Synchronization Transformations for Parallel Computing

PEDRO C. DINIZ
Information Sciences Institute
University of Southern California
and
MARTIN C. RINARD
Laboratory for Computer Science
Massachusetts Institute of Technology

This paper describes a framework for synchronization optimizations and a set of transformations for programs that implement critical sections using mutual exclusion locks. The basic synchronization transformations take constructs that acquire and release locks and move these constructs both within and between procedures. They also eliminate acquire and release constructs that use the same lock and are adjacent in the program.

The paper also presents a synchronization optimization algorithm, lock elimination, that uses these transformations to reduce the synchronization overhead. This algorithm locates computations that repeatedly acquire and release the same lock, then transforms the computations so that they acquire and release the lock only once. The goal of this algorithm is to reduce the lock overhead by reducing the number of times that computations acquire and release locks. But because the algorithm also increases the sizes of the critical sections, it may decrease the amount of available concurrency. The algorithm addresses this trade-off by providing several different optimization policies. The policies differ in the amount by which they increase the sizes of the critical sections.

Experimental results from a parallelizing compiler for object-based programs illustrate the practical utility of the lock elimination algorithm. For three benchmark applications, the algorithm can dramatically reduce the number of times the applications acquire and release locks, which significantly reduces the amount of time processors spend acquiring and releasing locks. The resulting overall performance improvements for these benchmarks range from no observable improvement to up to 30% performance improvement.

Categories and Subject Descriptors: D.3.4 [**Compilers**]: Parallelizing Compilers; D.4.1 [**Synchronization and Mutual Exclusion**]: Synchronization Transformations

General Terms: Parallelizing Compilers, Compilers, Parallel Computing, Synchronization Transformations

1. INTRODUCTION

The characteristics of future computational environments ensure that parallel computing will play an increasingly important role in many areas of computer science. As small-scale shared-memory multiprocessors become a commodity source of computation, customers will demand the efficient parallel software required to fully exploit the parallel hardware. The growth of the World-Wide Web will provide a new distributed computing environment with unprecedented computational power and functionality. Parallel computing will continue to play a crucial role in delivering maximum performance for scientific and engineering computations. The increasing use of multiple threads as an effective program construction technique (used, for example, in user interface systems and multi-threaded servers [Hauser et al. 1993; Cardelli and Pike 1985; Reppy 1992]) demonstrates that parallelism is not just for performance — it can also increase the expressive power of a language.

This research was supported in part by an Alfred P. Sloan Research Fellowship.

Efficient synchronization is one of the fundamental requirements of effective parallel computing. The tasks in fine-grain parallel computations, for example, need fast synchronization for efficient control of their frequent interactions. Efficient synchronization also promotes the development of reliable parallel software because it allows programmers to structure programs as a set of synchronized operations on fine-grain objects. This development methodology helps programmers overcome the challenging problems (nondeterministic behavior, deadlock, etc.) that complicate the development of parallel software.

Given the central role that efficient synchronization plays in parallel computing, we expect that future compilers will apply a wide range of synchronization optimizations. This paper takes a first step towards that goal by presenting a transformation framework and set of specific transformations for programs that contain synchronization operations. It also describes a novel synchronization optimization algorithm called *lock elimination*. This optimization is designed for programs that use mutual exclusion locks to implement critical sections. Lock elimination drives down the locking overhead by coalescing multiple critical sections that acquire and release the same lock multiple times into a single critical section that acquires and releases the lock only once. This algorithm provides a concrete example of how the transformations enable meaningful optimizations.

Finally, this paper presents experimental results that demonstrate the practical utility of lock elimination. These experimental results come from a compiler that automatically parallelizes object-based programs written in a subset of serial C++. This compiler uses a new analysis technique called commutativity analysis [Rinard and Diniz 1996]. As part of the parallelization process, the compiler automatically inserts synchronization constructs into the generated parallel code to make operations execute atomically. The significant performance improvements that synchronization optimizations deliver in this context illustrates their importance in achieving good parallel performance.

This paper makes the following contributions:

—It presents a new set of basic synchronization transformations. These synchronization transformations provide the solid foundation that a compiler requires to effectively apply synchronization optimizations.

—It presents a novel optimization algorithm, lock elimination, that a parallelizing compiler can use to reduce the synchronization overhead.

—It presents experimental results that characterize the performance impact of applying the lock elimination optimization in a parallelizing compiler for object-based programs. These results show that the optimization has a substantial impact on the performance of three benchmark programs.

2. THE MODEL OF COMPUTATION

The framework is designed for programs that execute a sequence of parallel and serial phases. Each parallel phase executes a set of parallel threads that access shared data objects. Parallel phases start by creating several parallel threads, and end when all of the created parallel threads have completed. Examples of typical concurrency generation constructs include the structured **parbegin**, **parend**, and **parfor** constructs [Dijkstra 1968], or parallel function calls [Halstead 1985; Blumofe et al. 1995].

The threads in the parallel phases use critial sections to atomically access one or more pieces of shared data. Programs implement critical sections by acquiring a mutual exclusion lock at the beginning of the critical section (using the **acquire** construct), then releasing the lock at the end of the section (using the **release** construct). In practice we expect programmers to mentally associate mutual exclusion locks with data; each critical section would then acquire and release the lock associated with the manipulated data.

This paper presents experimental results from a parallelizing compiler for object-based programs [Rinard and Diniz 1996]. The generated parallel code conforms to this model. Each parallel phase executes a set of threads. These threads are created either by a **parfor** loop or by parallel function calls. As the threads execute, they periodically update shared objects. Each shared object has its own mutual exclusion lock; each update to a shared object makes its execution atomic by acquiring and releasing the lock in the updated object. Our benchmark applications create many shared objects; the generated code therefore uses many different mutual exclusion locks to synchronize its execution.

3. PROGRAM REPRESENTATION

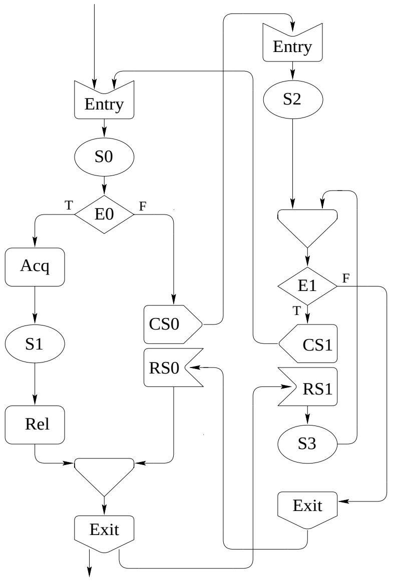
We represent the computation of each thread using an interprocedural control flow graph (ICFG) [Reps et al. 1995]. The ICFG consists of the control flow graphs of the procedures that the thread executes. The control flow graphs are augmented with edges that link procedure call sites with the entry and exit nodes of the invoked procedures. Each procedure call site is represented by two nodes: a call node and a return node. There is an edge from each call node to the entry node of the invoked procedure, and an edge back from the exit node of the invoked procedure to the corresponding return node at the call site. Each node in the ICFG has four attributes:

- **Type:** The type of the computation the node performs. Standard types include:
  - acquire (acquire a mutual exclusion lock),
  - release (release a mutual exclusion lock),
  - assignment (set a variable to a new value),
  - call (invoke a procedure),
  - return (return from a procedure),
  - entry (dummy node at the beginning of a procedure),
  - exit (dummy node at the end of a procedure),
  - if (flow of control), and
  - merge (flow of control).

  There is also a summary type (described below in Section 4.1) that represents the computation of several nodes. All release, acquire, call, return, assignment and summary nodes have a single incoming edge and a single outgoing edge. All entry and merge nodes have a single outgoing edge; all exit and if nodes have a single incoming edge.
- **Expressions:** One or more expressions representing the computation associated with the node. For example, the expression for an acquire or release node specifies the lock to acquire or release, and a call node has one expression for each parameter.
- **Read Set:** A conservative approximation of the set of variables that the node's computation reads. In general, the compiler may have to use an interprocedural pointer or alias analysis to compute a reasonably precise read set [Rugina and Rinard 1999; Emami et al. 1994; Wilson and Lam 1995; Landi et al. 1993]. In restricted contexts, the compiler may be able to use simpler algorithms. Our prototype compiler, for example, is designed for object-based programs. Because these programs use references to objects instead of pointers, it is possible to extract a reasonable read set directly from the expressions in the node [Rinard and Diniz 1996].

—**Write Set:** A conservative approximation of the set of variables that the node's computation writes.

Figure 1 contains an example ICFG. The different shapes correspond to nodes of different types. To simplify the figure, we have omitted the expressions, read sets and write sets of the nodes.



