Static Analysis of Atomicity for Programs with Non-Blocking Synchronization*

Liqiang Wang
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794, USA
liqiang@cs.sunysb.edu

Scott D. Stoller
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794, USA
stoller@cs.sunysb.edu

ABSTRACT

In concurrent programming, non-blocking synchronization is very efficient but difficult to design correctly. This paper presents a static analysis to show that code blocks are atomic, i.e., that every execution of the program is equivalent to one in which those code blocks execute without interruption by other threads. Our analysis determines commutativity of operations based primarily on how synchronization primitives (including locks, load-linked, store-conditional, and compare-and-swap) are used. A reduction theorem states that certain patterns of commutativity imply atomicity. Atomicity is itself an important correctness requirement for many concurrent programs. Furthermore, an atomic code block can be treated as a single transition during subsequent analysis of the program; this can greatly improve the efficiency of the subsequent analysis. We demonstrate the effectiveness of our approach on several concurrent non-blocking programs.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Verification, Algorithms, Design

Keywords

Atomicity, Non-Blocking, Lock-Free, Synchronization, Static Analysis, Verification, Linearizability.

1. INTRODUCTION

Many concurrent programs use blocking synchronization primitives, such as locks and condition variables. *Non-blocking synchronization primitives*, such as Compare-and-Swap, and Load-Linked / Store-Conditional, never block (suspend execution of) a thread. Non-blocking (also called "lock-free") synchronization is becoming increasingly popular, because it offers several advantages, including better performance, immunity to deadlock, and tolerance to priority inversion and pre-emption [12, 13].

An important use of non-blocking synchronization is in the implementation of non-blocking objects. A concurrent implementation of an object is *non-blocking* if it guarantees that some process can complete its operation on the object after a finite number of steps of the system, regardless of the activities and speeds of other processes [7]. Non-blocking synchronization is also used to implement blocking objects, such as spin locks.

Algorithms that use non-blocking synchronization are often subtle and difficult to design and verify. This paper presents a static analysis to show that code blocks are *atomic*. Informally, a code block is atomic if every execution is equivalent to one in which the code block is executed serially, *i.e.*, without interruption by other threads. Atomicity is well known in the context of transaction processing, where it is sometimes called *serializability*.

Atomicity is an important correctness requirement for many concurrent programs. Furthermore, each atomic code block can be treated as a single transition during subsequent static or dynamic analysis of the program; this can greatly improve the efficiency of the subsequent analysis.

This paper presents a conservative intra-procedural static analysis to infer atomicity. We build on Flanagan *et al.*'s work on atomicity types [3, 4] and purity [2] in order to develop an analysis that is much more effective for programs that use non-blocking synchronization primitives.

Our analysis first classifies all actions (*i.e.*, operations) in the program into different types based on their commutativity and atomicity, which are determined based primarily on how locks and non-blocking synchronization is used in the program. The analysis then combines those atomicity types to determine the atomicity types of larger code blocks. We formalize the analysis for a language that allows declaration of top-level procedures (as in an API) that implicitly get concurrently called by the environment. The language does not allow explicit procedure calls; internal procedures are inlined, and we do not handle recursion. The analysis can be extended to be inter-procedural.

The analysis is incomplete (*i.e.*, sometimes fails to show atomicity), but is effective for common patterns of non-

*This work was supported in part by NSF under Grant CCR-0205376 and ONR under Grant N00014-02-1-0363.

PPoPP'05, June 15–17, 2005, Chicago, Illinois, USA.

blocking synchronization, as demonstrated by the applications in Section 6.

It applies equally to non-blocking objects and blocking objects. We did not implement our analysis algorithm into an automated tool, but we applied it manually to four interesting non-trivial non-blocking programs, as described in Section 6. Although in two cases we must modify the algorithm before applying our analysis, we consider the results encouraging, since we do not know of any other algorithmic (*i.e.*, automatable) analysis that can show atomicity of the same (or larger) code blocks in the modified or original versions. We believe our analysis provides a useful method for manual verification of atomicity, as well as being suitable for automation.

