performs the same computation. We use Sage++ [3] as a front end. The analysis phase consists of approximately 14,000 lines of C++ code, with approximately 1,800 devoted to interfacing with Sage++.

The code generation phase of the compiler automatically generates lock constructs to ensure that operations on objects execute atomically. The analysis phase of the compiler yields programs that conform to the model of computation presented in Section 4. The compiler uses the lock coarsening algorithms described in Section 5 to reduce the lock overhead. A run-time library provides the basic concurrency management and synchronization functionality. The library consists of approximately 5,000 lines of C code.

The current version of the compiler imposes several restrictions on the dialect of C++ that it accepts. The goal of these restrictions is to simplify the implementation of the prototype while providing enough expressive power to allow the programmer to develop clean object-based programs. The major restrictions include:

- The program has no virtual methods and does not use operator or method overloading. The compiler imposes this restriction to simplify the construction of the call graph.
- 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 does not use pointers to members or 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.

7.2 Methodology

To evaluate the impact of the lock coarsening policy on the overall performance, we implemented the three lock coarsening policies described in Section 6. We then built three versions of the prototype compiler. The versions are identical except that each uses a different lock coarsening policy. We then used the three versions of the compiler to automatically parallelize the benchmark applications. We obtained three automatically parallelized versions of each application — one from each version of the compiler. The generated code for each application differs only in the lock coarsening policy used to reduce the lock overhead. [2] We evaluated the performance of each version by running it on a 32-processor Stanford DASH machine [8] running a modified version of the IRIX 5.2 operating system. Because the prototype compiler is a source-to-source translator, we use a standard C++ compiler to generate object code for the automatically generated parallel programs. We compiled the parallel programs using the IRIX 5.3 CC compiler at the -O2 optimization level.

7.3 Barnes-Hut

The Barnes-Hut application simulates the trajectories of a set of interacting bodies under Newtonian forces [1]. It uses a sophisticated pointer-based data structure: a space subdivision tree that dramatically improves the efficiency of a key phase in the algorithm. The application consists of approximately 1500 lines of serial C++ code. The compiler is able to automatically parallelize phases of the application that together account for over 95% of the execution time.

We start our discussion of the experimental results by reporting the execution time for each version of the parallel code running on one processor. We also report the execution time for the serial version. All performance results are reported for a version of the program running with 16384 bodies. To eliminate cold start effects, the instrumented computation omits the first two iterations. In practice the computation would perform many iterations and the amortized overhead of the first two iterations would be negligible.

A comparison of the performance of the serial version with the performance of the parallel versions provides a measure of the lock overhead. The performance results in Table 1 show that the lock coarsening algorithm has a significant impact on overall performance. Without optimization, the original parallel version runs 2.0 times slower than the serial version. The Bounded optimization level reduces the lock overhead, and the Aggressive optimization level virtually eliminates it.

Version	Execution Time	Overhead
Serial	143.5	—
Original	287.7	50%
Bounded	208.6	31%
Aggressive	148.6	3%

Table 1: Locking Overhead for the Barnes-Hut for 16384 bodies on a Single Processor

Table 2 presents the number of executed lock acquire and release constructs that each version performs. It also presents the number of allocated locks. As expected, the decreases in execution times for the different versions are closely correlated with decreases in the number of executed lock constructs. The data lock coarsening algorithm also reduces the number of allocated locks by a factor of five.

Table 3 presents the execution times for the different parallel versions running on a variety of processors; Figure 10 presents the corresponding speedup curves. The speedups are calculated relative to the serial version of the code, which executes with no lock or parallelization overhead. All versions scale well, which indicates that the compiler was able to effectively parallelize the application. Although the absolute performance varies with the lock coarsening policy, the performance of the different parallel

[2] The sequential source codes and automatically generated parallel codes can be found at `http://www.cs.ucsb.edu/~pedro/CA/apps/LockCoarsening`.

Version	Number of Executed Acquire/Release Pairs	Number of Allocated Locks
Original	$15,537 \times 10^3$	$81,920$
Bounded	$7,777 \times 10^3$	$16,384$
Aggressive	82×10^3	$16,384$

Table 2: Lock Counts for the Barnes-Hut for 16384 bodies

versions scales at approximately the same rate. This indicates that the lock coarsening algorithms introduced no significant contention.