Shape	Attributes			
	Type	Expressions	Read Set	Write Set
Acq Rel	Acquire Release	Lock	Yes	No
Entry Exit	Entry Exit	None	No	No
S0	Assignment	Left Hand Side Right Hand Side	Yes	Yes
	Summary	None	Yes	Yes
T E0 F	Conditional	Condition	Yes	No
	Merge	None	No	No
CS0	Call	Arguments	Yes	No
RS0	Return	None	No	No

Fig. 1. ICFG Example

4. TRANSFORMATIONS

We next present the basic program transformations. The lock elimination algorithm described in Section 5 uses these basic transformations as the foundation for a synchronization optimization algorithm.

4.1 Abstraction Transformations

Since the synchronization transformations deal primarily with the movement and manipulation of synchronization nodes, it is appropriate for the compiler to use an abstract, simplified representation of the actual computation in the ICFG. The compiler can therefore apply several transformations that replace concrete representations of computation with more abstract representations. The end result is a simpler and smaller ICFG, which improves the performance and functionality of the synchronization optimization algorithms. The transformations are as follows:

—**Node Abstraction:** A connected set of assignment, conditional nodes or summary nodes with a single incoming edge and a single outgoing edge is replaced by a single summary node. Figure 2 presents this transformation.

—**Procedure Abstraction:** The invocation of a procedure that consists only of assignment, conditional nodes or summary nodes is replaced with a single node summarizing the execution of the procedure. Figure 3 presents this transformation.[1]

In both cases the read set and write set of the new summary node are, respectively, the union of the read sets and the union of the write sets of the set of summarized nodes. The compiler can apply these transformations both at the beginning of the optimization phase before any other transformations, and during intermediate steps of the optimization phase as they become enabled.

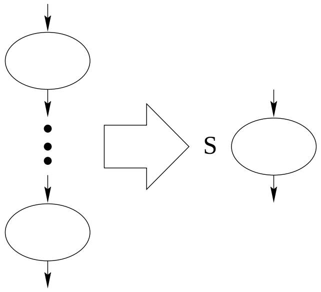


Fig. 2. Node Abstraction Transformation

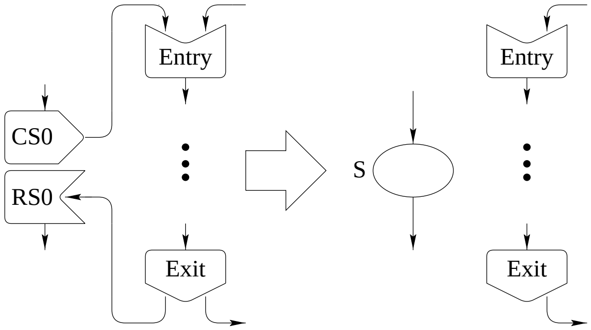


Fig. 3. Procedure Abstraction Transformation

4.2 Lock Cancellation

If a computation releases a lock, then immediately reacquires the same lock, it is possible to reduce the lock overhead by eliminating the adjacent release and acquire constructs. A similar situation occurs when the computation acquires, then immediately releases a lock. The conditional lock cancellation transformations in Figures 4 and 5 start with two adjacent release and acquire nodes and introduce a new if node that tests if the nodes acquire and release the same lock. If so, the transformed ICFG simply skips the acquire and release constructs.

[1] It is straightforward to extend this transformation to abstract sets of mutually recursive procedures.

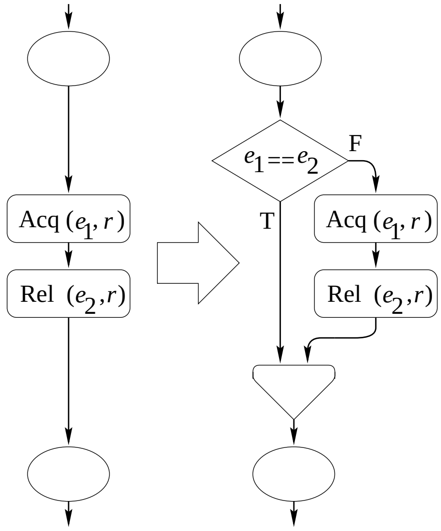


Fig. 4. Conditional Lock Cancellation Transformation

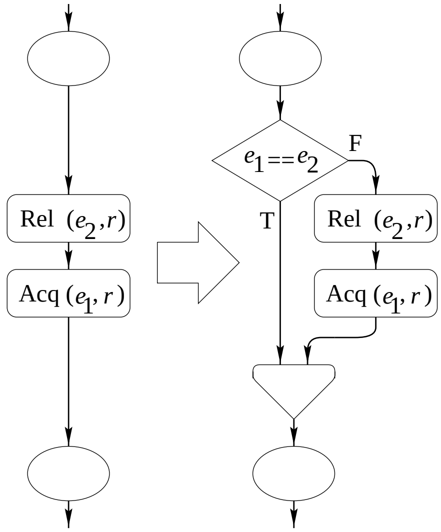


Fig. 5. Conditional Lock Cancellation Transformation

The compiler may be able to detect statically that the acquire and release manipulate the same lock. This is clearly the case, for example, if the expressions in the acquire and release are the same. In this case the compiler can simply eliminate the two nodes as illustrated in Figure 6. In effect, these transformations combine one of the conditional lock cancellation transformations with dead code elimination.

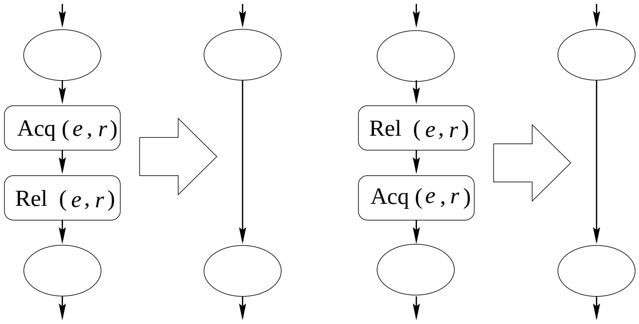


Fig. 6. Lock Cancellation Transformations

It is not usually the case that the original ICFG contains adjacent acquire and release nodes that manipulate the same lock. Our experimental results indicate, however, that when combined with the lock movement transformations described below in Section 4.3, the lock cancellation transformations can significantly reduce the number of executed acquires and releases.

4.3 Lock Movement

The lock movement transformations move an acquire or release node across an adjacent node. There are two dual transformations — one for acquire nodes and one for release nodes. Figure 7 presents the transformation that moves an acquire node A against the flow of control across an adjacent node N. The transformation introduces new acquire nodes before N, removes the original acquire node A, and introduces new release nodes on all of the edges out of N except the one that led to A. In effect, the transformed code moves N into the critical section that started with A in the original code. The release nodes ensure that the original successor of A is the only successor of N that is in the newly enlarged critical section.

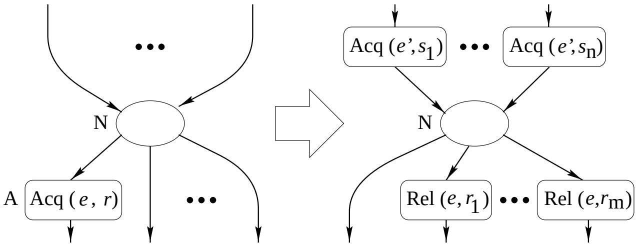


Fig. 7. Acquire Lock Movement Transformation

One possible concern is that the transformations may introduce deadlock by changing the order in which the program acquires multiple locks. To eliminate this possibility, the algorithm checks that, N can never be part of a critical section before the transformation. In this case, there is no execution path to N that holds a lock, and bringing N into another critical section will never introduce deadlock. We formalize this requirement by introducing the concept of an acquire-exposed node.

DEFINITION 1. *A node is* acquire-exposed *if there may exist a path in the ICFG to that node from an acquire node and the path does not go through a release node that releases the lock that the acquire node acquires.*

It is illegal to move an acquire node A across an acquire-exposed node N.

The original acquire node A has an expression e and read set r. The new acquire nodes have expression e' and read sets $s_1, \ldots, s_n$; the new release nodes have expression e and read sets $r_1, \ldots, r_m$. The expression e' may differ from e because e' is evaluated before N executes rather than after N executes. The expression manipulations required to transform e to e' may involve performing variable substitutions to undo the parameter bindings when moving a node out of a procedure and replacing variables with corresponding expressions when moving across an assignment node. If there are multiple edges out of N, the compiler must also ensure that the expressions in the new nodes always denote a valid lock.

The read sets in the new acquire nodes may differ from the original read set because they summarize how a new expression, e', reads variables. Even if the new expressions are identical, the new read sets may differ from the original read set and from each other because the expressions may be evaluated in different contexts.

