Using Data Structure Knowledge for Efficient Lock Generation and Strong Atomicity

Gautam Upadhyaya, Samuel P. Midkiff and Vijay S. Pai
Purdue University
{gupadhya, smidkiff, vpai}@purdue.edu

Abstract

To achieve high-performance on multicore systems, shared-memory parallel languages must efficiently implement atomic operations. The commonly used and studied paradigms for atomicity are fine-grained locking, which is both difficult to program and error-prone; optimistic software transactions, which require substantial overhead to detect and recover from atomicity violations; and compiler-generation of locks from programmer-specified atomic sections, which leads to serialization whenever imprecise pointer analysis suggests the mere possibility of a conflicting operation. This paper presents a new strategy for compiler-generated locking that uses data structure knowledge to facilitate more precise alias and lock generation analyses and reduce unnecessary serialization. Implementing and evaluating these ideas in the Java language shows that the new strategy achieves eight-thread speedups of 0.83 to 5.9 for the five STAMP benchmarks studied, outperforming software transactions on all but one benchmark, and nearly matching programmer-specified fine-grained locks on all but one benchmark. The results also indicate that compiler knowledge of data structures improves the effectiveness of compiler analysis, boosting eight-thread performance by up to 300%. Further, the new analysis allows for software support of strong atomicity with less than 1% overhead for two benchmarks and less than 20% for three others.The strategy also nearly matches the performance of programmer-specified fine-grained locks for the SPECjbb2000 benchmark, which has traditionally not been amenable to static analyses.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Parallel programming

General Terms Algorithms, Performance, Design

Keywords Automatic lock generation, transactional memory, parallel programming

1. Introduction

The rise of multicore processor architectures has increased the need for higher productivity development of parallel applications. General purpose multicore processors support shared memory hardware, and are often targeted with a shared memory programming model such as Pthreads, Java, or OpenMP. Concurrent and parallel application development on shared memory systems requires support for mechanisms for controlling a thread's access to shared data values, and more specifically, to enforce atomic execution of a sequence of accesses to, and operations on, shared data. Three means of enforcing this atomicity are in widespread use or are areas of intense research: (1) programmer-controlled locking, (2) transactions that rely on speculative execution and roll back the execution of the atomic section when necessary [9, 11, 18, 20, 21], and (3) programmer-specified atomic sections, with locks generated by a compiler [4, 6, 8, 12, 17, 24]. While these strategies make parallel programming possible, and in some cases easier, all lead to significant problems.

A programmer-controlled locking strategy requires a whole-program locking protocol and sufficient programmer awareness of what data needs to be protected by which locks. The locking protocol typically must interact with third party code or standard library code that is not available to the programmer. Failure to properly follow the correct locking protocol can lead to deadlock and races, and techniques to detect these failures form an active area of research.

Transactions execute atomic sections speculatively to achieve concurrency dynamically. Hardware transactional memory (HTM) achieves this speculation with low overhead by exploiting the cache coherence protocol and processor speculative state to detect and recover from atomicity violations, but only very limited HTM has been implemented in commercial processors [18]. Software transactional memory (STM) adds instruction overhead to track which threads are touching which data and to recover if needed. Further, non-transactional code such as I/O and other system calls and writes to volatile registers are problematic in transactions because of the speculative nature of transactions.

Programmer-specified atomic sections that are enforced with compiler-generated locks are non-speculative; thus, they avoid the issues that HTMs and STMs encounter as a result of speculation. High-performance implementation of atomic sections does depend, however, on effective compiler analysis. In particular, conservativeness in alias or escape analyses can result in too many locks or overly coarse-grained locks, as shown in [4, 8]. This conservativeness typically arises from complex pointer-chasing code of the sort found in standard data structures like hashmaps, trees, lists, and queues. Thus, while these atomic sections work well with relatively simple code, their performance is often worse in the presence of pointer-based data structures.

As a further limitation, most implementations of atomic sections (whether enforced via lock inference or implemented using STMs) do not provide *strong atomicity* – the ability to prevent races resulting from unsynchronized code in one thread containing accesses to data guarded by synchronization in another thread. Strong atomicity is important because it lets a function use locally placed atomic sections to guarantee its correctness without any possible interference from external code.

This paper describes a technique to enforce the semantics of atomic sections using locks. It does so in the context of the Java programming language. We have developed an efficient and effective algorithm to enable the insertion of locks to enforce atomic regions. A major impediment to automatically generated locks to enforce atomic regions occurs in code with pointer-based data structures such as hashmaps and linked lists. Our system uses the concept of *smart data structures*, which are a library of concurrent data structures whose internal aliasing semantics are exported to

PPoPP'10, January 9–14, 2010, Bangalore, India.

the compiler, thus enabling less conservatism in our alias analysis and allowing locks to be generated efficiently even in the presence of complicated pointer-chasing code in libraries. Finally, this more precise alias analysis allows strong atomicity to be enforced while preserving scalability in the benchmarks we have studied.

This paper makes several key contributions.

- It illustrates how smart data structures sharpen alias analysis and other pointer-based analyses.
- It explains our analysis and lock assignment algorithms as applied to Java programs, and how they interact with smart data structures.
- It provides empirical evidence of the effectiveness of these algorithms, achieving speedups of 0.83 to 5.9 on eight threads (compared to a fully sequential version with no parallel constructs or synchronization) for the five STAMP benchmarks studied, outperforming STM on all but one of the benchmarks, and nearly matching the performance of programmer-specified fine-grained locks on all but one of the benchmarks. The results also indicate that compiler knowledge of data structures boosts eight-thread performance by up to 300%. Our techniques allow for software support of strong atomicity with less than 1% overhead for two benchmarks and less than 20% for three others. In addition, we present results from the SPECjbb2000 benchmark (which has traditionally not been amenable to static analyses) showing performance nearly equal to that of programmer-supplied fine-grained locking.

The paper is organized as follows. Section 2 makes explicit the advantages of our smart data structure approach. Section 3 describes our extension of Soot's SPARK analysis [16] to be smart data structure aware, and Section 4 describes our lock generation algorithm. Section 5 presents experimental results that show the effectiveness of our technique. Finally Section 6 gives an overview of related work, and Section 7 presents our conclusions.

2. A Case for Data Structure Aware Compilers

Multicore architectures have made the problem of how to synchronize parallel programs much more widespread. Almost all applications run in a parallel hardware environment, and could benefit from increased parallelism within the application. Concurrent and parallel program development requires a means of enforcing the *atomicity* of blocks of code - sequences of instructions which operate on shared data and which must appear to execute atomically. As discussed in the introduction, the use of compiler-generated locks to enforce programmer-specified atomic regions would overcome the problems faced by other techniques if accurate compile-time analysis of complicated pointer-based algorithms were possible.

The problems with compile-time analysis are of two kinds. First, to bound the running times of alias and escape analyses, it is necessary to "name" objects based on static program constructs. A common technique is to name all objects by their allocation site or type, thus treating all objects with the same allocation site (or type) as the same object. Second, complicated control flow makes it difficult to determine statically what object some reference is actually pointing to. This in turn makes inferring what objects are shared, and thus what objects must be locked, less precise. Note that even relatively simple data structures can lead to conservative decisions on aliasing. A simple linked list, for example, presents problems to an analyzer ignorant of the nature of the underlying data structure and the semantics of the operation being performed. Consider the following fragment of code used to insert data into a sorted linked list:

```
void insert(int data){
   while(node.data < data)
      node = node.next
   // insert new node
}
```

Given that the compiler cannot make any assumption about where within the list the new node must be inserted, it must conservatively assume that every node is being modified. The resulting lockset serializes access to the entire list, resulting in a severe performance degradation.

Fortunately, in modern languages some of the most complicated pointer-chasing code is found in standard library code. Standard libraries have allowed significant functionality to be added to programs without complicating the base language. Moreover, in languages like Java with well-defined naming conventions for libraries, a compiler can *know* what the semantics of a particular library call is, and how this call will affect aliasing relationships. Moreover, these libraries are typically thread safe, allowing a compiler to know that it need not be concerned with generating locks for the internal "meta-data" of the library. This knowledge about libraries can be exploited to allow efficient generation of locks by a compiler using alias and/or escape analyses that are no more complex than the current state-of-the-art. We refer to such concurrent data structures with exportable aliasing semantics as *smart data structures*. Section 4.4 describes how the library programmer specifies the semantics of the various smart data structures to the compiler. We note that the compiler does not check to ensure the thread-safety of the various data structures, just as the Java programmer trusts in the reentrancy of various structures in the standard library, and does not explicitly check them.