The rest of the paper is organized as follows. Section 2 discussed related work. Section 3 describes background. Section 4 defines pure loops, and gives algorithm for identifying pure loops. Section 5 presents the approach to infer atomic code blocks based on locks and non-blocking synchronization. Section 6 shows the application of our approach on four non-blocking algorithms.

2. RELATED WORK

Linearizability [8] is a correctness condition for objects shared by concurrent processes. Informally, a concurrent object o is linearizable if and only if each concurrent operation history h for o is equivalent to some legal sequential history s, and s preserves the real-time partial order of operations in h. The equivalence is based on comparing the arguments and return values of procedure calls. Legality is defined in terms of a specification of the correct behavior of the object. We focus on proving atomicity rather than linearizability, because atomicity does not require a correctness specification. Atomicity can help establish linearizability: first show that the concurrent implementation executed sequentially (*i.e.*, single-threaded) satisfies the sequential specification, and then apply our analysis to show the procedures of the implementation are atomic.

Gao and Hesselink [5] used simulation relations to prove that a non-blocking (called lock-free in [5]) algorithm refines a higher-level (coarse-grained) specification. Using the PVS theorem prover, they proved correctness of algorithms similar to the ones in Sections 6.2 6.3. The proofs took a few man-months and are not easily re-usable for new algorithms.

Flanagan *et al.* developed type systems [3, 4] based on Lipton's reduction theorem [10] to verify atomicity. Wang and Stoller [17] and Flanagan *et al.* [1] developed runtime algorithms to check atomicity. All of this work focuses on locks and is not effective for programs that use non-blocking synchronization.

Flanagan *et al.* extended their atomicity type system with a notion of purity [2]. A code block is pure if, when it terminates normally, it does not change the program state. Non-blocking programs often contain code blocks that abort an attempted update to a shared variable if the variable was updated concurrently by other threads; these code blocks are often pure according to our definition of purity, which generalizes the definition in [2] by taking into account liveness of variables and use of unique references. The type system in [2] can show atomicity of simple non-blocking algorithms but not of any of the algorithms in Section 6, because it does not accurately analyze usage of non-blocking synchronization primitives; for example, it has no analogue of the notions of "matching read" or "matching LL" in Section 5.2, and does not analyze exceptional variants (also defined in Section 5.2) of a procedure separately.

Atomicity used to optimize model checking can be regarded as a partial-order reduction [14], *i.e.*, a method for exploiting commutativity to reduce the number of states explored by a verification algorithm. For non-blocking algorithms, traditional partial-order reductions are less effective than our analysis, because they do not distinguish left-movers and right-movers, and they focus on exploiting commutativity of operations with little regard for the context in which the operations are used, while our analysis considers in detail the context (surrounding synchronization and conditions) of each operation.

The model-checking (*i.e.*, state-space exploration) algorithm in [15] dynamically identifies transactions, which correspond roughly to executions of atomic blocks. Their algorithm relies on a separate analysis to determine commutativity of actions. An inter-procedural extension of our analysis could be used for this. This would allow their algorithm to be applied effectively to non-blocking programs.

3. BACKGROUND

3.1 Non-Blocking Synchronization Primitives

Non-blocking synchronization primitives include *Load-Linked* (LL) and *Store-Conditional* (SC), supported by PowerPC, MIPS, and Alpha, and *Compare-and-Swap* (CAS), supported by IBM System 370, Intel (IA-32 and IA-64), Sun SPARC and the JVM in Sun JDK 1.5.

LL($addr$) returns the content of the given memory address. SC($addr, val$) checks whether any other thread has written to the address $addr$ (by executing a successful SC on it) after the most recent LL($addr$) by the current thread; if not, the new value val is written into $addr$, and the operation returns `true` to indicate success; otherwise, the new value is not written, and `false` is returned to indicate failure. Another primitive VL (validate) is often supported. VL($addr$) returns `true` iff no other thread has written to $addr$ after the most recent LL($addr$) by the current thread.