In some cases, the compiler may be unable to apply the transformation because it cannot generate the new expression or read sets. Consider, for example, moving an acquire node A across a return node N. In this case, the acquire node is moving from a caller to a callee and no longer has access to the naming environment of the caller. If the expression in the acquire node contains local variables or parameters of the caller, it may be impossible to build a new expression in the naming environment of the callee that evaluates to the same lock as the original expression in the naming environment of the caller. Appendix A provides a complete specification of the cases that the compiler must handle.

Figure 8 presents the lock release transformation that moves a release node R with the flow of control across an adjacent node N. This transformation is the dual of the lock acquire transformation. In effect, the lock release transformation moves N into the critical section that originally ended with R. As for the acquire lock movement transformation, we require that N not be acquire-exposed. If there are multiple edges into N, the compiler must verify that all of the expressions in the new nodes always denote a valid lock.

As for the acquire lock movement transformation, the compiler may be unable to apply the release transformation because it can not generate the new expression or read sets. Consider, for example, moving a release node R with expression $a[i]$ across an assignment node that sets i to 0. In this case, the assignment node destroys information required to compute the released lock, and it may be impossible to build a new expression after the assignment node that always evaluates to the same lock as the original expression before the assignment node. As was the case with the lock acquire transformation, the compiler may also in some cases be unable to move a release node from a caller into a callee because of the different naming environments in the caller and callee. This case may occur when moving a release node R across a call node N. Appendix A provides a complete specification of the cases that the compiler must handle.

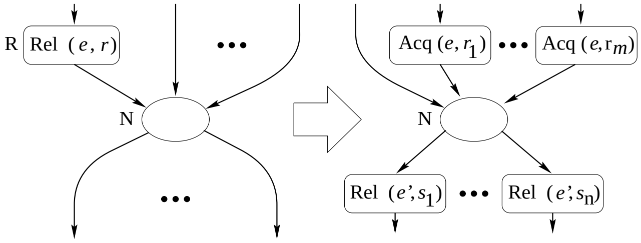


Fig. 8. Release Lock Movement Transformation

In principle, both transformations are reversible. When used in the other direction (moving acquires with the flow of control and releases against the flow of control), they have the effect of reducing the size of the critical section. It is therefore possible to use the transformations to minimize the sizes of the critical sections, which may increase the amount of available parallelism in the program. There is one important requirement, however. To ensure that the computation in the original critical sections still executes atomically, the transformations must not move a node out of a critical section if the node reads a variable that other parallel threads may write or writes a variable that other parallel threads may access.

5. LOCK ELIMINATION ALGORITHM

The goal of the lock elimination algorithm is to reduce the number of times the computation releases and acquires locks. The basic idea is to identify a computation that contains multiple critical sections that acquire and release the same lock, then transform the computation so that it contains one large critical section that acquires and releases the lock only once. Since the transformed computation acquires and releases the lock fewer times, it generates less lock overhead.

Given a region over which to eliminate synchronization constructs, the algorithm uses the lock movement transformation to increase the sizes of critical regions that acquire and release the same lock until they are adjacent in the ICFG. It then uses the lock cancellation transformation to eliminate adjacent release and acquire nodes. The end result is a single larger critical section that encompasses all of the computation between the start of the first critical section and the end of the last critical section.

5.1 False Exclusion

An overly aggressive lock elimination algorithm may introduce *false exclusion*. False exclusion may occur when a processor holds a lock during an extended period of computation that was originally part of no critical section. If another processor attempts to execute a critical section that uses the same lock, it must wait for the first processor to release the lock even though the first processor is not executing a computation that needs to be in a critical section. False exclusion may therefore reduce the performance by decreasing the amount of available concurrency.

The amount of false exclusion in a given parallel execution depends on information such as the dynamic interleaving of the parallel tasks and the relative execution times of pieces of the ICFG. This information is, in general, unavailable at compile time, and may even be different for different executions of the same program. The lock elimination algorithm addresses the issue of false exclusion using a heuristic false exclusion policy. This policy is based exclusively on information from the static call graph of the program. The basic idea is to limit the potential severity of false exclusion by limiting the subgraphs of the ICFG to which the algorithm applies the lock elimination algorithm. The policy considers each procedure in turn to determine if it should apply the algorithm to the subgraph of the ICFG containing that procedure and all procedures that it (transitively) invokes. The lock elimination algorithm therefore only applies lock elimination to a subgraph if the subgraph satisfies the false exclusion policy. The current compiler supports four different policies:

—**Original:** Never apply the transformation — always use the default placement of acquire and release constructs. In the default placement for our current compiler, each operation that updates an object acquires and releases that object's lock.

—**Bounded:** Apply the transformation only if the new critical region will contain no cycles in the call graph. The idea is to limit the severity of any false exclusion by limiting the dynamic size of the critical region.

—**Aggressive:** Always apply the transformation unless the transformation would serialize the entire computation of the parallel phase. In the current implementation, the compiler checks if the application of the transformations completely serializes the computation of a parallel phase. If the compiler is unable to verify this condition, it applies the transformation.

—**Greedy:** Always apply the transformation whenever possible.

5.2 The Lock Elimination Algorithm

The basic idea behind the lock elimination algorithm is to find an acquire node and a release node, find a path in the ICFG along which they can move until they are adjacent, then use lock cancellation transformations to eliminate them. A by-product of the sequence of transformations is a set of new acquire and release nodes introduced on edges that lead into and out of the path. The algorithm performs the following steps:

—**Apply False Exclusion Algorithm:** The algorithm performs a depth-first traversal of the call graph. At each node of the call graph, the algorithm considers the subgraph reachable from that node in the call graph. If this subgraph satisfies the false exclusion policy, the algorithm invokes the lock elimination algorithm on the procedure corresponding to that node.

—**Reachability Tree:** The lock elimination algorithm chooses a release node and an acquire node, then computes the *reachability tree* for each node. The reachability tree contains the set of edges to which the algorithm can move the acquire or release node using the lock movement transformations. It also contains the new expressions and read sets computed in the lock movement transformations. Figure 9 contains an example reachability tree for an acquire node. This figure omits the expressions and read sets in the reachability trees; the edges in the reachability tree are shaded. The reachability tree does not extend past the return node RS1 because of the naming environment issues discussed in Section 4.3 associated with moving an acquire node out of a caller into a callee.

Figure 10 contains an example reachability tree for a release node. The reachability tree does not extend past the call node CS1 because of the naming environment issues discussed in Section 4.3 associated with moving a release node out of a caller into a callee.

—**Reachability Tree Intersection:** Given two reachability trees, the algorithm next checks if they intersect and have the same expression for the lock at an edge where they intersect. If so, it is possible to move the acquire and release nodes to be adjacent in the ICFG. Note that it may be possible to move a release node and an acquire node to be adjacent even though neither node's reachability tree reaches the other node. This may occur, for example, if it is necessary to move the nodes out of invoked procedures into a common caller procedure. Figure 11 contains an example of this situation. It identifies the intersection edge using a thick line. In general, there is no requirement that the intersection of the reachability trees be a single edge. In some cases, the intersection may be multiple connected edges, or even multiple disjoint sets of connected edges.

—**Movement Paths:** If the trees intersect, the algorithm chooses one of the edges in the intersection and follows the edges in the reachability trees to obtain paths from the release and acquire nodes to this intersection edge. The acquire and release can move along these paths to become adjacent. Figure 12 shows the movement paths in our example.

—**Transformation:** To apply the transformation, the algorithm eliminates the original acquire and release nodes, then introduces new acquire and release nodes into edges that lead into and out of the two movement paths. In effect, the algorithm applies all of the lock movement and cancellation transformations in one step to move all of the nodes in the path into the enlarged critical section. Figure 13 presents the transformed ICFG in our example. It identifies the new acquire and release nodes using thick boxes.

—**Repetition:** The algorithm repeatedly eliminates acquire and release nodes until there are no such nodes whose reachability trees intersect. Figure 14 shows the final transformed ICFG.

Appendix B contains a precise specification of the lock elimination algorithm. We next discuss a few of its properties.

5.3 No Introduced Deadlock

The lock elimination algorithm has the same effect as performing multiple lock movement and lock cancellation transformations. Recall that because these transformations do not move acquire or release nodes past acquire-exposed nodes, they do not change the order in which the program acquires multiple locks.[2] The transformations therefore do not introduce deadlock. If the original program never deadlocks, then the transformed program never deadlocks.