Our compiler exploits information about these data structures implemented as standard libraries to improve the quality of our alias analysis, which is an enhanced form of the SPARK analysis supplied by Soot. Because data structure knowledge enables superior alias information, we use a lock generation algorithm that is simple to implement and maintain.

As an example of the performance gains that are possible (more extensive results are provided in Section 5), we measured the performance of a simple synthetic benchmark with a data structure-aware and unaware analysis. The benchmark attempts to insert and delete values to and from a concurrent linked list. The analyzer which was aware of the semantics of the underlying concurrent data structure could generate more aggressive locksets, leading to marked performance gains (up to 700%).

3. Alias Analysis

Our alias analysis is based on the SPARK points-to set analysis [16]. SPARK is a context and flow-insensitive, subset-based, field-sensitive analysis that uses, and is distributed with, the Soot optimization framework. We have changed SPARK in two significant ways. First, SPARK "names" objects by their allocation site. We "name" objects by the class static or threadable object constructor argument that they may get their value from. In Java, only those objects that are aliased to, or reachable from, a class static or thread constructor may be accessed concurrently in multiple threads, i.e. may thread-escape. For brevity we call this class of references *escape portals* or EPs. Our analysis gives special treatment to EP references. Secondly, we extend SPARK to provide a context-sensitive (but flow-insensitive) points-to analysis.

Spark has two main phases: construction of the pointer assignment graph (PAG) by initializing the graph with the effects of individual references within the statements of a procedure (Section 3.1), and an interprocedural propagation phase that allows summarization of the interprocedural effects on the PAG (Section 3.3).

3.1 Construction of the pointer assignment graph

The PAG, as its name suggests, summarizes the effects of assignments between pointers, or references. The PAG is a tuple (N, V), where nodes represent reference variables that are the targets of assignments (i.e. n_x is the node for reference variable x), and edges $e = (t_e, n_x, n_y)$, where t_e is the edge type (*assignment*, *load* and *store*), n_x is the left hand side of the expression for each type, and n_y is the right hand side. The three types of edges (assignment, load and store) correspond to the assignments $x = y$, $b1 = a.b$ and $a.b = b1$, respectively. Their transfer functions (i.e. the rules governing how the various points-to sets are propagated) are summarized in Table 1.

allocation edge ($a \Leftarrow newA()$)	$PointsTo(a) \cup = A$
assignment edge ($x \Leftarrow y$)	$PointsTo(x) \cup = PointsTo(y)$
load edge ($b1 \Leftarrow a.b$)	$\forall a_i \in PointsTo(a)$ $PointsTo(b1) \cup = PointsTo(a_i.b)$ $read\text{-}set(PointsTo(a_i.b)) \cup = C$
store edge ($a.b \Leftarrow b1$)	$\forall a_i \in PointsTo(a)$ $PointsTo(a_i.b) \cup = PointsTo(b1)$ $write\text{-}set(PointsTo(a_i.b)) \cup = C$
smart data structure PUT ($\omega.put(a)$)	$\forall \omega_i \in PointsTo(\omega)$ $PointsTo(\omega_i.data) \cup = PointsTo(a)$ $write\text{-}set(\omega_i) \cup = C$
smart data structure GET ($a \Leftarrow \omega.get(\ldots)$)	$\forall \omega_i \in PointsTo(\omega)$ $PointsTo(a) \cup = PointsTo(\omega_i.data)$ $read\text{-}set(\omega_i) \cup = C$

Table 1. Points-to set transfer functions for the various PAG edges; *C* is the current context.

As in SPARK, the PAGs are always rooted at allocation sites since multiple chains of references to allocated objects are what indicate aliasing. However, only those allocation sites which are reachable from a common EP object actually thread-escape, and only those allocation sites are considered during lock generation.

SPARK is context insensitive: it does not differentiate, for example, between multiple calls to the same method. In addition, there is no provision within SPARK to recognize atomic sections. To improve the accuracy of the alias analysis, we convert SPARK into a context sensitive analysis. We do so in the standard way: by converting the call graph into a call tree, as in, for example, [25]. In addition, we incorporate support for atomic sections within SPARK by creating a new context for each atomic section.

3.2 Transfer functions for smart concurrent data structures

We now provide some details about how smart data structures interact with our compiler's alias analysis. In this discussion, we differentiate between operations that insert values into the data structure (`PUT` operations) and operations that return objects from the data structure (`GET` operations).

A data structure's primary purpose is to serve as a repository for data; the underlying repository may be an array, a linked list, or something more esoteric. Regardless of the structure, every data structure has a field (or a set of fields) that stores the data. We call this field $data$ and model it as an object, to be compatible with the rest of the SPARK framework.

Let D be a smart data structure and let $PointsTo(v_i)$ be the points-to set of a reference to object v_i being inserted into D via a `PUT` operation. At the end of the `PUT`, the points-to-set of the field $D.data$ updated by the `PUT` contains $PointsTo(v_i)$, in addition to its previous contents, i.e. $PointsTo(D.data) \cup = PointsTo(v_i)$. Let $v_1, v_2, \ldots, v_N$ be the set of all objects which are added to D via `PUT` operations. Then, at the end of this sequence of `PUT` operations the set of all allocation sites in the points-to set for *D.data* is $PointsTo(D.data) = \bigcup_{i=1}^{i=N} PointsTo(v_i)$.

Let there be an assignment $x = \texttt{GET}_D$, which returns a reference to some object v referenced from *D.data*. Then, because we model `GET` operations as returning any possible data referenced in D, the updated point-to set of x is found by unioning it with the points-to set of v, which is *PointsTo(D.data)* from above, i.e.

$$PointsTo(x) \cup = PointsTo(D.data)$$

Table 1 gives the complete transfer functions of the points-to set analysis.

3.3 Alias set propagation

Our enhanced algorithm, like the original SPARK algorithm, uses a worklist to propagate alias information. However, unlike the original SPARK algorithm, the worklist used in our algorithm is a per-context worklist. The worklist is initialized with all allocation edges contained within that context. The algorithm processes a node on the worklist by inspecting all of its various edges and setting the points-to, read, and write sets as summarized in Table 1. If this processing changes the points-to set of any node, that node must then be added to the worklist. The algorithm continues until the worklist empties and every possible alias relationship has thus been determined.

4. Lock DAG Generation and Lockset Generation

The lock DAG data structure is the key data structure used in our analysis for assigning locks. A lock DAG is built for every EP referenced in the program. Nodes in a lock DAG for an EP are the classes, and fields of those classes, reachable from the EP. Using the *PointsTo* sets computed during alias analysis, the nodes of the lock DAG can be annotated with read and write operations performed on the node, and the thread performing the operation. It is this use of alias analysis that leads to poor performance in the presence of overly-conservative alias information, and that enables superior application performance when smart data structure information is used to improve the alias information. The annotated lock DAG is then used to determine which class fields point to objects involved in races, and therefore which class fields require that locks be generated to prevent those races.

In Section 4.1 the construction of the lock DAG is described. Section 4.2 then shows how the lock DAG is used to generate and optimize locksets.

4.1 Lock DAG creation

A lock DAG (or DAG for short) is constructed for each EP. Thus a DAG is created for each class static variable and each object reference variable that was initialized during the construction of the object. Given an EP, a DAG is constructed by making the root of the DAG a node whose type (T) is the type of the EP. The DAGs themselves are rooted at the nodes corresponding to the various EPs. We then recursively follow the points-to sets of the fields of references reachable from the various EPs and add DAG edges to the nodes corresponding to their allocation sites. Naively adding nodes to the DAG would lead to a DAG of unbounded size when confronted with a recursive data structure. We handle this by checking if a node corresponding to an allocation site to be added is already present in the DAG. If it is, no action is taken for that node. Figure 2 shows the pseudo-code for the lock DAG construction.