For a run of a program, the *matching* LL (if any) for a SC(v, val) or VL(v) action is the last LL(v) before that action in the same thread. If there is no matching LL for a SC, the SC must fail.

CAS($addr, expval, newval$) compares the content of address $addr$ to the expected value $expval$; if the two values are equal, then the new value $newval$ is written to $addr$, and the operation returns `true` to indicate success; otherwise, the new value is not written, and the operation returns `false` to indicate failure.

CAS is also supported in the JVM of Sun JDK 1.5. There are almost no `synchronized` blocks or methods in the `java.util.concurrent` package. A `Lock` class implemented using CAS, which offers higher performance, is used instead.

Non-blocking synchronization primitives talked here, *i.e.*, LL, SC and CAS, are wait-free.

3.2 A Language: SYNL

We formalize our analysis for a language SYNL (Synchronization Language). The syntax of SYNL is shown in Table 1. There is no explicit procedure call, as discussed in Section 1.

In SYNL, thread-local and procedure-local variables are together called *local variables*. Global and local variables are distinguished syntactically, as described below. An *unshared object* is an object accessed by only one thread. An *unshared variable* is a local variable or a field of an unshared object. A *shared variable* is a global variable or a field of a shared object. A simple escape analysis is used to determine when objects becomes shared.

A program consists of global variable declarations, thread-local variable declarations and procedure definitions. The values of thread-local variables persist between procedure calls.

An execution of a SYNL program consists of an arbitrary number of invocations (by the environment) of its procedures with arbitrary type-correct arguments (for brevity, we leave the type system implicit), and with an arbitrary amount of concurrency. Therefore, SYNL does not need constructs to create threads.

Expressions in SYNL include constant values, variables, field accesses, array accesses, non-blocking synchronization, `new` operation to allocate objects, and calls to primitive operations. Variables may have primitive types and reference types. A local variable may contain a reference a shared object. For example, a field access `x.fd` may access both a local variable `x` and a shared variable (*i.e.*, a field of a shared object). Primitive operations have no side effect, such as arithmetic operations, etc.

The statements include assignments, lock synchronization primitives, sequential composition, conditionals, `local` blocks, loops, `return`, `break` and `skip`. A `local` statement introduces a scoped procedure-local variable. The loop statement defines an unconditional loop: "`loop` s" is equivalent to "`while (true)` s". Any program with `while` loops can be re-written using `loop`, `if`, and `break`. All loops talked in the paper are unconditional.

As syntactic sugar, we allow non-blocking primitives to be used as statements when their return values are not needed; for example, $\mathrm{SC}(x, e)$ used as a statement is a syntactic sugar for: `local` $dummy = \mathrm{SC}(x, e)$ `in skip`.

Program	::=	`global` var^* ; `thread-local` var^* ; `procedure` $proc^*$
Procedure	::=	$pn(var^*)\ stmt^*$
Statement	::=	$loc := e$ | `synchronized`$(e)\ s$ | $s; s$ | `if` $e\ s\ s$ | `local` $x := e$ `in` s | `loop` s | `return` | `return` e | `break` | `skip`
Expr	::=	val | loc | `CAS`(loc, e, e) | `LL`(loc) | `VL`(loc) | `SC`(loc, e) | `new` C | $prim(e, ...)$
Location	::=	x | $x.fd$ | $x[e]$
$proc$	$\in$	*Procedure*
s	$\in$	*Statement*
e	$\in$	*Expr*
loc	$\in$	*Location*
pn	$\in$	*ProcedureName*
val	$\in$	*Val*
x	$\in$	*Variable*
fd	$\in$	*Field*
$prim$	$\in$	*Primitive*

Table 1: Syntax of SYNL

An *execution* is an initial state and a sequence of transitions. A program state is a tuple which consists of a global store G, a heap H, each thread's local store L and statements. Each transition corresponds to one step of evaluation of an expression or statement in a standard way. The formal descriptions are in [16]. For each transition, we consider the action performed by it. These actions capture the relevant behavior of the transition for our analysis and are described in Section 3.3. Note that all constructs in SYNL are deterministic, so the intermediate states during an execution are uniquely determined by the initial state and the sequence of transitions, and we will sometimes talk about executions as if those states were present in it.