5.4 Termination

We next address the termination property of the lock elimination algorithm. Because the lock movement transformations introduce new acquire and release nodes, it may not be completely obvious that the algorithm always terminates. The key observation is that each lock elimination transformation inserts at least one non-synchronization node into a critical section and takes no nodes out of critical sections, i.e., the critical sections are always expanded by at least one node. The algorithm terminates when all of the nodes in the

[2] The transformation may, however, create a new critical section at a place in the program where previously there was none.

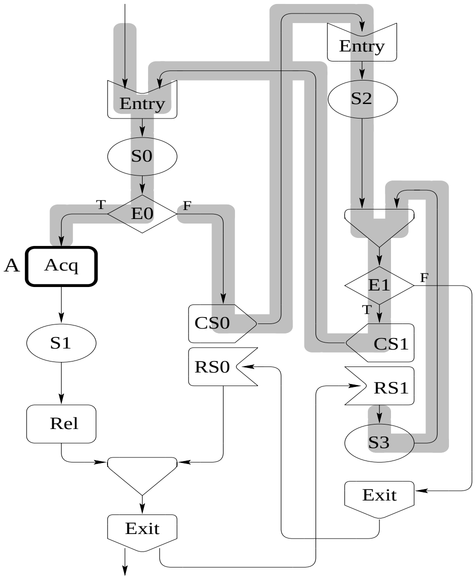


Fig. 9. Reachability Tree for Acquire Node A

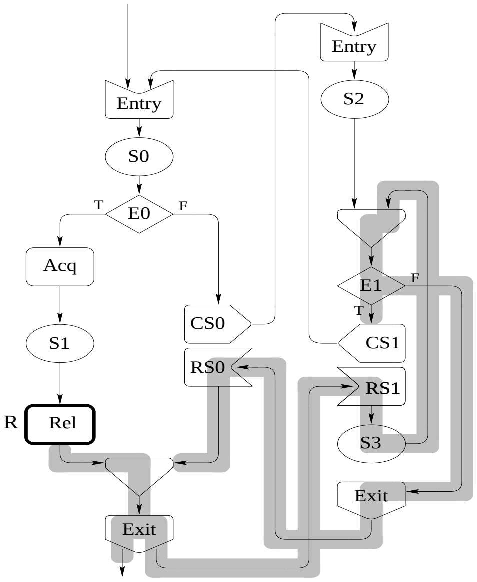


Fig. 10. Reachability Tree for Release Node R

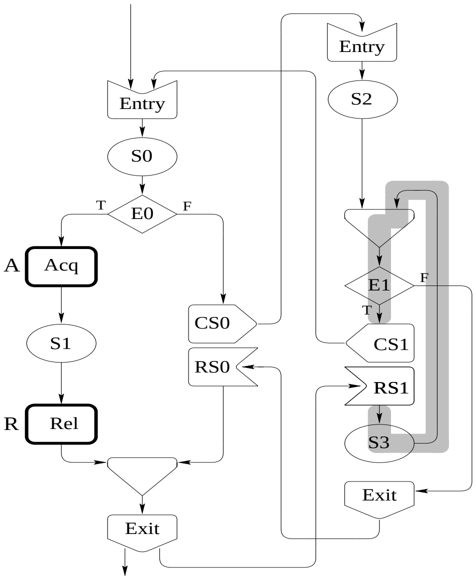


Fig. 11. Intersection of Reachability Trees for Acquire Node A and Release Node R

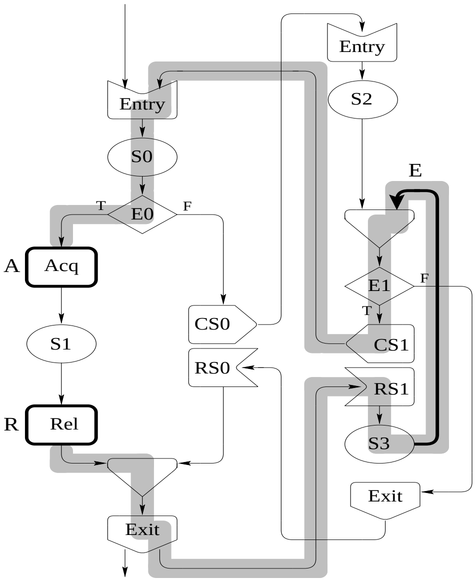


Fig. 12. Movement Paths and Intersection Edge E for Acquire Node A and Release Node R

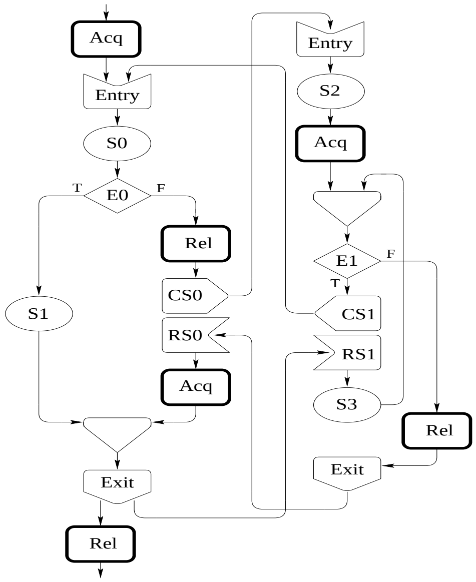


Fig. 13. Result of a Single Lock Elimination Step

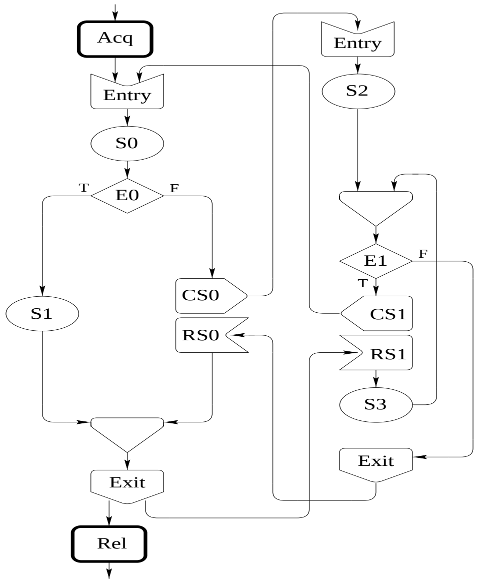


Fig. 14. Final Result of Lock Elimination Algorithm

ICFG are in critical sections. It therefore performs at most as many transformations as there are nodes in the ICFG.

5.5 Time Complexity

We now address the algorithmic time complexity of the lock elimination algorithm. To simplify the discussion and avoid pathological cases, we normalize the ICFG by applying two minor ICFG simplifications. These two transformations reduce the number of acquire and release pairs the algorithm needs to check.

The first simplification is to eliminate any adjacent acquire and release nodes with the exact same lock expression. If the acquire node precedes the release node, the simplification eliminates an empty critical section. If the release node precedes the acquire node, the simplification merges the two critical sections. To apply this simplification, the compiler can scan all of the edges of the ICFG to find adjacent acquire and release nodes with the same expressio. The second simplification is to test an acquire or release node for a possible lock elimination only if it is connected to a non-synchronization node.

In this normalized form of the ICFG, the upper bound on the running time of the lock elimination algorithm is $O(ne^2 \max(e, nC))$, where e is the number of edges in the ICFG, n is the number of nodes and C is the complexity of computing the new expression and read sets for a single lock movement.

The first observation is that each lock elimination transformation inserts at least one non-synchronization node into a critical section and takes no nodes out of critical sections, i.e., the critical sections are always expanded. The algorithm therefore performs at most $n + 1$ iterations. We next consider the amount of work done per iteration. The algorithm considers at most 2 synchronization nodes per edge of the original ICFG in the pairs of tested acquire and release nodes. The total number of tested pairs is therefore at most $O(e^2)$. For each pair the algorithm performs at most $O(\max(e, nC))$ work. To construct the reachability trees, the algorithm must visit at most $O(e)$ edges, computing the new expression and read sets at most $O(n)$ times. It is possible to fold the reachability tree intersection into the reachability tree construction. The movement path and transformations can also be computed and performed in $O(e)$ time. Each iteration therefore takes $O(\max(e, nC))$ time, and the total running time is $O(ne^2 \max(e, nC))$. We expect that, in practice, the number of synchronization nodes will be small relative to the number of edges and nodes in the ICFG, and the running time will be substantially faster than this upper bound might suggest.

Because of the symbolic variable substitution, it is possible for the computations of the new expressions and read sets to generate expressions that are exponentially larger than the original expressions in the graph. We expect that, in practice, few programs will contain lock expressions that elicit this behavior.

6. EXPERIMENTAL RESULTS

We now describe the context and methodology used to quantitatively evaluate the performance of the lock elimination algorithm described in this article.