Version	Processors									
	1	2	4	8	12	16	20	24	28	32
Serial	143.5	—	—	—	—	—	—	—	—	—
Original	287.7	146.3	82.2	41.0	29.2	24.2	20.4	19.9	16.6	15.5
Bounded	208.6	108.8	56.3	31.1	21.8	17.7	15.2	13.8	12.2	11.0
Aggressive	148.6	78.3	38.9	21.8	16.0	13.2	11.7	10.4	9.6	8.7

Table 3: Execution Times for Barnes-Hut for 16384 bodies (seconds)

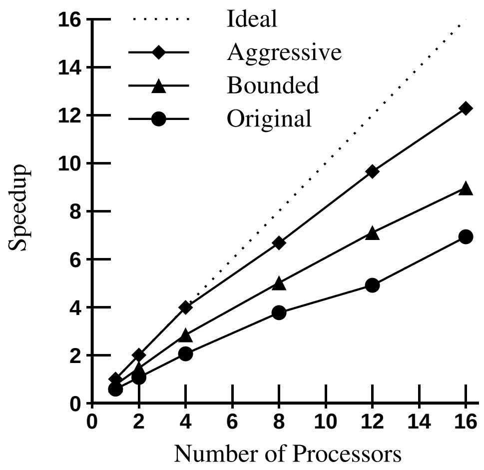


Figure 10: Speedup for Barnes-Hut (16384 bodies)

7.4 Water

Water uses a $O(n^2)$ algorithm to simulate a set of n water molecules in the liquid state. The application consists of approximately 1850 lines of serial C++ code. The compiler is able to automatically parallelize phases of the application that together account for over 98% of the execution time.

Table 4 presents the running times on one processor for the different versions of Water. With no optimization the lock operations impose an overhead of 17% over the original serial version; the

Bounded and Aggressive optimization levels drive the overhead down substantially.

Version	Execution Time	Overhead
Serial	159.5	—
Original	193.3	17%
Bounded	176.0	9%
Aggressive	168.7	5%

Table 4: Lock Overhead for the Water for 512 molecules on a Single Processor

Table 5 presents the number of executed lock acquire and release constructs that each version performs. It also presents the number of allocated locks. As for the Barnes-Hut application, the decreases in execution times for the different versions are closely correlated with decreases in the number of executed lock constructs. At the Aggressive optimization level, the lock coarsening algorithm also decreases the number of allocated locks by over a factor of 17.[3]

Version	Number of Executed Acquire/Release Pairs	Number of Allocated Locks
Original	$4,200 \times 10^3$	17,921
Bounded	$2,100 \times 10^3$	18,944
Aggressive	$1,578 \times 10^3$	1,026

Table 5: Lock Counts for Water for 512 molecules

Table 6 presents the execution times for the different parallel versions running on a variety of processors; Figure 11 presents the corresponding speedup curves. The Original and Bounded versions initially perform well (the speedup over the sequential C++ version at sixteen processors is approximately 5.2). But both versions fail to scale beyond eight processors. The Aggressive version fails to scale well at all — the maximum speedup for this version is only 2.0.

	Processors									
Version	1	2	4	8	12	16	20	24	28	32
Serial	159.5	—	—	—	—	—	—	—	—	—
Original	193.3	105.5	54.3	36.3	33.4	30.1	31.7	33.1	34.4	39.4
Bounded	176.0	93.5	48.3	33.4	30.6	31.5	31.4	31.3	34.7	39.5
Aggressive	168.6	121.9	97.7	84.7	81.0	81.8	84.3	84.9	83.7	85.9

Table 6: Execution Times for Water for 512 molecules (seconds)

[3]Surprisingly, the number of allocated locks actually increases slightly at the Bounded optimization level. Further investigation reveals that this increase is caused by the following phenomenon. In one parallel phase the lock coarsening algorithm succeeds and generates code that uses a lock in an enclosing object instead of several locks in nested objects. Because the enclosing object originally had no lock (none of the parallel regions wrote any of its instance variables), the lock coarsening algorithm introduces a new lock at the level of the enclosing object. In another parallel phase, the lock coarsening algorithm for this object does not coarsen the lock granularity. The compiler therefore does not eliminate the locks in the nested objects. The overall effect is an increase in the number of allocated locks.