We now present an example of DAG construction. Figure 1(c) shows the constructed lock DAG for a class static EP referenced instance of the class $Data$ (shown in Figure 1(a)). W_{T_i} next to a node indicates a write-access by thread Ti to that node, while R_{T_i} indicates a read-access. Thus, in Figure 1(c), node B is being write-touched by thread T_1, while node C is being read-touched by thread T_3. Figure 1(b) shows some pseudo-code illustrating how thread T_1 performs a write-access to a node; the pseudocode for the other accesses is not shown, but the DAG in Figure 1(c) is generated assuming that thread T_1 has a read-access to node E, thread T_2 has a write-access to node D, thread T_3 has a read-access to node C, thread T_4 has a write-access to node E and that thread T_5 has a write-access to node F. These thread-access annotations are generated by the alias analysis and are copied over onto the lock DAG during the construction of the DAG, as shown in Figure 2.

4.2 Generating Locksets from a lock DAG

After creating a lock DAG our algorithm generates an initial conservative set of locks for the atomic sections in the program.

The lockset generation is achieved by iterating through the lock DAG and eliminating (or *pruning*) those nodes whose corresponding operations are not involved in any data races. Once the locksets for the different threads have been generated, a further round of optimizations attempts to reduce the total number of locks required. We present the DAG-pruning and lockset optimization algorithms below.

4.2.1 Lock DAG pruning and initial lockset generation

Before proceeding, we give some needed definitions.

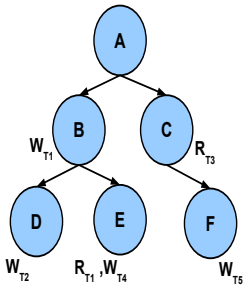
DEFINITION 1. *A node B is an* ancestor *of a node A if* any *path from the root R of the lock DAG to A passes through B. The*

```
class Data {
    TypeB B;
    TypeC C;
}

class TypeB {
    int D;
    char E;
}

class TypeC {
    float F;
}
```

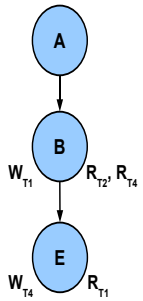
(a) Pseudo-code for accessed data structures

```
Class T extends Thread {
    static Data A;
    atomic {
        ...
        A.B = <...>
    }
}
Class Main {
    T1 = new T(); T2 = new ...; ...
    T1.run(); T2.run(); ...
}
```

(b) Pseudo-code to access shared reference



(c) Lock DAG before pruning



(d) Lock DAG after pruning

Figure 1. Lock DAG example.

ancestor relationship node B is an ancestor of node A *is written* $A \preceq B$. *By definition:* $A \preceq A$. *Also,* B *is a* strict ancestor *of* A *if* $A \preceq B$ *and* $B \neq A$. *This is is written* $A \prec B$.

As an example, consider the lock DAG shown in Figure 1(c). From our definition, node `B` is an ancestor of `D`. This is because some path from the root `A` to `D` must pass through `B`. It is also a strict ancestor, and so we write $D \prec B$.

DEFINITION 2. *If thread* T_A *accesses node* A *and thread* T_B *accesses node* B *then* $T_A \preceq T_B$ *if and only if* $A \preceq B$.

Again, in Figure 1(c), the read-access by thread T_3 to node `C` is an ancestor of the write-access by thread T_5 to node `F`, because `C` is an ancestor of `F`. This relationship is represented as: $T_5 \preceq T_3$.

DEFINITION 3. *Two accesses are related by the* may happen in parallel (MHP) *relation if the accesses are in separate thread instances.*

Since each of the accesses depicted in our example occur in separate threads, any access may happen in parallel with every other access.

DEFINITION 4. *A conflict exists if two or more concurrent accesses occur to object references, the objects are aliased, and at least one of those accesses is a write. If one of a pair of conflicting references is not guarded by an atomic section, there is the potential of a race at runtime. Only non-conflicting accesses can run concurrently, without being guarded by atomic sections.*

We note that every access to a node in the lock DAG implicitly involves a read access to an ancestor. This is intuitively obvious since, by the definition of the term, paths to a given node must traverse through its ancestors. Given this observation, in Figure 1(c), the write-access by thread T_1 to node `B` conflicts with the write-access by thread T_4 to node `E`, because the `write` on node `E` would also involve an implicit `read` of node `B` by thread T_4. In addition, thread T_2 and thread T_4 never conflict, because neither thread has a write-access to a node being read or written by the other thread.

Pruning lock DAGs *If an ancestor relationship exists between accesses in two threads then the accesses need not be guarded by critical sections if, and only if, the ancestor access is a read access. If no ancestor relationship exists between the two accesses then they are allowed to run in parallel.*

Two accesses are allowed to run in parallel if and only if no data race exists between them. An access to a node implies a read-access to each of its strict ancestors. If one of those ancestors is also being concurrently written-to in another thread, then a data race exists between that pair of accesses and, by our definition, the two threads cannot be allowed to run concurrently without synchronization. Thus, if an ancestor relationship exists between

```
procedure constructLockDAG(EP, L)
input: EP: list of escape portals
input: L: list of all allocation sites
Map M = new Map()
for each l in L do
    D = new DAGNode(l)
    M.put(l, D)
end for
for each e in EP do
    E = M.get(e)
    call constructSubDAG(E, M, new List())
end for

procedure constructSubDAG(N, M, L)
input: N: current DAG node
input: M: Map of all DAG nodes
input: L: DAG nodes in current DAG
for each field f in N do
    for each allocation site a ∈ points-to(f) do
        R = READ-SET(a), W = WRITE-SET(a)
        A = M.get(a)
        Add R,W to A
        Add edge from N to A
        if A is NOT present in L then
            Add A to L
            call constructSubDAG(A, M, L)
        end if
    end for
end for
```

Figure 2. Lock DAG construction algorithm.

accesses that may be in two threads, and the ancestor access is a write, the accesses must be locked. We say these accesses are *multi-touched*. If every access to some subtree S of the lock DAG can execute concurrently without locks, then that subtree can safely be pruned away. Intuitively, locks are needed to guard against the possibility of data races occurring between competing accesses to some region of memory: they are not needed if data races cannot occur. Figure 3 shows the complete lock DAG pruning algorithm.

Lock DAG pruning example Figure 1 shows an example of a lock DAG before and after pruning. Figure 1(c) shows the DAG before pruning, with nodes labeled R_{T_i} and/or W_{T_i} to indicate read/write touches by thread T_i. From the figure, the only nodes being multi-touched are nodes B, C and E. Node B is being multi-touched because thread T_1 is writing to it, while threads T_2 and T_4 are reading from it (during the access of nodes D and E, respectively). Node C is being multi-touched because threads T_3 and T_5 are reading it concurrently. Node E is being multi-touched because thread T_1 is reading from it, while thread T_4 is writing to it. In this example, the shared accesses all occur in separate threads. If accesses occur in the same thread then the may happen

```
procedure lock-dag-prune(N)
input: N: current node
output: canBePruned: boolean value indicating whether N can be pruned
canBePruned = true
if N is write-touched by thread T_i then
    if N is read or write-touched by thread T_j then
        // This node cannot be pruned
        // if a data-race exists
        canBePruned = false
    end if
end if
for each child C_i of N do
    pruneChild = lock-dag-prune(C_i)
    if pruneChild == true then
        remove child C_i
    end if
    // cannot prune this node if cannot prune all its children
    canBePruned &= pruneChild
end for
return canBePruned
```

Figure 3. Lock DAG pruning algorithm.

in parallel rule (defined above) is applied first to determine whether accesses occur concurrently. A recursive, depth-first application of the pruning algorithm yields the following steps:

- Node D can be pruned because only thread T_2 accesses it. It is therefore pruned away. Node E cannot be pruned because there are multiple accesses to it and at least one of those accesses is a write: thread T_4 has a write-access to it, while thread T_1 has a read-access to it. Node E is now the only leaf node with B as an ancestor.
- Node B cannot be pruned because it has at least one child which cannot be pruned. The subtree rooted at Node B is therefore retained in the lock DAG.
- Node F can be pruned because only thread T_5 accesses it. Node F is therefore pruned away. Node C is now a leaf node.
- Node C can also be pruned because even though there are multiple concurrent accesses to it (threads T_3 and T_5 both read it), all of those accesses are read accesses. Per our algorithm, therefore, no write access that is an ancestor to the subtree rooted at C exists, and it can be safely pruned.

Figure 1(d) shows the lock DAG after applying the pruning algorithm.

Lock Generation Algorithm The following algorithm is executed once for every reference in an atomic section.