Code blocks in a program P are *atomic* if: for all reachable states s of P, if all threads are executing outside those code blocks in s, then s is also reachable in an execution of P in which those blocks are executed atomically, *i.e.*, without interruption by other threads (note that the reverse implication trivially holds).

3.3 Commutativity and Atomicity Types

A *local action* is an access to an unshared variable or a field access performed by dereferencing a unique pointer stored in a local variable. Both of these kinds of accesses are always both-movers (define below) and are treated the same way in our analysis, so it is convenient to refer to both of them as local actions. Any static uniqueness analysis may be used to identify unique pointers. [16] presents a specialized uniqueness analysis for non-blocking algorithms that use working copies of a shared object; no other uniqueness analysis is needed for the examples in this paper. Other variable accesses are *global actions*. Acquire and release on shared locks are also global actions. Thus, there are four kinds of global actions: read, write, acquire lock and release lock. Let $R(v)$, $W(v)$, $acq(v)$ and $rel(v)$ denote these global actions, respectively, where v denotes the accessed variable or lock. LL and VL are global reads. SC and CAS are global writes to their first argument and, if their second or third argument are shared variables, also global reads of those variables. Note that arithmetic operations, *etc.*, do not affect our analysis and hence are not treated as actions.

Following [10], actions are classified according to their commutativity. An action is a *right-mover/left-mover* if, whenever it appears immediately before/after an action from a different thread, the two actions can be swapped without changing the resulting state. An action is a *both-mover* if it is both a left-mover and a right-mover. An action not known to be a left-mover or right-mover is *atomic* (since a single action is executed in a single step of execution).

THEOREM 3.1. *Local actions are both-movers.*

Proof. Accesses to unshared variables are obviously both-movers. For a field access performed by dereferencing a unique reference stored in a local variable, suppose that the field is f, and thread t executes an action a that accesses $o.f$ by dereferencing some local variable l (*i.e.*, l contains a unique reference to o). Before another thread can access $o.f$, t must transfer the unique reference in l into a global variable. This implies that a is a right-mover (because any action of another thread that occurs immediately after a cannot access $o.f$). Symmetrically, a is a left-mover because, between a and the closest preceding access to $o.f$ by

another thread, t must transfer the unique reference from a global variable into l. □

We assume that acquire and release have the same semantics as in Java (actually, Java bytecode).

THEOREM 3.2. *Lock acquires are right-movers. Lock releases are left-movers.*

Proof. See [4], from which this theorem is taken. Here is a proof sketch. For $acq(v)$, its immediate successor global action a from another thread can not be a successful $acq(v)$ or $rel(v)$, because $acq(v)$ would block, and $rel(v)$ would fail (in Java, it would throw an exception). Hence $acq(v)$ and a can be swapped without affecting the result, so lock acquire is a right-mover. For similar reasons, lock release is a left-mover. □

THEOREM 3.3. *(1) For a global read $R(v)$, if no global write $W(v)$ from other threads can happen immediately before/after $R(v)$, $R(v)$ is a left/right mover. (2) For a global write $W(v)$, if no global read $R(v)$ or write $W(v)$ from other threads can happen immediately before/after $W(v)$, $W(v)$ is a left/right mover.*

Proof sketch. The main observations are that two reads commute, and accesses to different variables commute. □