6.1 Parallelizing Compiler

We have implemented a lock elimination algorithm in the context of a parallelizing compiler for serial object-based programs. The compiler uses commutativity analysis [Rinard and Diniz 1996] to extract the concurrency in the program. It views the computation as consisting of a sequence of operations on objects, 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 code that executes the operations in parallel. Objects that may be updated by multiple parallel threads are called *shared objects*. In the generated code, each shared object has a mutual exclusion lock. Each operation in a parallel phase that updates a shared object uses that object's lock to ensure that it executes atomically. Because our benchmark applications create many shared objects, the generated code uses many locks to synchronize its execution. Because the generated code holds at most one lock at any given time, it does not deadlock. Even though the parallel execution may change the order in which the operations are performed relative to the serial computation (which may violate the data dependences), the fact that all operations commute guarantees that all parallel executions generate the same final result as the serial execution.

The compiler exploits the structure present in the object-based programming paradigm to use a significantly simplified lock elimination algorithm. In this paradigm, each operation accesses only the local variables, the parameters and the object that it operates on. The compiler also exploits this control to simplify the data structures used in the implementation of the lock elimination algorithm — the implemented algorithm operates on the call graph and control flow graph for each procedure rather than on an explicit enlarged ICFG.

6.2 Methodology

We report performance results for three automatically parallelized scientific applications: the Barnes-Hut hierarchical N-body solver [Barnes and Hut 1986], the Water code [Singh et al. 1992] and the String code [Harris et al. 1990]. Barnes-Hut simulates the trajectories of a set of interacting bodies under Newtonian forces; it consists of approximately 1500

lines of C++ code. Water simulates the interaction of water molecules in the liquid state; it consists of approximately 1850 lines of C++ code. String constructs a two-dimensional discrete velocity model of the geological medium between two oil wells; it consists of approximately 2050 lines of C++ code. The performance of the serial C++ versions of Barnes-Hut and Water is slightly better than the performance of highly optimized parallel C versions from the SPLASH-2 benchmark set [Singh et al. 1992] running on one processor. The performance of the serial C++ version of String is approximately 1% slower than the original C version.

The compiler currently supports all four false exclusion policies described in Section 5.1. We generated four instrumented versions of each application; each version uses a different false exclusion policy. We evaluated the performance of each version by running it on a 16-processor Stanford DASH machine [Lenoski 1992].

6.3 Barnes-Hut

We evaluate the overhead of each false exclusion policy by running the three automatically parallelized versions on one processor and comparing the execution times with the execution time of the sequential program. The performance results in Table I show that the lock elimination algorithm has a significant impact on the overall performance. Without lock elimination, the original parallel version runs significantly slower than the serial version. Lock elimination with the Bounded policy reduces the lock overhead, and the Aggressive and Greedy policies virtually eliminate it. For this application, the Greedy and Aggressive policies produce the same generated parallel code. The table presents the number of executed acquire and release pairs for each of the different versions; these numbers correlate closely with the execution times. The table also presents the number of static acquire and release constructs for each version; this is the number of these constructs in the code.

Version	Execution Time	Execution Time Overhead	Acquire/Release Pairs	
			Executed	Static
Serial	147.8	—	—	—
Original	217.2	46.9%	15,471,682	3
Bounded	191.7	29.7%	7,744,033	3
Aggressive Greedy	149.9	1.4%	49,152	2

Table I. Locking Overhead for Barnes-Hut (16384 bodies) on a Single Processor

Table II presents the execution times for the different parallel versions running on a variety of processors; Figure 15 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.[3] All versions scale well, which indicates that the compiler was able to effectively parallelize the application. Although the absolute performance varies with the false exclusion policy, the performance of the different parallel versions scales at approximately the same rate. This indicates that the lock elimination algorithm introduced no significant false exclusion.

[3] The speedup is the running time of the serial version divided by the running time of the parallel version. The speedup curves plot the speedup as a function of the number of processors executing the parallel version.

Version	Processors					
	1	2	4	8	12	16
Serial	147.8	—	—	—	—	—
Original	217.2	111.6	56.59	32.61	20.76	15.64
Bounded	191.7	97.25	49.22	26.98	19.62	15.12
Aggressive Greedy	149.9	76.30	37.81	21.88	15.57	12.87

Table II. Execution Times for Barnes-Hut (16384 bodies) (seconds)

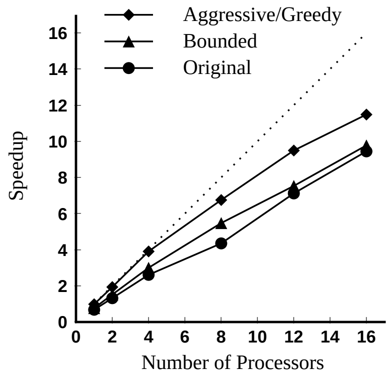


Fig. 15. Speedups for Barnes-Hut (16384 bodies)

6.4 Water

Table III presents the execution statistics for the single processor runs of Water. For this application, the Aggressive and Bounded policies produce the same generated parallel code. With no lock elimination, the synchronization overhead is 16% over the original serial version. Lock elimination with the Bounded, Aggressive and Greedy policies drives the overhead down substantially. As expected, the number of executed acquire and release constructs is correlated with the execution times.

Version	Execution Time	Execution Time Overhead	Acquire/Release Pairs	
			Executed	Static
Serial	159.5	—	—	—
Original	184.4	16%	4,200,448	3
Bounded Aggressive	175.8	10%	2,099,200	5
Greedy	165.3	4%	1,577,980	5

Table III. Locking Overhead for Water (512 molecules) on a Single Processor

Table IV presents the execution times for the different parallel versions running on a variety of processors; Figure 16 presents the corresponding speedup curves. The Original, Bounded and Aggressive versions initially perform well (the speedup over the sequential

C++ version at eight processors is close to six). But both versions fail to scale beyond twelve processors. The Greedy version fails to scale well at all.

Version	Processors					
	1	2	4	8	12	16
Serial	159.5	—	—	—	—	—
Original	184.4	94.60	47.51	28.39	22.06	19.87
Bounded Aggressive	175.8	88.36	44.28	26.42	21.06	19.50
Greedy	165.3	115.2	88.45	79.18	75.16	73.54

Table IV. Execution Times for Water (512 molecules) (seconds)

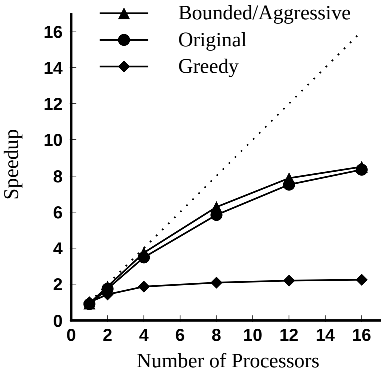


Fig. 16. Speedups for Water (512 molecules)

We instrumented the parallel code to determine the source of the performance loss. Figure 17 presents the *contention proportion*, which is the proportion of the time that processors spend waiting to acquire a lock held by another processor.[4] This figure clearly shows that lock contention is the primary cause of performance loss for this application, and that the Greedy false exclusion policy generates enough false exclusion to severely degrade the performance.

6.5 String

Table V presents the execution statistics for the single processor runs of String. All experimental results are for the Big Well input data set. For this application, the Aggressive, Bounded, and Original policies produce the same generated parallel code. With no lock elimination, the synchronization overhead is 18% over the original serial version. Lock elimination with the Greedy policy reduces the overhead to 6%. As expected, the number of executed acquire and release constructs is correlated with the execution times.

[4] More precisely, the contention proportion is the sum over all processors of the amount of time that each processor spends waiting to acquire a lock held by another processor divided by the execution time times the number of processors executing the computation.

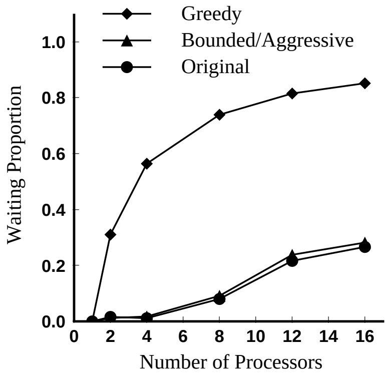


Fig. 17. Contention Proportion for Water (512 molecules)

Version	Execution Time	Execution Time Overhead	Acquire/Release Pairs Executed	Static
Serial	2208.9	—	—	—
Original Bounded Aggressive	2599.0	18%	30, 286, 506	1
Greedy	2337.7	6%	2, 313	2

Table V. Locking Overhead for String (*big* well) on a Single Processor

Version	Processors					
	1	2	4	8	12	16
Serial	2208.9	—	—	—	—	—
Original Bounded Aggressive	2599.0	1289.4	646.7	331.9	223.9	172.3
Greedy	2337.7	2313.5	2231.9	2244.3	2254.8	2260.9

Table VI. Execution Times for String (seconds)

Table VI presents the execution times for the different parallel versions running on a variety of processors; Figure 18 presents the corresponding speedup curves. The Original, Bounded and Aggressive versions perform very well, attaining a speedup of more than 12 on 16 processors. The Greedy version fails to scale at all: the Greedy false exclusion policy serializes the entire computation.