Input:

i. A: an access to a node N in the unpruned lock DAG represented as the path string $P = R.f_1.f_2.\ldots.f_mN$ from the root R of the unpruned lock DAG to N.

ii. The pruned lock DAG T'. Let n_{f_i} represent a node in the lock DAG for the type of some field f_i.

Algorithm: Initially, $\alpha = R$.

1. If $\alpha == N$ then if access A is a `read` emit `readLock(A)` and exit the algorithm else emit `writeLock(A)` and exit the algorithm.
2. If n_α has a write access by some other thread to it then emit `readLock(`α`)`.
3. Let f_i = next path component in P. If node n_α does not have an edge $E = (n_\alpha, n_{f_i})$ then exit the algorithm. If edge E exists then set $\alpha = f_i$ and goto step (1).

Continuing the example in Figure 1, assume that we wish to generate the lockset for the write-access $A.B.D$ by the thread T_2. The path P is given by the string $P = A.B.D$ and the target node, (N in the algorithm above), is set to D. The first component in the path P is the node A, which is the root of the lock DAG. The algorithm sets $n_\alpha = n_A$. In step (1), the check if $\alpha == N$ fails, as does the check in step (2) to see if any write-access to node n_α exists. The algorithm thus proceeds to step (3) where it extracts the next component in the path, B. It determines that that the edge $E = (n_A, n_B)$ exists. The algorithm sets $\alpha = B$ and returns to step (1). Again, in step (1), $\alpha \neq N$ and the algorithm proceeds to step (2), where it determines that node n_B has a write-access by a different thread (T_1) and so it emits a `readLock(B)` and proceeds to step (3). At step (3) the algorithm extracts the next component in the path (D), determines that the edge (B, D) does not exist (having been pruned away) and so exits.

We emphasize that locking a node in a path involves (read)locking every strict ancestor of that node. Thus, locking the node C, in the path $A.B.C$ implicitly involves locking nodes A and B. More specifically, the `readLock(B)` generated in the example above effectively induces a `readLock(A)` first (since A is an ancestor of B in the path $P = A.B.D$). This path-locking is generated automatically by the locking library.

The example given in Figure1 generates only a single lock call per atomic section. In the more general case, a given atomic section could be guarded by multiple lock calls. To prevent deadlock, we use a strict two-phase locking protocol and, in addition, canonicalize lock strings lexicographically. The lock string is the complete lock DAG path to the node being locked. Also, locks generated by the inner atomic section(s) of nested atomic scopes are automatically hoisted up to the outermost atomic scope.

It should be noted that this DAG traversal is performed at compile-time, and is done once per shared reference in the program.

4.2.2 Lockset optimization

The lock pruning algorithm may cause redundant locks to be generated, i.e. locks that enforce a weaker constraint than another already generated lock. An example of this is generating a read and write lock on the same object.

To eliminate such unnecessary lock calls, the compiler uses a thread-local lockset optimization heuristic. This heuristic uses a copy of the lock DAG from the alias analysis phase (from which all read/write information has been removed) to determine if any read or write lock calls are extraneous and can safely be discarded. Note that this decision making is purely thread-local: no attempt is made at any global lockset optimizations. Briefly, the heuristic makes the following decisions: (i) A write lock to a node N obviates the need to read/write lock any node that N is a strict ancestor of, and (ii) A read or write lock to N obviates the need to read lock any (strict) ancestor of N. To accomplish these objectives, the lockset optimization procedure begins by first marking every node that is read or write locked by the current thread. It then iterates, in a depth-first fashion, over the nodes, checking them against the heuristic. It prunes (i.e. removes) every descendent of a node that is write locked. It also removes the read marking of every node that is a (strict) ancestor of a node that is also read (or write) locked. At the end of the procedure, the algorithm collects all of the remaining read and write marked nodes and emits the new locksets. Continuing the example in Figure 3(d), the write lock by thread T_1 on node $A.B$ obviates the need for T_1 to read lock node $A.B.E$, while the write lock by thread T_4 on node $A.B.E$ obviates the need for T_4 to read lock node $A.B$. Thus, the final locksets as emitted by the optimization procedure are: (i) $R_{T_1} = \{\emptyset\}, W_{T_1} = \{A.B\}$, (ii) $R_{T_2} = \{A.B\}, W_{T_2} = \{\emptyset\}$, (iii) $R_{T_3} = \{\emptyset\}, W_{T_3} = \{\emptyset\}$, (iv) $R_{T_4} = \{\emptyset\}, W_{T_4} = \{A.B.E\}$ and (v) $R_{T_5} = \{\emptyset\}, W_{T_5} = \{\emptyset\}$

4.3 Refinements to the basic algorithm

4.3.1 Array accesses

SPARK labels allocation nodes based on their respective sites. Individual elements of an array allocated within a loop, for example, share a common allocation site and are effectively aliased to each other (because their points-to sets contain the same allocation node). As a consequence, accesses to different elements of an array are treated as accesses to the common allocation node. This

can lead to overly conservative locksets. Our compiler differentiates between such accesses by annotating array reads and writes with the subscript expression used to index into the array. The analyzer in turn uses this subscript expression to decide whether different accesses to the array are independent. In addition, during the code generation phase, when at least one subscript expression for an array is invariant in the atomic section, the compiler can perform locking at a finer granularity than locking the entire array. In this case, the analyzer creates an array of locks (*lock-array*), and maps the subscript expression into the array of locks. We note that for space efficiency the lock-array can be smaller than the array being synchronized, or can be reused across several arrays, with a possible loss of concurrency. In our current implementation, the lock-array is of a constant size (32 elements). The appendix contains an example of a lock-array.

The analyzer treats locks generated on values derived from concurrent hashmap operations similarly, creating a lock-array for such situations (after verifying that the *key* value used in the operation is invariant over the atomic scope) and indexing into the lock-array (with the hash value of the *key* as the subscript expression). If the *key* value used is not in scope at the beginning of the atomic section (or is not invariant over the atomic scope) then this special fine-grained *per-key* locking optimization cannot take place and the analyzer generates a coarse-grained lock on the entire hashmap.

4.3.2 Ensuring strong atomicity

The PAG building phase builds up information about which references are being accessed. We augment this information-gathering phase with line numbers and enclosing atomic scope information. During the propagation phase, the analyzer tracks which of these references are EP-reachable. Accesses to EP-reachable references which fall outside of an atomic scope are tagged as such. A warning is emitted for every such reference, and lock/unlock function calls are generated as required. Results comparing the net effect of enforcing strong atomicity are given in Section 5.4.

4.3.3 Exploiting mutating smart data structures

Operations which return values from the data structure may do so either by first removing the value from the data structure and then returning it, or by leaving the underlying data structure unchanged. We refer to the former as *mutating* `GET` operations and the latter as *non-mutating* `GET` operations. Examples of mutating `GET` operations include concurrent stack `pop` operations, while non-mutating `GET` operations would include stack `peek` and hashmap `get` operations. Needless to say, having smart data structures greatly simplifies the detection of mutating `GET` operations.

If it can be proven by the compiler that every datum added to the data structure is unique, and that the `GET` is mutating, the compiler knows that the objects returned by repeated `GET` operations are not aliased, and accesses to them need not be synchronized with one another. An example of when this can occur, and be detected by compiler analysis, is the initialization of a data structure within a loop where every data structure value (perhaps read from a file) is placed in a unique object and `PUT` into the data structure. Benchmark results, presented in Section 5.3 show that substantial performance improvements are possible with this optimization.