We briefly review atomicity types which were introduced by Flanagan and Qadeer [4]. An atomicity type is associated with an expression or statement. The atomicity types are: right-mover (R), left-mover (L), both-mover (B), atomic (A), and non-atomic (N, called compound in [4]). The first three mean that all actions executed by the expression or statement have the specified commutativity. Atomic has the same meaning as in Section 3.2. Non-atomic is used when none of the other atomicity types applies. Atomicity types are partially ordered such that smaller ones give stronger guarantees. The ordering is: $B \sqsubset t \sqsubset A \sqsubset N$ for $t \in \{L, R\}$. The atomicity type of an expression or statement can be computed from the atomicity types of its parts using the following operations on atomicity types. The join operation induced by this ordering is denoted by $\sqcup$. The *iterative closure* t^* of an atomicity type t denotes the atomicity of a statement that repeatedly executed a sub-statement with atomicity type t. It is defined by: $B^* = B$, $R^* = R$, $L^* = L$, $A^* = N$, $N^* = N$. The *sequential composition* $a; b$ is defined by the following table (the first argument is on the left; the second argument is on the top):

;	B	R	L	A	N
B	B	R	L	A	N
R	R	R	A	A	N
L	L	N	L	N	N
A	A	N	A	A	N
N	N	N	N	N	N

4. PURE LOOPS

For a loop (recall that all loops in SYNL are unconditional, *i.e.*, `while (true)` s), if the loop body terminates exceptionally, via a `break` or `return` statement, it is called *exceptional termination*; otherwise, it is called *normal termination*. We define pure loops based on the notion of pure statements introduced in [2] [1]. Informally, a loop is pure if its body (not the whole loop) has no side effect under normal termination. Intuitively, a pure loop body that terminates normally checks some state conditions (which are not satisfied) and has no side effects. When these conditions are satisfied, the loop body terminates exceptionally and may have side effects. Therefore, following the idea proposed in [2], to determine the atomicity of a pure loop, we may ignore its normal termination and focus on its exceptional termination.

Note that pure is not the same as side-effect free, because a pure loop may have side effects under exceptional termination.

A simple example of a pure loop appears in the following implementation of the `Down` operation on a semaphore. Iterations that end at `return` are exceptionally terminating. Iterations that end at line 4 (*i.e.*, when `tmp > 0` is false) or line 5 (*i.e.*, when SC returns `false`) are normally terminating and have no side effects.

```
1  Down(sem) {
2      loop
3       local tmp = LL(sem) in
4           if (tmp > 0)
5               if (SC(sem, tmp-1))
6                   return;
7  }
```

A loop is *pure* if all actions that can occur in a normally terminating iteration of the loop body (not the whole loop) are pure actions with respect to that loop. An action is *pure* with respect to a loop if any update performed by the action is "invisible" after the normally termination of iteration. Formally, a pure action should satisfy the following conditions: (*i*) it is a global action that does not perform an update, or (*ii*) it is a local action that either does not perform an update or performs an update to a variable v such that (*ii.a*) for all paths in the control flow graph from the end of loop body (*i.e.*, the program point at the end of the loop body from which control flows back to the beginning of the loop body) to procedure exit points, the next access to v, if any, is a write, and (*ii.b*) if v is not accessed on some such path, then v is procedure-local. Note that v can be a field of an object, *i.e.*, $p.fd$. If p is a unique reference stored in a local variable, (*ii.a*) implies that $p.fd$ is rewritten before p is assigned to a global variable (*i.e.*, p escapes). Informally, for a local action that performs an update, (*ii*) means that v is dead at the end of the loop body, and the written value is not visible outside the procedure. For LL actions, another condition is required for the action to be pure: (*iii*) for each LL(v) that can be executed under normal terminations of the loop, each SC(v,-) that can match it is also in the loop and there is a LL(v) on every path from loop entry to the SC. This ensures that, in every execution, the matching LL for that SC occurs in the same iteration as the SC. This special condition for LL is needed because LL implicitly performs an update that can affect subsequent SCs by the same thread.