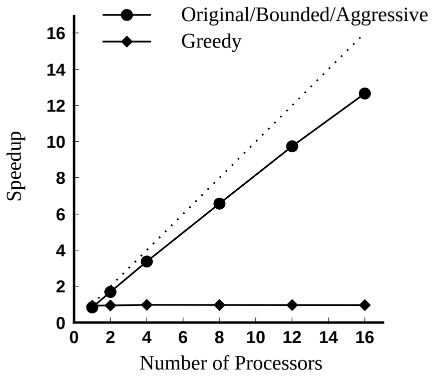


Fig. 18. Speedups for String (*big* well)

Figure 19 presents the contention proportion for this application. This figure shows that lock contention is the primary cause of performance loss for this application, and that it generates enough false exclusion to severely degrade the performance.

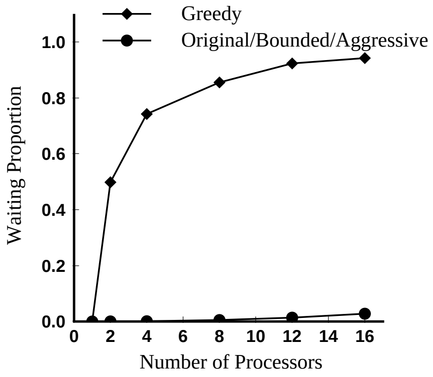


Fig. 19. Contention Proportion for String (*big* well)

6.6 Discussion

The lock elimination algorithm is designed for programs with significant lock overhead. In general, the lock overhead is determined by two factors: the relative cost of the constructs that acquire and release locks, and the frequency with which the computation executes these constructs. The measured overhead of acquiring and releasing a lock on the Stanford DASH machine is approximately 5 to 6 microseconds. Our experimental results indicate that our benchmark applications originally execute lock constructs relatively frequently and generate a significant amount of lock overhead. The lock elimination algorithm, however, is very effective at reducing the number of times that the applications acquire and release locks. Our experimental results also show that the lock elimination algorithm is capable of introducing enough false exclusion to significantly degrade the performance. For Water and String, the Greedy version performed significantly worse than all other versions, with the performance degradation directly attributable to false exclusion.

In addition, our experimental results show that there are significant performance differences between the different false exclusion policies. For all of the benchmark applications, however, the Aggressive false exclusion policy yields the best performance. In general, we expect different applications to have distinct best false exclusion policies. Moreover, different parallel phases of the same application might exhibit distinct best false exclusion policies. A production system could therefore choose a default policy, but allow the programmer to override the default to obtain better performance.

Another alternative is to dynamically sample the performance of the different policies, then use the policy with the best performance. The generated code could resample at regular intervals to adapt to dynamic changes in the best policy. We have implemented a system that implements this approach, and found that it works well in practice [Diniz and Rinard 1997]. This system is capable of automatically generating code that, without programmer assistance, automatically chooses the best policy for the current application running in the currrent computational environment.

7. RELATED WORK

The closest related work is our own previous research on techniques to reduce lock overhead in automatically parallelized object-based programs [Diniz and Rinard 1996]. This research used a monolithic algorithm that depends heavily on the restrictions of the object-based programming paradigm and the fact that the compiler controls the placement of the acquire and release constructs. The algorithm is formulated as a set of conditions on the call graph. If the call graph meets the conditions, the compiler can omit the automatic insertion of synchronization constructs into some of the procedures. There is no clearly identified set of transformations, the algorithms are only capable of increasing the sizes of the critical sections, and they work only at the granularity of entire procedures.

This article, on the other hand, describes a general set of transformations for programs that use mutual exclusion locks to implement critical sections. This flexible set of transformations enables the movement and cancellation of acquire and release constructs both within and across procedures. Because of the extra structure present in the object-oriented paradigm, however, the presented algorithm and our previous algorithm generate identical code for our set of benchmark applications.

The lock elimination algorithm in this article is formulated as a reachability problem in the ICFG rather than as a set of conditions on a call graph. While the two algorithms yield identical results in the context of our prototype compiler, formulating the problem as a reachability problem means that the new algorithm inherits all of the advantages of the basic transformations. In particular, the new algorithm is more flexible and applies to explicitly parallel programs that already contain synchronization constructs. It is possible to apply the optimization both within and across procedures, rather than only at the granularity of procedures. The formulation also removes the dependence on the compiler's ability to control the placement of the synchronization constructs.

Plevyak, Zhang and Chien have developed a similar synchronization optimization technique, *access region expansion*, for concurrent object-oriented programs [Plevyak et al. 1995]. Because access region expansion is designed to reduce the overhead in sequential executions of such programs, it does not address the trade off between lock overhead and waiting overhead. The goal is simply to minimize the lock overhead.

7.1 Parallel Loop Optimizations

Other synchronization optimization research has focused almost exclusively on parallel loops in scientific computations [Midkiff and Padua 1987]. 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 [Tseng 1995]. Several researchers have also explored optimizations geared towards exploiting more fine-grained concurrency available within loops [Cytron 1986]. These optimizations automatically insert one-way synchronization constructs such as post and wait to implement loop-carried data dependences.

The transformations and algorithms presented in this article address a different problem. They are designed to optimize mutual exclusion synchronization, not barrier synchronization or post/wait synchronization. We believe, however, that it would be possible and worthwhile to combine both classes of optimizations into a single unified synchronization optimization framework.

7.2 Analysis of Explicitly Parallel Programs

The transformations presented in this article operate on explicitly parallel programs. Other researchers have investigated the issues associated with performing standard serial compiler analyses and optimizations in the presence of explicit concurrency [Chow and Harrison III 1992; Midkiff and Padua 1990]. Our research is orthogonal to this research in the sense that it focuses on optimization opportunities that appear only in explicitly parallel programs rather than on the significant challenges associated with applying standard optimizations to parallel programs.

7.3 Concurrent Constraint Programming

The lock movement transformations are reminiscent of transformations from the field of concurrent constraint programming that propagate *tell* and *ask* constructs through the program [Saraswat et al. 1991]. The goal is to make tells and corresponding asks adjacent in the program. This adjacency enables an optimization that removes the ask construct. A difference is the asymmetry of asks and tells: the optimization that eliminates the ask leaves the tell in place. The lock cancellation transformation, of course, eliminates both the acquire and the release.

7.4 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 [Goodman et al. 1989; Lim and Agarwal 1994].

The research presented in this article is orthogonal to and synergistic with this work. Lock elimination 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.

8. CONCLUSION

As parallel computing becomes part of the mainstream computing environment, compilers will need to apply synchronization optimizations to deliver efficient parallel software. This paper describes a framework for synchronization optimizations, a set of transformations for programs that implement critical sections using mutual exclusion locks, and a synchronization optimization algorithm for reducing synchronization overhead in such programs. Experimental results from a parallelizing compiler for object-based programs illustrate the practical utility of this optimization. The resulting overall performance improvement for these benchmarks range from no observable improvement to up to 30% performance improvement.

REFERENCES

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

Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., and Zhou, Y. 1995. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York, Santa Barbara, CA.

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

Chow, J. and Harrison III, W. 1992. Compile-time analysis of parallel programs that share memory. In *Proceedings of the 19th Annual ACM Symposium on the Principles of Programming Languages*. 130–141.

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

Dijkstra, E. 1968. The structure of the THE multiprogramming system. *Commun. ACM 11,* 5.

Diniz, P. and Rinard, M. 1996. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, San Jose, CA, 285–299.

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

Emami, M., Ghiya, R., and Hendren, L. J. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*. Orlando, FL.

Goodman, J., Vernon, M., and Woest, P. 1989. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*. 64–75.

Halstead, Jr., R. 1985. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems 7,* 4 (Oct.), 501–538.

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

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

Landi, W., Ryder, B., and Zhang, S. 1993. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN '93 Conference on Program Language Design and Implementation*. ACM, New York, New York, NY, 56–67.

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

Lim, B.-H. and Agarwal, A. 1994. Reactive synchronization algorithms for multiprocessors. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, San Jose, CA.

Midkiff, S. and Padua, D. 1987. Compiler algorithms for synchronization. *IEEE Transactions on Computers 36,* 12 (Dec.), 1485–1495.

Midkiff, S. and Padua, D. 1990. Issues in the optimization of parallel programs. In *Proceedings of the 1990 International Conference on Parallel Processing*. II–105–113.