4.4 Extending the smart data structure library

In Section 3.2 we saw how the alias analyzer uses semantic knowledge of concurrent data structures to propagate alias information. The analyzer requires two critical pieces of information to do so. First, it needs to know if a given reference is an instance of a smart data structure. Second, it needs to know whether the operation being performed on the data structure is a `GET` or a `PUT`. Given these two pieces of information, the analyzer can then propagate points-to information by using the transfer functions of Table 1. In addition, Section 4.3.3 illustrated how certain optimizations can be introduced for *mutating* `GET` operations. Because the fundamental nature of operations (i.e. whether it inserts data into or removes data from the data structure) is thread-agnostic (a stack `push` is semantically a `PUT` operation, and a stack `pop` is semantically a *mutating* `GET` operation, regardless of the number of threads accessing the stack), the job of the system architect is reduced to merely providing the name of the concurrent data structure class, the names of the various methods that perform its operations, and the semantic nature of those operations (mutating/non-mutating `GET` or `PUT`). Figure 4 shows the grammar used to notify the analyzer of the presence of various concurrent data structures, while Figure 5 gives an example of the specification of some common current data structures. The `isArray` qualifier tells the analyzer to maintain a per-index list of references, and is commonly used in hashmaps and other data structures which store data in $<key, value>$ pairs. Omitting this specifier will still lead to correct, if overly conservative, lock allocations.

```
class ⇒ class name (isArray)? methodList
name ⇒ string
methodList ⇒ (method)+
method ⇒ method name methodSpec
methodSpec ⇒ typeSpec mutatingSpec
typeSpec ⇒ opType = GET | PUT
mutatingSpec ⇒ isMutating = (true | false)
```

Figure 4. Concurrent Data Structure specification grammar.

```
class ConcurrentStack
    method push opType = PUT isMutating = true
    method pop opType = GET isMutating = true
    method peek opType = GET isMutating = false
class ConcurrentQueue
    method pushBack opType = PUT isMutating = true
    method popFront opType = GET isMutating = true
class ConcurrentHashMap isArray
    method put opType = PUT isMutating = true
    method get opType = GET isMutating = false
    method remove opType = GET isMutating = true
```

Figure 5. Example specification of some concurrent structures.

5. Experimental Evaluation

This section presents the experimental evaluation of the systems studied, including the experimental methodology, quantitative performance results, detailed analysis, and discussion.

5.1 Methodology

The strategies discussed in this paper are evaluated using five benchmarks from the Stanford Transactional Applications for Multi-Processing (STAMP) benchmark suite [2], the SPECjbb2000 benchmark [22] and a benchmark derived from Doug Lea's Bounded Buffer benchmark [15]. We have modified Soot to output either C++ code or Java code, as required by the benchmark. In what follows, we refer to the parser, alias analyzer and code generator as SmartLock.

The STAMP benchmarks were designed for transactional memory systems, but have also become popular as a benchmarking tool for conservative concurrency control. The set of STAMP benchmarks used in this evaluation is identical to that seen in the lock-generation work of Cherem et al. [4], except that Intruder is used instead of Bayes (Bayesian network learning); the STM implementation provided with the current distribution of Bayes experienced a segmentation fault during this study. The five STAMP benchmarks considered were: (1) *genome*: A gene sequencing program using Hashmaps and Linkedlists, (2) *intruder*: A network intrusion detection program using Hashmaps, FIFO queues, Linkedlists and extensible Vectors, (3) *kmeans*: A K-means clustering program, (4) *labyrinth*: A maze routing program using FIFO queues, Linkedlists and Vectors and (5) *vacation*: A travel reservation program using Hashmaps and Linkedlists.

The STAMP programs synchronize using explicit atomic sections. This paper evaluates four implementations of each program:

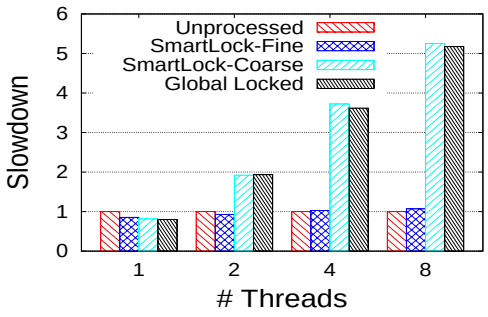


Figure 7. Relative slowdowns in SPECjbb2000.

a C version using the TL2 software transactional memory system (called STM) [5], a C version that implements all atomic sections using a single global lock (Single-Lock), a C version in which atomic sections are manually converted to fine-grained locks using detailed analysis by the programmer (Multi-Lock), and a SmartLock port of the C code. As discussed in Sections 2, 3 and 4, SmartLock's compiler analyzes the code to generate fine-grained locks for the programmer-coded atomic sections, exploiting knowledge of SmartLock's memory model and primitive data structures for efficiency.

The SPECjbb2000 benchmark [22] is a widely-used Java server benchmark, consisting of several threads performing various transactions on a backend database. This benchmark has been used previously in other works on transactional systems [3, 8]. Although the benchmark itself is considered to be embarrassingly parallel [3], its combination of a complex B-tree database structure and non-conventional access patterns have been shown to be difficult to analyze in a static context [8], and is a powerful example of how compiler knowledge of data structures can greatly improve lockset allocations. We evaluate four versions of the SPECjbb2000 benchmark: an unmodified, unanalyzed version which preserves the original synchronized methods and blocks (Unprocessed), a SmartLock version which uses coarse-grained atomic blocks (SmartLock-Coarse), a SmartLock version which uses finer-grained atomic blocks (SmartLock-Fine) and one which uses a single, global lock to protect against concurrent accesses (Global).

The benchmarks discussed above do not make use of mutating operations. To showcase our mutating `GET` optimization strategy (Section 4.3.3), therefore, we also include a benchmark derived from Doug Lea's Bounded Buffer program [15]. The benchmark consists of a number of producer threads repeatedly inserting newly allocated objects onto a bounded buffer, and a number of consumer threads repeatedly modifying objects extracted from the buffer via mutating reads. We compare the performance of a version which uses the mutating `GET` optimization (Optimized) versus one that does not (Unoptimized).

The experiments are all performed on a Dell Poweredge 2950 server with two 1.8 GHz quad-core Intel Xeon E5320 processors based on the Core 2 microarchitecture (8 cores total). This system has 16 GB of system RAM and 4 MB L2 caches shared between pairs of processors. This system runs Linux kernel 2.6.18 and GNU C library 2.3.6 with the Native POSIX Threads Library in the Debian AMD64 distribution. All C/C++ programs were compiled using the GNU `gcc` compiler at optimization level: `-O3`. We used version 1.5.0_10 of the Sun JVM to run the Java versions of the benchmarks.

5.2 Base results and analysis

Figure 6 presents experimental results showing the performance and scalability of five STAMP benchmarks under each of the implementations described above. In each case, the X axis represents the number of threads while the Y axis shows the execution time in seconds on the given platform. Note that Labyrinth has no Multi-Lock implementation as its irregular access patterns made its atomic sections unamenable to fine-grained locking.

Across all five benchmarks, STM has substantial overheads related to tracking read and write sets: these overheads degrade single thread performance by a minimum of 27% (on Labyrinth) and a maximum of 822% (on Vacation). STM does provide speedup in all of the STAMP benchmarks studied. Single-Lock, on the other hand, yields speedups only for Genome and Kmeans, and only up to 4 threads in both cases. Multi-Lock provides the best performance and scalability for each of the four applications for which it applies. SmartLock achieves performance nearly identical to Multi-Lock (and far better than STM) in Genome, Intruder, and Kmeans. SmartLock's performance lags Multi-Lock in Vacation, but still exceeds Single-Lock and STM substantially. Since SmartLock's atomic sections are implemented using locks, it does not experience the overheads of STM. In Labyrinth, however, the performance of SmartLock matches Single-Lock exactly, and STM yields the best performance since its speedups from dynamic concurrency compensate for the overheads of data tracking.