To check whether each loop is pure, we construct a control flow graph (CFG), analyze it to identify actions that can occur in normally terminating iterations of the loop body, and then check whether those actions are pure with respect to the loop according to the above definition. There is a special case for SC and CAS. When a SC is used as the test condition of an `if` statement (*e.g.*, the last SC in `Deq` in Figure 1), if only the false branch of the `if` statement can

[1] In our framework, unlike [2], purity is a property (of loops) that has no effect on the operational semantics.

be executed under normal termination of the loop body, the SC is treated as a read (not an update). CAS is handled similarly.

Deleting a transition from an execution means removing it and adjusting the subsequent states. Details are described in [16].

THEOREM 4.1. *Let σ be an execution of a program P. Let σ' be an execution obtained from σ by deleting all transitions in a normally terminating iteration of a pure loop in a procedure p_0. Then σ' is also an execution of P, and σ and σ' contain the same states in which all threads are executing outside p_0.*

Proof sketch. When the body of a loop terminates normally, the thread begins another iteration of the same loop body (recall that loops in SYNL are equivalent to `while (true)` s). According to the definition of pure loop, under normal termination its body performs no live residual update in local actions, and no update in global actions, even if its execution is interleaved with actions of other threads. Note that an update performed by a local action of thread t that dereferences a unique reference o stored in a local variable is not visible to other threads, because (1) the update is not live at the end of the iteration, and (2) during the iteration, o is accessible only to t, since the iteration does not contain global updates and hence cannot make o accessible to other threads. The syntax of SYNL ensures that acquire and release actions occur in matching pairs in an execution of a loop body, so deleting them does not affect the resulting state or operations on the lock by other threads that could have occurred while this thread held the lock. A more detailed proof appears in [16]. □

5. CHECKING ATOMICITY

The main issue in applying Theorem 3.3 is determining whether a global action can happen immediately before or after another global action. Our analysis determines this based on how synchronization primitives are used.

5.1 Lock Synchronization

Lock synchronization is well studied. We sketch a simple treatment of lock synchronization, to illustrate how analysis of locks fits into our overall analysis algorithm.

THEOREM 5.1. *If expressions e_1 and e_2 both appear in the bodies of different* `synchronized` *statements that synchronize on the same lock, then e_1 cannot be executed immediately before or after e_2.*

Proof sketch. At least one acquire and release must occur between e_1 and e_2. □

Alias analysis may be used to determine whether two `synchronized` statements synchronize on the same lock when actions in them may access the same variable.

5.2 Non-Blocking Synchronization

Based on Theorem 4.1, for pure loops, it suffices to analyze atomicity of the loop body under exceptional termination. For each `break` or `return` statement in a loop, the backward slice of the loop body starting at that `break` or `return` and ending at the loop's entry point is called an *exceptional slice* of the loop.

To increase the precision of the analysis, we split each procedure into *exceptional variants*. Each exceptional variant is a specialized version of the procedure, and corresponds to a selection of exceptional slices of its pure loops, with each pure loop replaced by its selected exceptional slice. An example appears in Section 6.1. If the selected exceptional slice includes only the true branch of an "`if` e S_1 S_2" statement, then we replace the `if` statement with "`TRUE`(e); S_1" in the corresponding exceptional variants of the procedure; if the slice includes only the false branch, we replace the `if` statement with "`TRUE`$(!e)$; S_2". A SC action in `TRUE(SC(`v, val`))` must be successful. Note that non-pure loops appear unchanged in the exceptional variants.

THEOREM 5.2. *If all exceptional variants of a procedure p are atomic, then p is atomic.*

Proof Sketch. Let P denote the original program which contains p. Let P' denote the program obtained by replacing procedure p with its exceptional variants. Let σ be an execution of P. Let φ be a state in σ in which all threads are executing outside p.

According to Theorem 4.1, an execution σ' of P can be obtained from σ by deleting all transitions in a normally terminating iteration of a pure loop in procedure p, and φ is reachable in σ'. By the definition of exceptional variant, σ' is also an execution of P'.

By hypothesis, all exceptional variants of p are atomic. By the definition of atomicity, there exists an execution σ'' of P' in which all of the exceptional variants of p are executed atomically and in which φ is reachable.