Plevyak, J., Zhang, X., and Chien, A. 1995. Obtaining sequential efficiency for concurrent object-oriented languages. In *Proceedings of the 22nd Annual ACM Symposium on the Principles of Programming Languages*. ACM, New York, San Francisco, CA.

Reppy, J. 1992. Higher–order concurrency. Ph.D. thesis, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y.

Reps, T., Horowitz, S., and Sagiv, M. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd Annual ACM Symposium on the Principles of Programming Languages*. ACM, New York, New York, NY, 49–61.

RINARD, M. AND DINIZ, P. 1996. Commutativity analysis: A new framework for parallelizing compilers. In *Proceedings of the SIGPLAN '96 Conference on Program Language Design and Implementation*. ACM, New York, Philadelphia, PA, 54–67.

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

SARASWAT, V., RINARD, M., AND PANANGADEN, P. 1991. Semantic foundations of concurrent constraint programming. In *Proceedings of the 18th Annual ACM Symposium on the Principles of Programming Languages*. Orlando, FL, 333–352.

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

TSENG, C. 1995. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York, Santa Barbara, CA, 144–155.

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

A. LOCK MOVEMENT ALGORITHMS

The lock movement algorithms are given an expression e, read set r and node N with read set r_N and write set w_N. They compute the new expression e' and read sets $r_1, \ldots, r_m, s_1, \ldots, s_n$ required to move a synchronization node with expression e and read set r across N. There are several potential complications:

—The compiler may be unable to generate the new expression e'. This may happen, for example, if N writes variables in the read set r. In this case the compiler cannot apply the transformation.

—If the algorithm moves an acquire node over a node with multiple outgoing edges or a release node over a node with multiple incoming edges, the new expressions may be evaluated in different contexts than the original expression. The compiler must therefore ensure that the evaluation of the expressions in the new contexts does not generate an error and that the new nodes always acquire or release a valid lock.

In general, the compiler may have to use an interprocedural pointer or alias analysis to verify that these conditions hold [Rugina and Rinard 1999; Emami et al. 1994; Wilson and Lam 1995; Landi et al. 1993]. In restricted contexts, the compiler may be able to use simpler algorithms. Our prototype compiler, for example, is designed for object-based programs. These programs structure the computation as a sequence of *operations* on *objects*. Each object contains a lock. Within an operation on an object, the expression that denotes the object's lock always denotes a valid lock.

—Whenever the new expressions may be evaluated in a different context than the original expression, the new read sets must reflect the behavior of the expressions in the new contexts. In general, the compiler may have to use an interprocedural pointer or alias analysis to compute the new read sets. The structure of the object-based paradigm allows the compiler to use a simpler approach. Expressions in object-based programs contain only local variables, parameters, and references to instance variables of objects. For a given expression, the read set is simply the set of variables that appear in the expression. The read set therefore depends only on the expression, not the context in which it appears.

Here are the cases that the acquire lock movement algorithm must handle:

—type(N) = entry. (The acquire node moves out of a callee towards the caller and can no longer access the local variables of the callee.) e must contain no local variables (but may contain parameters and references to instance variables of objects); $e' = e$ and $s_1, \dots, s_n = r$. The compiler may be able to generate more precise read sets for $s_1, \dots, s_n$.

—type(N) = exit. (The acquire node moves into a callee from a caller.) $e' = e$. The compiler must verify that e always denotes a valid lock in all of the new contexts. It must also generate new read sets $s_1, r_1, \dots, r_m$ to reflect the variables that e may read in the new contexts. For object-based programs, $s_1, r_1, \dots, r_m = r$.

—type(N) = merge. (The acquire node moves up into several different flow of control paths.) $e' = e$ and $s_1, \dots, s_n = r$. The compiler may be able to generate more precise read sets for $s_1, \dots, s_n$.

—type(N) = call. (The acquire node moves into the caller of the procedure that it just moved out of.) $e' = e$ with the expressions in the call node that denote the values of the actual parameters substituted in for the corresponding formal parameters in e. If e contains any formal parameters, $s_1 = r \cup r_N$, otherwise $s_1 = r$. The compiler may be able to generate a more precise read set s_1.

—type(N) = return. (The acquire node moves out of a caller towards a callee). In the new context, the acquire node will no longer have access to the local naming environment of the caller.) Expression e must contain no local variables or parameters; $e' = e$ and $s_1 = r$.

—type(N) = summary. If $w_N \cap r = \emptyset$, (if the summary writes no variables that the release reads) $e' = e$ and $s_1 = r$. Otherwise the transformation can not be applied.

—type(N) = assignment. There are several cases:

—$w_N \cap r = \emptyset$. (N writes no variables that the acquire node may read.) $e' = e$ and $s_1 = r$.

—The assignment is of the form $v = exp$, where v is a local variable. Expression e must not dereference a pointer variable that may point to v; $e' = e$ with exp substituted in for v in e. If expression e contains at least one occurrence of v, $s_1 = r \cup r_N$, otherwise $s_1 = r$. The compiler may be able to generate a more precise read set s_1.

—Otherwise the transformation can not be applied.

—type(N) = if. (The acquire node moves across a branch.) The compiler must verify that expression e always denotes a valid lock in all of the new contexts. The branch may, for example, test if the lock is NULL, and execute the acquire node only if the lock is not NULL. In this case, the compiler cannot safely move the acquire node past the branch node. The compiler must also generate new read sets $s_1, r_1, \dots, r_m$ to reflect the variables that e may read in the new contexts. For object-based programs, $s_1, r_1, \dots, r_m = r$.

—type(N) = acquire or release. The transformation can not be applied.

The release lock movement algorithm must handle the following cases:

—type(N) = entry. (The release node is moving into a procedure from the caller of the procedure.) $e' = e$. The compiler must verify that expression e always denotes a valid lock in all of the new contexts. It must also generate new read sets $r_1, \dots, r_m, s_1$ to reflect the variables that e may read in the new contexts. For object-based programs, $r_1, \dots, r_m, s_1 = r$.

—type(N) = exit. (The release node is moving out of a callee towards the caller, and can no longer access the local variables of the callee.) Expression e must contain no local variables (but may contain parameters); $e' = e$ and $s_1, \ldots, s_n = r$. The compiler may be able to generate more precise read sets for $s_1, \ldots, s_n$.

—type(N) = merge. $e' = e$. The compiler must verify that expression e always denotes a valid lock in all of the new contexts. The merge brings together multiple control flow paths, and the lock expression may be valid on some but not all of the paths. If the compiler cannot verify that the lock expression is valid on all paths, it cannot apply the transformation. The compiler must also generate new read sets $r_1, \ldots, r_m, s_1$ to reflect the variables that e may read in the new contexts. For object-based programs, $r_1, \ldots, r_m, s_1 = r$.

—type(N) = call. (The release node is moving out of a caller towards a callee and no longer has access to the local variables or parameters of the caller.) e must contain no local variables or parameters; e' = e and $s_1 = r$.

—type(N) = return. (The release node is moving into a caller from a callee.) First find the call node that corresponds to N. None of the nodes in the invoked procedure or any procedures that it directly or indirectly invokes may write any of the variables in the call node's read set. $e' = e$ with the expressions in the call node that denote the values of the actual parameters substituted in for the corresponding formal parameters in e. If expression e contains any formal parameters, $s_1 = r \cup r_N$, otherwise $s_1 = r$. The compiler may be able to generate more precise read sets.

—type(N) = assignment or type(N) = summary. If $w_N \cap r = \emptyset$, (if the assignment or summary writes no variables that the release reads) $e' = e$ and $s_1 = r$. Otherwise the transformation can not be applied.

—type(N) = if. $e' = e$ and $s_1, \ldots, s_n = r$. The compiler may be able to generate more precise read sets for $s_1, \ldots, s_n$.

—type(N) = acquire or release. The transformation can not be applied.

B. LOCK ELIMINATION ALGORITHM

The lock elimination algorithm uses the following primitives.

—`invokedProcedures`(p) : the set of procedures directly or indirectly invoked by procedure p.

—`procedure`(N) : the procedure that the ICFG node N is in.

—`type`(N) : the type of the ICFG node N.

—`predecessor`(N) : the predecessor of N in the ICFG. Only valid for nodes with one predecessor.

—`successor`(N) : the successor of N in the ICFG. Only valid for nodes with one successor.

—`predecessors`(N) : the set of predecessors of N in the ICFG.

—`successors`(N) : the set of successors of N in the ICFG.

—`insertNode`$(\langle N_{from}, N_{to}\rangle, t, \langle e, r\rangle)$: insert a new node into the ICFG whose type is t, expression is e and read set is r. There is an edge from N_{from} to the new node and an edge from the new node to N_{to}. Remove the edge $\langle N_{from}, N_{to}\rangle$ from the ICFG.