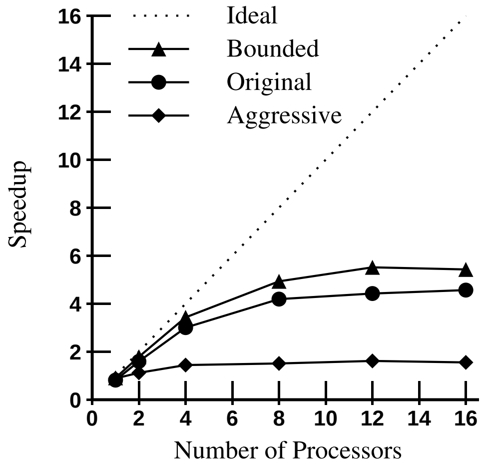


Figure 11: Speedup for Water (512 molecules)

We investigated the source of the lack of scalability by instrumenting the generated parallel code to measure how much time each processor spends in different parts of the parallel computation. The instrumentation breaks the execution time down into the following categories:

- **Parallel Idle:** The amount of time the processor spends idle while the computation is in a parallel section. Increases in the load imbalance show up as increases in this component.
- **Serial Idle:** The amount of time the processor spends idle when the computation is in a serial section. Currently every processor except the main processor is idle during the serial sections. This component therefore tends to increase linearly with the number of processors, since the time the main processor spends in serial sections tends not to increase dramatically with the number of processors.
- **Blocked:** The amount of time the processor spends waiting to acquire a lock that an operation executing on another processor has already acquired. Increases in contention for objects are reflected in increases in this component of the time breakdown.
- **Parallel Compute:** The amount of time the processor spends performing useful computation during a parallel section of the computation. This component also includes the lock overhead associated with an operation's first attempt to acquire a lock, but does not include the time spent waiting for another processor to release the lock if the lock is not available. Increases in the lock overhead show up as increases in this component.
- **Serial Compute:** The amount of time the processor spends performing useful computation in a serial section of the program. With the current parallelization strategy, the main processor is the only processor that executes any useful work in a serial part of the computation.

Given the execution time breakdown for each processor, we compute the cumulative time breakdown by taking the sum over all processors of the execution time breakdown at that processor. Figure 12 presents the cumulative time breakdowns for Water as a function of the number of processors executing the computation. The height of each bar in the graph represents the total processing time required to execute the parallel program; the different gray scale shades in each bar represent the different time breakdown categories. If a program scales perfectly with the number of processors then the height of the bar will remain constant as the number of processors increases.

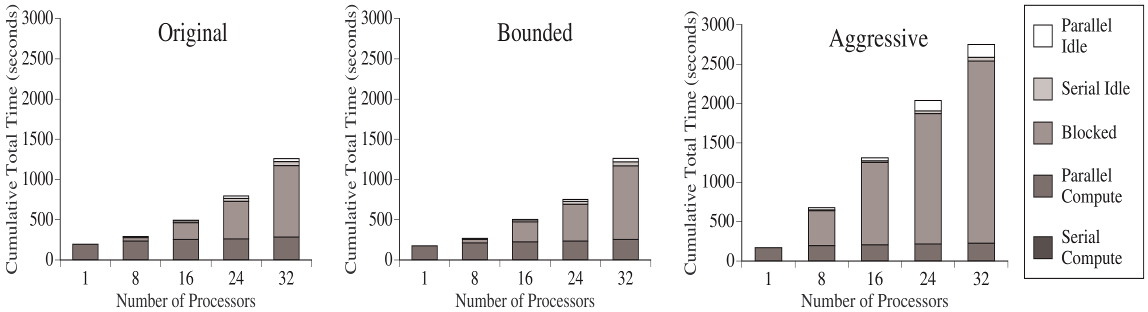


Figure 12: Cumulative Time Breakdowns for Water for 512 molecules

The cumulative time breakdowns clearly show why the Original and Bounded versions fail to scale beyond eight processors and why the Aggressive version fails to scale at all. The fact that the blocked component of the time breakdowns grows dramatically while all other components either grow relatively slowly or remain constant indicates that lock contention is the primary source of the lack of scalability.