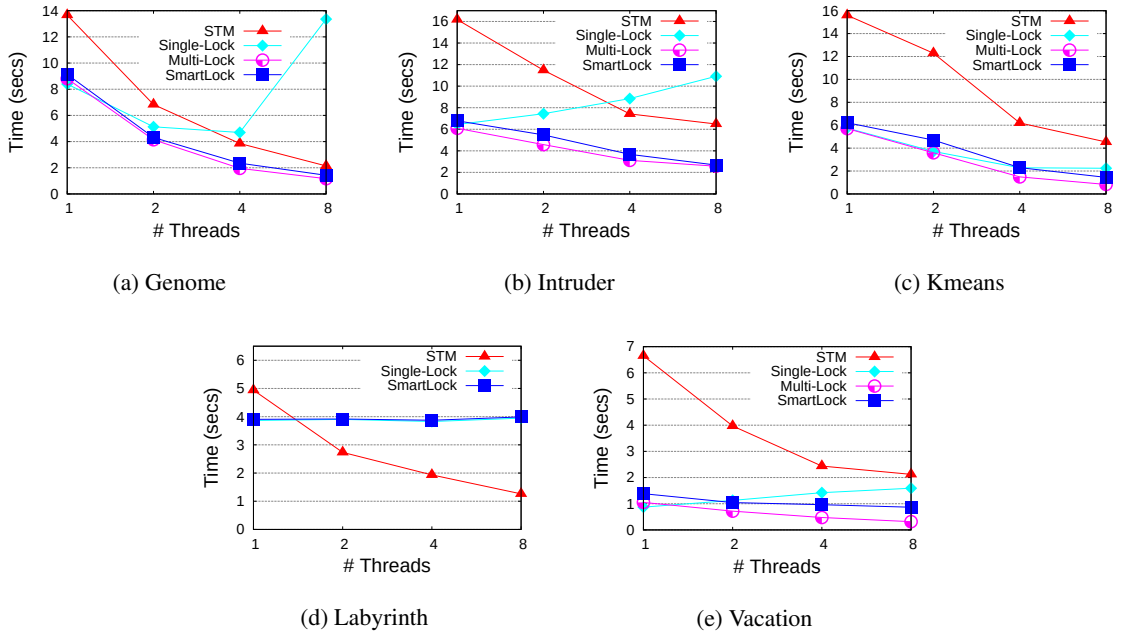
Detailed analysis of these results shows that the behavior of the different implementations depends largely on the data accessed within the atomic sections. Genome and Intruder behave similarly since the primary data structure accessed in the atomic sections is a smart Hashmap, understood by the SmartLock compiler. Any given atomic section operates on no more than one Hashmap, using no more than one key, and doing either `get` or `put` (but not both). The SmartLock compiler automatically assigns separate locks to atomic sections using the Hashmap key to index into an array of per-bucket locks. In this way, atomic sections based on different Hashmap keys can execute concurrently since they are unrelated.

Kmeans does not use smart data structures, but it has short atomic sections that access arrays. Each atomic section in the clustering algorithm repeatedly accesses one row in each of two 2-D arrays. Since the row-index is constant in any given atomic section, SmartLock can lock only that row of each array as described in Section 4.3.1. This allows the SmartLock version to outperform the Single-Lock version at eight threads. The hand-coded Multi-Lock version outperforms the SmartLock version, however, because the SmartLock version performs *two* locking operations per atomic section (one per array), as opposed to the hand-coded version, which only performs one. This is because the SmartLock analyzer conservatively assigns a separate entry in the lock DAG for each array, even though the row-index is common to both arrays and the arrays are always used together. A lock optimization strategy could detect such cases and help to optimize away one of the locks [12].

The atomic sections of Labyrinth access data in an irregular way that is not statically analyzable using the methods described in Section 3 or by a detailed manual inspection of the code. Thus, the only locking scheme used here is effectively Single-Lock, whether in C or generated by the SmartLock compiler.

Vacation lies somewhere between the extremes of the other benchmarks. Atomic sections in this code perform tasks such as atomically getting one or more entries from one hashmap and putting one or more entries into another. Unlike Labyrinth, the code can be analyzed by a sufficiently sophisticated programmer who must establish a canonical ordering among the different locks used in the program and then acquire them in a particular order to allow for a Multi-Locked implementation. Unlike Genome and Intruder, however, the SmartLock compiler only has limited analytical success, as it must generally make conservative assumptions about concurrent map `gets` and `puts` in certain atomic sections. As a result, SmartLock can generate some fine-grained locks based on analysis of concurrent map operations in some atomic sections, but cannot exploit techniques like per-key locks in others.

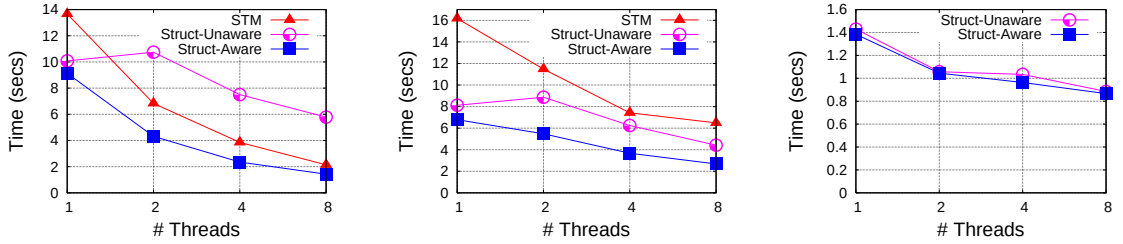
In summary, SmartLock achieves nearly all of the performance of the Multi-Locked implementation for three of the four benchmarks that have a Multi-Locked version, but does this while allowing the programmer to specify only atomic sections rather than fine-grained locks. Conservative synchronization allows SmartLock to avoid the overhead of access tracking associated with STM. Con-



(a) Genome (b) Intruder (c) Kmeans

(d) Labyrinth (e) Vacation

Figure 6. STAMP benchmark performance and scalability results.



(a) Genome (b) Intruder (c) Vacation

Figure 8. Impact of data structure knowledge on STAMP benchmarks.

sequently, the SmartLock solution also outperforms STM for four out of five benchmarks (and substantially so for three of them). This is despite the fact that these benchmarks were drawn from a suite specifically designed to showcase algorithms appropriate for transactional memory systems.

Figure 7 presents the results from the SPECjbb2000 benchmark, under each of the implementations described above. The X axis represents the number of threads, while the Y axis depicts normalized slowdowns against the unprocessed implementation. The coarse grained implementation features atomic blocks around the outermost transactions and, as such, cannot take advantage of the SmartLock's ability to use per-key locking for smart maps, leading to a reduction in concurrency and loss of performance. The finer grained implementation, on the other hand, places atomic blocks within the various transactions themselves, bringing into scope the various map `gets` and `puts` and allowing SmartLock to optimize locksets using an in depth knowledge of the smart data structures used. It consequently nearly matches the performance of the unprocessed implementation. The global locked implementation uses a single, global lock around each of the transactions and performs better than the coarse grained implementation due to lower locking overheads on the locks themselves (the coarse grained locking scheme uses a lock DAG while the global locked implementation does not).

5.3 Impact of data structure knowledge

Although previous work has also performed automatic lock generation from atomic sections, none has achieved performance superior to both Single-Lock and STM for STAMP benchmarks [4]. In contrast, SmartLock does so for four out of the five benchmarks considered – including two in which Single-Lock alone performs much worse than STM as the number of threads increases. This section explores why this happens, focusing on SmartLock's use of data structure knowledge.

Figure 8 shows the three benchmarks for which data structure knowledge played any role: Genome, Intruder, and Vacation. The graphs for Genome and Intruder show structure-aware and unaware versions of the SmartLock compiler, along with the STM performance results for comparison. In both of these cases, having a compiler that understands data structure implementations contributes substantially to the scalability of SmartLock's locking scheme. In Genome, automatic lock generation without data structure knowledge performs 170% worse than STM for eight threads. Without detailed knowledge of the Hashmap implementation, gets or puts to the underlying structure are conservatively considered

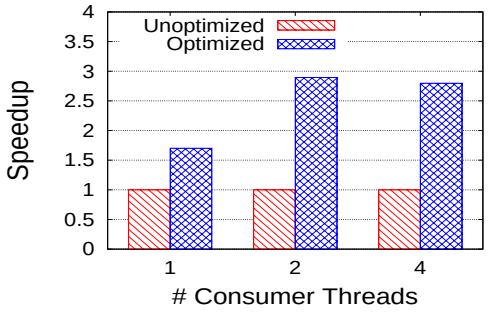


Figure 9. Relative speedups in Bounded Buffer benchmark demonstrating effectiveness of mutating `GET` optimizations.

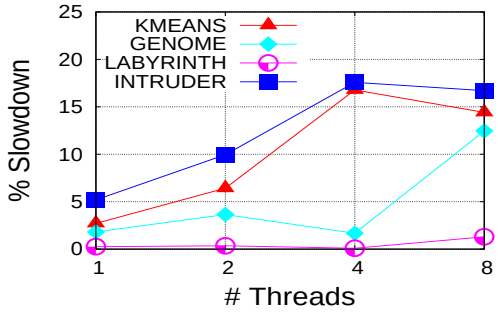


Figure 10. Performance impact of strong atomicity on STAMP benchmarks.

related. Even with interprocedural analysis, the compiler cannot make sense of the highly non-linear hashing function or the pointer-chasing as the Hashmap `get` and `put` methods traverse the hash buckets. For the same reason, the structure-unaware version of the SmartLock compiler also leads to code that is 55% slower than the structure-aware version for Intruder; however, this is still 56% faster than the STM version. In both cases, however, knowledge of data structure implementations provided substantial scalability advantages.