—`removeNode`(N) : remove node N from the ICFG. Make all predecessors of N predecessors of the successor of N and all successors of N successors of the predecessor of N. Only valid for nodes with one successor and one predecessor.

—$\langle e', r_1, \ldots, r_m, s_1, \ldots, s_n \rangle =$ `acquireTransform`(N, e, r) :
computes the new expression e' and read sets r_1, . . . , r_m, s_1,
. . . ,s_n that result from moving an acquire node with expression e and read set r across node N. If the transformation cannot be applied, $e' = \epsilon$.

—$\langle e', r_1, \ldots, r_m, s_1, \ldots, s_n \rangle =$ `releaseTransform`(N, e, r) :
computes the new expression e' and read sets $r_1, \ldots$,$r_m, s_1, \ldots$,s_n that result from moving a release node with expression e and read set r across node N. If the transformation cannot be applied, $e' = \epsilon$.

```
// lockElimination($p$) applies the lock elimination algorithm to the procedure $p$.
lockElimination($p$){
  do {
    $ps =$ invokedProcedures($p$);
    $ns = \{N :$ procedure($N$) $\in ps\}$;
    $ns_{acq} = \{N \in ns :$ type($N$) $= acquire\}$;
    $ns_{rel} = \{N \in ns :$ type($N$) $= release\}$;
    $applied = false$;
    for all $\langle N_{acq}, N_{rel} \rangle \in \{ns_{acq} \times ns_{rel}\}$
      if(attemptTransform($N_{acq}, N_{rel}, ns$))
        $applied = true$;
        break;
  } while ($applied = true$);
}
```

Fig. 20. Lock Elimination Algorithm

```
// attemptTransform($N_{acq}, N_{rel}, ns$) attempts to propagate and cancel $N_{acq}$ and $N_{rel}$.
// To implement the false exclusion policy, the transformation must be confined to the set of nodes $ns$.
attemptTransform($N_{acq}, N_{rel}, ns$){
  // Step 1. Compute the reachability trees, expressions and read sets for the acquire and release nodes.
  $\langle parent_{acq}, edges_{acq}, ed_{acq}, visted_{acq} \rangle =$
    acquireTraverse(predecessor($N_{acq}$), $N_{acq}$, exp($N_{acq}$), read($N_{acq}$), $\emptyset, \emptyset, \emptyset, \emptyset, ns$);
  $\langle parent_{rel}, edges_{rel}, ed_{rel}, visted_{rel} \rangle =$
    releaseTraverse($N_{rel}$, successor($N_{rel}$), exp($N_{rel}$), read($N_{rel}$), $\emptyset, \emptyset, ed_{acq}, \emptyset, ns$);
  // Step 2. Check if the two reachability trees intersect and if the acquire and release
  // manipulate the same lock.
  if($\exists \langle N_{from}, N_{to} \rangle \in edges_{acq} \cap edges_{rel} : ed_{acq}(\langle N_{from}, N_{to} \rangle) = ed_{rel}(\langle N_{from}, N_{to} \rangle)$) {
    choose any $\langle N_{from}, N_{to} \rangle \in edges_{acq} \cap edges_{rel}$;
    // Step 3. Find the path in the reachability trees from the release node to the acquire node.
    $path = \texttt{computePath}(N_{to}, N_{acq}, parent_{acq}) \cup \texttt{computePath}(N_{from}, N_{rel}, parent_{rel})$;
    // Step 4. Find the edges coming into the path and the edges going out of the path.
    // The algorithm will insert new acquire nodes on all of the incoming edges and new
    // release nodes on all of the outgoing edges.
    $acq = \bigcup_{N \in path}(\texttt{predecessors}(N) - (path \cup \{N_{rel}\})) \times \{N\}$;
    $rel = \bigcup_{N \in path}\{N\} \times (\texttt{successors}(N) - (path \cup \{N_{acq}\}))$;
    // Step 5. Insert new acquire and release nodes.
    for all $\langle N_{from}, N_{to} \rangle \in acq$ do
      insertNode($\langle N_{from}, N_{to} \rangle, acquire, ed_{rel}(\langle N_{from}, N_{to} \rangle)$);
    for all $\langle N_{from}, N_{to} \rangle \in rel$ do
      insertNode($\langle N_{from}, N_{to} \rangle, release, ed_{rel}(\langle N_{from}, N_{to} \rangle)$);
    // Step 6. Remove original acquire and release nodes.
    removeNode($N_{acq}$);
    removeNode($N_{rel}$);
    return true;
  } else {
    return false;
  }
}
```

Fig. 21. attemptTransform Algorithm

```
// computePath computes the path from the node $N$ in $parent$ back to $N_{final}$.
computePath($N, N_{final}, parent$){
  if($N = N_{final}$)
    return $\emptyset$
  else
    return $\{N\} \cup$ computePath($parent(N), N_{final}, parent$);
}
```

Fig. 22. computePath Algorithm

```
// acquireTraverse computes the reachability tree for an acquire node. At each step
// it computes the result of moving the acquire node across another node in the ICFG.
// The meanings of the variables are as follows:
// $\langle N_{from}, N_{to}\rangle$: edge that the traversal reached in the last step.
//     The traversal will next try to move the acquire node back across $N_{from}$.
// $e$ and $r$: expression and read set that result from propagating the acquire node through
//     the ICFG to the edge $\langle N_{from}, N_{to}\rangle$.
// $parent$: partial function from ICFG nodes to ICFG nodes.
//     It records the path back to the acquire node.
// $edges$: set of ICFG edges reachable by propagating the acquire node through the ICFG
//     against the flow of control.
// $ed$: partial function from ICFG edges to tuples of expressions and read sets.
//     For each edge it records the expression and read set that would result from propagating
//     the acquire node back through the ICFG to that edge.
// $visited$: set of ICFG nodes already visited by the traversal.
// $ns$: set of ICFG nodes. To implement the false exclusion policy, the tree must stay within this set.
acquireTraverse($N_{from}, N_{to}, e, r, parent, edges, ed, visited, ns$){
  $edges = edges \cup \{\langle N_{from}, N_{to}\rangle\}$;
  $ed = ed[\langle N_{from}, N_{to}\rangle \mapsto \langle e, r\rangle]$;
  if($N_{from} \notin visited$ and $N_{from} \in ns$) {
    $visited = visited \cup \{N_{from}\}$;
    $parent = parent[N_{from} \mapsto N_{to}]$;
    // Compute the new expression and read set that result from moving the acquire across $N_{from}$.
    $\langle e', r_1, \ldots, r_m, s_1, \ldots, s_n\rangle$ = acquireTransform($N_{from}, e, r$);
    if($e' \neq \epsilon$) {
      // Record the expression and read set for any new release nodes.
      for all $N_i \in$ successors($N_{from}$) $- \{N_{to}\}$ do
        $ed = ed[\langle N_{from}, N_i\rangle \mapsto \langle e, r_i\rangle]$;
      for all $N_i \in$ predecessors($N_{from}$) do
        $\langle parent, edges, ed, visited\rangle$ =
          acquireTraverse($N_{from}, N_i, e', s_i, parent, edges, ed, visited, ns$);
    }
  }
  return $\langle parent, edges, ed, visited\rangle$;
}
// releaseTraverse computes the reachability tree for a
// release node. Essentially the dual of acquireTraverse.
releaseTraverse($N_{from}, N_{to}, e, r, parent, edges, ed, visited, ns$){
  $edges = edges \cup \{\langle N_{from}, N_{to}\rangle\}$;
  $ed = ed[\langle N_{from}, N_{to}\rangle \mapsto \langle e, r\rangle]$;
  if($N_{to} \notin visited$ and $N_{to} \in ns$) {
    $visited = visited \cup \{N_{to}\}$;
    $parent = parent[N_{to} \mapsto N_{from}]$;
    $\langle e', r_1, \ldots, r_m, s_1, \ldots, s_n\rangle$ = releaseTransform($N_{to}, e, r$);
    if($e' \neq \epsilon$) {
      for all $N_i \in$ predecessors($N_{to}$) $- \{N_{from}\}$ do
        $ed = ed[\langle N_i, N_{to}\rangle \mapsto \langle e, r_i\rangle]$;
      for all $N_i \in$ successors($N_{to}$) do
        $\langle tree, ed, visited\rangle$ =
          releaseTraverse($N_{to}, N_i, e', s_i, parent, edges, ed, visited, ns$);
    }
  }
  return $\langle parent, edges, ed, visited\rangle$;
}
```

Fig. 23. acquireTraverse and releaseTraverse Algorithms