For this application, the Bounded policy yields the best results and the corresponding parallel code attains respectable speedups. Although the Aggressive policy dramatically reduces the number of executed locking operations, the introduced lock contention almost completely serializes the execution.

8 Generating Self-Tuning Code

The most important remaining open question is how to choose the correct lock granularity for a given computation. The experimental results presented in Section 7 show that the best lock coarsening policy differs from program to program. In fact, we even expect the optimal policy to be different for different phases of the same program. As the experimental results show, the performance differences between different policies can be quite substantial. To generate acceptable performance, it must therefore be possible to adjust the lock coarsening policy to the needs of the program.

We believe it will prove to be difficult, if not impossible, for the compiler to choose the correct policy statically. In general, the correct policy choice may depend on such dynamically determined information as the topology of the data structures and the timing of the execution of parallel operations. We believe the difficulty or even impossibility of statically extracting this kind of information will prevent the compiler from obtaining the information it needs to make the correct policy decision.

Our current hypothesis is that generating self-tuning code will effectively solve the policy choice problem. A compiler that uses this approach will generate several different versions of the code for each parallelized computation. Each version will use a different lock coarsening policy. The compiler will instrument the code to collect information about the execution. We currently envision the collection of such data as the execution time of the parallel phase, the number of executed acquire and release constructs and the amount of time spent waiting to acquire locks held by other processors.

The generated parallel program will use the collected information to determine if it should dynamically switch policies. The use of a policy that is too fine will generate an excessive number of executed lock acquire and release operations. The use of a policy that is too coarse will generate an excessive amount of time spent waiting to acquire locks held by other processors. Every time the computation executes a given phase, it will compare the two measurements to determine if it should switch policies, and if so, whether to use a finer or coarser policy. We expect computations that use this approach to quickly converge on the correct lock granularity policy.

One potential concern is the instrumentation overhead. Our experimental results indicate that this overhead should be close to negligible. All of the performance results presented in Section 7 were generated from programs that collected all of the information listed above. We obtained an initial estimate of the instrumentation by running versions of the programs with the instrumentation turned off; there was no substantial difference in the execution times of the programs with and without instrumentation. These results provide encouraging evidence that the instrumentation overhead should not preclude the use of self-tuning techniques.

9 Related Work

In this section we survey related work in the area of synchronization optimizations for parallel computing and database concurrency control.

9.1 Automatically Parallelized Scientific Computations

Previous parallelizing compiler research in the area of synchronization optimization has focused almost exclusively on synchronization optimizations for parallel loops in scientific computations [10]. The natural implementation of a parallel loop requires two synchronization constructs: an initiation construct to start all processors executing loop iterations, and a barrier construct at the end of the loop. The majority of synchronization optimization research has concentrated on removing barriers or converting barrier synchronization constructs to more efficient synchronization constructs such as counters [14]. Several researchers have also explored optimizations geared towards exploiting more fine grained concurrency available within loops [4]. These optimizations automatically insert one-way synchronization constructs such as post and wait to implement loop-carried data dependences.

The research presented in this paper investigates synchronization optimizations for a compiler designed to parallelize object-based programs, not loop nests that manipulate dense arrays using affine access functions. The problem that our compiler faces is the efficient implementation of atomic operations, not the efficient implementation of data dependence constraints.

9.2 Database Concurrency Control

A goal of research in the area of database concurrency control is to develop efficient locking algorithms for atomic transactions. This goal is similar to our goal of efficiently implementing atomic operations in parallel programs. In fact, database researchers have identified lock granularity as a key issue in the implementation of atomic transactions, and found that excessive lock overhead can be a significant problem if the lock granularity is too fine [2, 6].

The proposed solution to the problem of excessive lock overhead in the context of database concurrency control is to dynamically coarsen the lock granularity using a technique called lock escalation. The idea is that the lock manager (which is responsible for granting locks to transactions) may coarsen the lock granularity by dynamically locking a large section of the database on behalf of a given transaction. If the transaction requests a lock on any object in that section, the lock manager simply checks that the transaction holds the coarser granularity lock.