As discussed earlier, these benefits were muted in Vacation; although SmartLock outperforms the Single-Lock implementation of atomic sections, it does not use data structure knowledge for any performance-critical atomic sections. Both versions generated from the SmartLock code, however, substantially outperform the STM implementation (140% faster than STM at eight threads). We note that the STM line is not shown to make the range of the Y axis small enough to allow differences between the structure-aware and unaware versions to be discerned.

Figure 9 presents results showing the performance improvements that are enabled by a deeper semantic understanding of the operations being conducted on concurrent data structures. The X axis depicts the number of consumer threads repeatedly performing destructive reads on a bounded buffer, while the Y axis depicts normalized speedups of the implementations described above, relative to the unoptimized implementation. The unoptimized version does not differentiate between mutating and non-mutating `GET` operations, as discussed in Section 4.3.3 and consequently generates locks for unique values retrieved from the buffer. The optimized version, on the other hand, uses an in depth semantic understanding of the mutating nature of the `GET` operations being performed on the buffer and optimizes away locks derived from values being stored in the buffer, leading to dramatic performance improvements.

5.4 Strong Atomicity

All of the previous discussion assumes weak atomicity, just as in most current software transactional memory systems (including TL2) as well as current locking implementations. Hardware TM systems and some software TM systems, however, support strong atomicity. In weak atomicity, accesses are only guaranteed to be atomic relative to those that are also marked in other atomic sections (or, in the case of fine-grained locks, to those that are protected by the same lock). In strong atomicity, however, accesses within an atomic section must appear atomic relative to all other code.

As discussed in Section 4.3.2, the SmartLock compiler can also support strong atomicity by tracking any references that might derive from EPs. Analysis of the Vacation benchmark found no EP references outside of atomic sections. Although Labyrinth did have some EP accesses outside of atomic sections, none of them were in portions of the code that affected performance by more than 1%.

Figure 10 shows the performance impact of maintaining strong atomicity for the remaining codes: Genome, Intruder, and Kmeans. The implementations compared are both generated from the same SmartLock code, with one having the compiler generate weakly-atomic code and the other having the compiler perform the extra analysis required to generate strongly-atomic code. In each of these cases, maintaining strong atomicity causes the insertion of a small number of extra locks as certain values are found to be reachable from EPs. The performance impact of these locks is relatively greater as the number of threads increases, but is never more than 20% slower than the weakly-atomic version of SmartLock. Further, even with strong atomicity, each of these three benchmarks continues to scale, and the system still does not see the tracking overhead of STM.

These results show that SmartLock can provide strong atomicity in a conservative fashion by using only atomic sections and compiler-generated locks. Support for strong atomicity degrades application execution time by less than 1% for two benchmarks and by less than 20% for the others. SmartLock can thus support functionality that is fundamentally different from the STM and locking schemes considered without encountering undue overheads or burdening the programmer.

Note that in all of these benchmarks, there are no actual operations outside of atomic sections that race with protected operations; however, the compiler must make conservative assumptions if it cannot positively identify that two accesses are definitely to different addresses. Thus, a more precise analysis should help to further reduce the overhead of supporting strong atomicity, by generating fewer "false positives" and thereby adding fewer additional locks.

6. Related Work

Previous research has indicated that naively transactionalizing code which uses fine-grained locking may lead to deadlocks where none previously existed [13]. Our work is predicated on the assumption that the code being analyzed has correctly been transactionalized.

The field of conservative implementations of atomic sections has been receiving much attention of late ([4, 6, 8, 12, 17, 24], amongst others). This is motivated, at least in part, by the shortcomings of software transactional memory systems: a high per-access memory tracking overhead and an inability to handle I/O.

Cherem et al. provide a formal framework for reasoning about locks, and provide a lock inference tool which is parameterized on a locking scheme [4]. They implement one such scheme. Like with our system, they use a multi-granularity locking protocol to implement deadlock-freedom. However, unlike our system, their use of intention locks borrows from the database community [7]; this allows them to potentially extend their multi-granularity locking abstraction to handle lock-lattices. While our multi-granularity lock DAG is powerful enough to handle all of our benchmarks, further work would be needed for efficient implementations of some more general cases. Unlike us, they do not make any guarantees about

strong atomicity, nor do they support a compiler-aware data structure library to refine their analysis. As a consequence, their performance results are, on average, no better than using a single global lock to protect all critical sections.

McCloskey et al. require the programmer to provide manual lock annotations [17]. Their tool then ensures composability and deadlock freedom. They provide some support for multi-granularity locks. However, they do not support strong atomicity. In addition, their requirement of programmer-supplied lock annotations improves the performance of the final program, but also imposes an additional burden on programmers.

Emmi et al. and Hicks et al. do not require manual lock annotations [6, 12]. These works prove that the lock allocation decision problem is NP-complete and that the minimum lock allocation problem is NP-hard, respectively. Both attempt to first generate a lock allocation and to then optimize the allocation: Emmi et al. by modeling the lock allocation problem as a 0-1 ILP and Hicks et al. by coalescing locksets. We believe their work is complementary to ours, because our system could use their lock optimization strategies. Unlike us, they do not support strong atomicity, and they do not use data structure understanding to improve their analysis results.

Boyapati et al. provide a type-based system which requires the programmer to specify a partial order amongst equivalence classes of locks [1]. Their system then ensures that the order does not result in a deadlock. Unlike their work, our approach allows individual objects of some type to be shared or not shared, and does not require any programmer involvement in the lock allocation decision.

Zhang et al. model the problem of lock allocation using a concurrency graph abstraction [24]. They then successively produce a solution by first solving one subset of the graph via graph coloring and then other subsets by handling special cases and propagating that information onto the main graph. Halpert et al. attempt to infer lock assignments by building a critical section interference graph (which is initially fully-connected) and then successively refining it through a series of analyses [8]. They then assign locks based on which components of the graph are connected. Neither approach ensures strong atomicity, since both approaches rely on the interfering relationship between critical sections to determine lock allocations. In addition, neither approach uses compiler-awareness of data structures to refine analysis results.

In conclusion, none of the previous works on conservative atomicity enable high performance by using compiler awareness of data structure semantics while at the same time ensuring strong atomicity.

Software transactional memory (STM) systems rely on optimistic concurrency ([9, 11, 20, 21], amongst many others). They speculatively execute potentially conflicting operations and rely on sophisticated, and expensive, memory-tracking mechanisms to detect and recover from conflicts that occur at runtime. Conflicting accesses may be rolled back to preserve system consistency. STM systems almost uniformly suffer from high overheads due to the memory tracking requirements, and fail in the face of non-reversible operations (such as I/O). In addition, while there have been some STM systems with strong atomicity guarantees, they still suffer from the high tracking overheads inherent to all STM systems [3, 19, 21].

Transactional boosting, proposed by Herlihy et al., is a relatively new methodology which seeks to improve the throughput of existing, concurrent data structures using a combination of abstract locks, transactional conflict resolution and function inverses [10]. Unlike our technique, it doesn't interact with the compiler, provide strong atomicity to general accesses outside the data structure in question, or enable the use of data structure knowledge to improve analysis in the remainder of the code. It is, however, a useful tool to develop data structures to be targeted by our compiler.

The use of smart data structures was inspired by magic functions in the Jikes RVM and the use of semantic inlining in the Ninja numerically intensive Java project [14, 23]. These efforts utilized knowledge of data structures to implement functionality that is not expressible in Java and to enable better dependence analysis by enabling multi-dimensional arrays in Java.

7. Conclusions

This paper shows that a combination of compiler understanding of standard data structure semantics and a restricted sharing model allows effective and efficient compiler generation of locks, easier programmer understanding of what data may be shared, and the ability to have low-overhead strong atomicity and compiler warnings when potentially escaping values are not locked by the programmer. The implementation of these ideas allows compiler-generated locking to outperform software transactional memory (STM) for four out of five benchmarks, achieving speedups of 0.83 to 5.9. This performance nearly matches the performance of manually-inserted fine-grained locks while allowing the programmer to merely specify atomic sections; compared to STM, our approach avoids overhead related to tracking and recovery from conflicts and also enables atomic sections to include I/O and other operations that cannot be safely speculated. Our compiler's advanced compiler analysis also allows for strong atomicity to further improve encapsulation of atomic sections. In addition, incorporating data structure semantics into the compiler allows us to successfully analyze benchmarks which were previously difficult to analyze statically.