There are several key differences between the lock manager algorithm and the lock coarsening algorithms presented in this paper. The lock manager algorithm only attempts to increase the data lock granularity — there is no attempt to increase the computation lock granularity. This paper presents algorithms that coarsen both the data and the computation lock granularities.

Several other differences stem from the fact that the lock manager algorithm takes place dynamically, which means that it can not change the program generating the lock requests. The programs therefore continue to execute lock acquire and release constructs at the fine granularity of individual items in the database. The goal of the lock manager algorithm is to make it possible to implement the fine grain lock requests more efficiently in cases when the lock manager has granted the transaction a coarse grain lock on a section of the database, not to change the granularity of the lock requests themselves.

Because the transactions always generate lock requests at the granularity of items in the database, the lock manager must deal with the possibility that a transaction may attempt to lock an individual item even though it does not hold a lock on the section of the database that includes the item. The locking algorithm must therefore keep track of correspondence between different locks that control access to the same objects. Tracking this correspondence complicates the locking algorithm, which makes each individual lock acquire and release less efficient in cases when the lock manager has not already granted the transaction a coarse grain lock.

The lock coarsening algorithms presented in this paper, on the other hand, transform the program so that it always generates lock requests at the coarser granularity. The fine grain lock acquire and release operations are completely eliminated and generate no overhead whatsoever. Furthermore, computations that access the same object always execute acquire and release constructs on the same lock. This property makes it possible for the implementation to use an extremely efficient lock implementation. Modern processors have synchronization instructions that make it possible to implement the required lock acquire and release constructs with an overhead of only several machine instructions [7].

9.3 Efficient Synchronization Algorithms

Other researchers have addressed the issue of synchronization overhead reduction. This work has concentrated on the development of more efficient implementations of synchronization primitives using various protocols and waiting mechanisms [5, 9].

The research presented in this paper is orthogonal to and synergistic with this work. Lock coarsening reduces the lock overhead by reducing the frequency with which the generated parallel code acquires and releases locks, not by providing a more efficient implementation of the locking constructs.

10 Conclusions

This paper addresses a fundamental issue in the implementation of atomic operations: the granularity at which the computation locks the data that atomic operations access. We have found that using the natural lock granularity for object-based programs (giving each object its own lock and having each operation lock the object that it accesses) may significantly degrade the performance. We have presented algorithms that can effectively reduce the lock overhead by automatically increasingly the granularity at which the computation locks data. We have implemented these algorithms and integrated them into a parallelizing compiler for object-based languages. We present experimental results that characterize the performance impact of using the lock coarsening algorithms in this context. These results show that the algorithms can effectively reduce the lock overhead to negligible levels.

References

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

[2] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and Beata Winnicka. Sage++: An object-oriented toolkit and class library for building Fortran and C++ structuring tools. In *Proceedings of the Object-Oriented Numerics Conference*, 1984.

[4] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, St. Charles, IL, August 1986.

[5] J. Goodman, M. Vernon, and P. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, April 1989.

[6] U. Herrmann, P. Dadam, K. Kuspert, E. Roman, and G Schlageter. A lock technique for disjoint and non-disjoint complex objects. In *Proceedings of the International Conference on Extending Database Technology (EDBT'90)*, pages 219–235, Venice, Italy, March 1990.

[7] G. Kane and J. Heinrich. *MIPS Risc Architecture*. Prentice-Hall, 1992.

[8] D. Lenoski. *The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor*. PhD thesis, Stanford, CA, February 1992.

[9] B-H. Lim and A. Agarwal. Reactive synchronization algorithms for multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1994.

[10] S. P. Midkiff and D. A. Padua. Compiler generated synchronization for DO loops. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 544–551, St. Charles, IL, August 1986.

[11] 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.

[12] 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. (An extended version of this document can be found at the authors' web address: http://www.cs.ucsb.edu/~{martin,pedro}).

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

[14] C. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 144–155, Santa Barbara, CA, July 1995.