While properly synchronizing parallel programs will always take some programmer effort, that effort can be minimized with the right combination of programming language features and high-performance implementations of atomicity. While this paper demonstrates significant progress towards reducing the complexity of writing high-performance parallel programs, this is not a solved area. It is clear, however, that just as structured programming and type safe languages led to programs that were easier for programmers to understand and maintain, easier for compilers to analyze and optimize, and provide good performance, changes in language semantics and exploiting the known semantics of standard libraries can help to achieve those same goals for parallel programs.

References

[1] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230, New York, NY, USA, 2002. ACM.

[2] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[3] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. *SIGPLAN Not.*, 41(6):1–13, 2006.

[4] S. Cherem, T. M. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In R. Gupta and S. P. Amarasinghe, editors, *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*, pages 304–315. ACM, 2008.

[5] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *20th International Symposium on Distributed Computing*, pages 194–208, 2006.

[6] M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 291–296, New York, NY, USA, 2007. ACM.

[7] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *Readings in database systems (2nd ed.)*, pages 181–208. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[8] R. L. Halpert, C. J. Pickett, and C. Verbrugge. Component-based lock allocation. In *PACT '07: International Conference on Parallel Architectures and Compilation Techniques*, pages 353–364, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[9] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.

[10] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 2008. ACM.

[11] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM Press.

[12] M. Hicks, J. S. Foster, and P. Prattikakis. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[13] O. S. Hofmann, C. J. Rossbach, and E. Witchel. Maximum benefit from a minimal htm. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 145–156, New York, NY, USA, 2009. ACM.

[14] Jikes RVM web page on magic functions. http://jikesrvm.org/Magic, last accessed Nov. 12, 2008.

[15] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, second edition, November 1999.

[16] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[17] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 346–358, New York, NY, USA, 2006. ACM.

[18] M. Moir, K. Moore, and D. Nussbaum. The adaptive transactional memory test platform: a tool for experimenting with transactional code for rock (poster). In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 362–362, 2008.

[19] F. T. Schneider, V. Menon, T. Shpeisman, and A.-R. Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. *SIGPLAN Not.*, 43(10):181–194, 2008.

[20] N. Shavit and D. Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[21] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 78–88, New York, NY, USA, 2007.

[22] The Standard Performance Evaluation Corporation. SPECjbb2000 Benchmark. At `http://www.spec.org/jbb2000`, 2000.

[23] P. Wu, S. P. Midkiff, J. E. Moreira, and M. Gupta. Efficient support for complex numbers in Java. In *Proceedings of the 1999 ACM Java Grande Conference*, pages 109–118, 1999. IBM Research Report RC21393.

[24] Y. Zhang, V. C. Sreedhar, W. Zhu, V. Sarkar, and G. R. Gao. Minimum lock assignment: A method for exploiting concurrency among critical sections. In *21st Annual Workshop on Languages and Compilers for Parallel Computing (LCPC '08)*, 2008.

[25] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 145–157, New York, NY, USA, 2004. ACM.

A. Example

In the interests of completeness, we present here a more concrete example to illustrate how code is generated for a sample program. Figure 11 shows a sample input program. Figure 12 shows the code generated (including the calls to the locking library). The `main` function is shown here for completeness.

As mentioned in Section 3.1, the alias analysis is initiated at allocation sites, which are regions of the program where new heap-space is allocated (typically via the `new` operator). Allocation sites allocate PAG nodes, called allocation nodes (or *AllocNodes*). AllocNodes are propagated via simple allocation edges of the form: $a = newA()$.

Consider Figure 11. Initially, the allocation edges in lines [7 – 9] are handled. AllocNodes corresponding to each of those sites are created, and the points-to sets of the variables being assigned to are populated. The AllocNodes are labelled by their respective allocation sites. We model these sites by their relative order. Thus,

```
 1: class Memory
 2:     var int []a
 3:     var SmartMap map
 4: end class
 5: class Main
 6:     procedure main()
 7:         Memory m1 = new Memory()
 8:         m1.a = new int[10]
 9:         m1.map = new SmartMap()
10:         MyThread t1 = new MyThread(m1), t2 = new MyThread(m1)
11:         t1.run(); t2.run();
12:     end procedure
13: end class
14: class MyThread extends Thread
15:     var Memory mem
16:     procedure MyThread(Memory m)
17:         // Thread constructor, "m" is (possibly) thread-escaping
18:         mem = m
19:     end procedure
20:     procedure run()
21:         // Thread run function
22:         int i=0, j=3
23:         int []a1
24:         atomic {
25:             // Atomic Section A1
26:             a1 = mem.a
27:             a1[i] = j
28:         } // end atomic
29:         atomic {
30:             // Atomic Section A2
31:             mem.map.put(i, a1)
32:         } // end atomic
33:     end procedure
34: end class
```

Figure 11. Input Program

```
 1: class MyThread extends Thread
 2:     ...
 3:     procedure run()
 4:         ...
 5:         {
 6:             // lock index i in lock-array
 7:             array-write-lock(Mem.a, i)
 8:             {
 9:                 a1 = mem.a
10:                 a1[i] = j
11:             }
12:             array-write-unlock(Mem.a, i)
13:         }
14:         // No locks needed for smart map insertion
15:         // because the map handles locking internally
16:         {
17:             mem.map.put(i, a1)
18:         }
19:     end procedure
20: end class
```

Figure 12. Generated Code

the first AllocNode (at line 7) is called `AllocNode-1`, the second (at line 8) is `AllocNode-2` and so on.

Within the `run` function, the individual edges are traversed and the points-to sets are updated according to the transfer functions defined in Table 1. In addition to the points-to sets, the read/write sets are populated as follows: in atomic section `A1` in Figure 11, line 26 involves a read of `mem.a`, while line 27 involves a write to subscript `i` of array `a1`, which points to AllocNode-2. Accordingly, the write-set of AllocNode-2 is updated with the current context. The analyzer also records the subscript expression used in this write to enable the lock-array optimizations described in Section 4.3.1. Line 31, in atomic section `A2`, involves a `put` operation on a smart hashmap which, as described in Section 3.2, involves a write to the underlying $data$ field. Because a hashmap is inherently an array-based data structure, the compiler again notes the key value passed to the function. Table 2 details the individual steps discussed above.

Line #	Action
7	PointsTo(m1) = AllocNode-1
8	PointsTo(m1.a) = AllocNode-2
9	PointsTo(m1.map) = AllocNode-3
18	PointsTo(mem) = AllocNode-1
26	PointsTo(a1) = AllocNode-2 read-set(AllocNode-2) = [26]
27	write-set(AllocNode-2[i]) = [27]
31	PointsTo(AllocNode3.data) = AllocNode-2 write-set(AllocNode3[i]) = [31]

Table 2. Table of actions

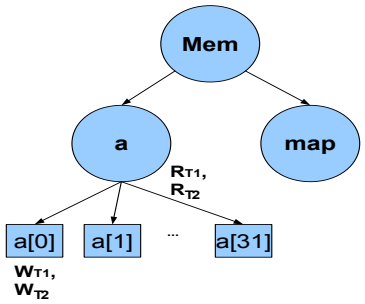


Figure 13. Lock DAG

In the interests of brevity, contexts are represented by the respective line numbers.

Figure 13 shows the lock DAG structure created for the program given in Figure 11. As explained in Section 4.1, a W_{T_i} next to a node indicates a write access to that node by thread T_i, while an R_{T_i} indicates a read. The square nodes are the (internal) lock-array nodes. Figure 12 shows the code generated after the lock DAG creation and optimization phases of Section 4.1 and 4.2.1. Lines 7 and 12 contain the lock and unlock calls. The calls take two arguments: a path in the lock DAG and an index into the lock-array. We emphasize that write-locking a node in a path implicitly read-locks every ancestor of that node in the path. Thus, the array write-lock call of node `Mem.a` at line 7 first read-locks node `Mem` and then write-locks lock-array node `a[0]`.

The comments on lines [14 – 15] in Figure 12 describe another advantage of incorporating data structure information into the complier: lock generation. In this scenario, the compiler knows that the smart hashmap is internally protected against concurrent accesses (using the specifications detailed in Section 4.4 and Figure 4, and illustrated in Figure 5). Thus, isolated reads and writes to these data structures need not be externally protected via any concurrency control mechanism, and no locks are generated for the atomic section.