By the definition of exceptional variants of a procedure, every invocation of an exceptional variant of p is also an invocation of p. Therefore, σ'' is also an execution of P, and all invocations of p in σ'' are executed atomically, and φ is reachable in σ''. Thus, by the definition of atomicity, p is atomic. □

In an execution, there is a unique matching LL action for each successful SC action in an execution. In program code, there might be multiple LL expressions or statements that can produce the matching LL action for an occurrence of SC. We call these the matching LL expressions of the SC expression. For example, if there is an `if` statement before a SC, and both branches of the `if` statement contain LL, either of the LL expressions can possibly match the SC.

For a SC(v, val) in a program, to find its matching LL expressions, we do a backward DFS on the control flow graph starting from the SC, and not going past edges labeled with LL(v). All of the visited occurrences of LL(v) match the SC. For a VL(v), its matching LLs can be found in the same way.

We implicitly assume hereafter that each SC has a unique matching LL expression. This assumption is not essential, but it simplifies the analysis and is satisfied by the non-blocking algorithms we have seen. We also implicitly assume that a variable updated by a SC is updated only by SC, not by regular assignment or CAS.

THEOREM 5.3. *For a successful SC(v, val) or VL(v) expression and its matching LL(v), a successful SC expression on v executed by another thread cannot be executed between them, so the successful SC or VL is a left-mover, and the matching LL is a right-mover.*

Proof. This follows from the semantics of LL, SC and VL. □

THEOREM 5.4. *For a successful SC(v, val) expression and its matching LL(v), and for a successful SC(v, val′) expression executed by another thread t′, the SC by t′, its matching LL, and all transitions of t′ between them cannot be executed between the SC(v, val) and its matching LL(v).*

Proof sketch. Let SC' and LL' denote the SC and LL, respectively, executed by t'. According to Theorem 5.3, SC' cannot happen between $LL(v)$ and $SC(v)$. Thus, there are two cases: SC' happens before $LL(v)$, or SC' happens after $SC(v, val)$. For the first case, the theorem obviously holds. For the second case, if LL' happens between $LL(v)$ and $SC(v, val)$, $SC(v, val)$ will happen between LL' and SC', which is impossible according to Theorem 5.3. If LL' happens after $SC(v, val)$, the theorem obviously holds. □

CAS is often used in a similar way as LL/SC. CAS requires an expected value as a parameter. There is often an assignment before CAS to save the old value into a temporary variable that is used as the expected value. For a CAS, its *matching read*, if any, is the action which reads the old value and saves it as the expected value. Note that a CAS can succeed even without a matching read; a SC cannot. We use a backward search on the control flow graph to find the matching reads for a CAS expression. We implicitly assume hereafter that there is a unique matching read for each CAS.

CAS-based programs may suffer from the ABA problem: if a thread reads a value A of a shared variable v, computes a new value A', and then executes CAS(v, A, A'), the CAS may succeed when it should not, if the shared variable's value was changed from A to B and then back to A by CASs of other threads. The common solution is to associate a modification counter with each variable accessed by CAS [12]. The counter is read together with the data value, and each CAS checks whether the counter still has the previously read value. A successful CAS increments the counter. With this mechanism, variants of Theorem 5.3 and 5.4 hold for CAS: just replace "matching LL" with "matching read", and replace "SC" with "CAS".

5.3 Condition-based Non-Blocking Synchronization

A predicate $p(lvar)$ is called a *local condition* of a code block `local` $lvar = e$ `in` $stmt$ (which is called a *local block* on $lvar$), if it satisfies the following two conditions: (i) $lvar$ is not updated in $stmt$, and (ii) $p(lvar)$ holds throughout execution of $stmt$.

Condition (i) is easy to check, because there is no aliasing of local variables in SYNL. When condition (i) holds, the local condition can easily be obtained from the `TRUE` statements in $stmt$ that depend only on $lvar$. For example, in Figure 3, a local condition for the code block a5-a8 is $next == null$, and a local condition for the code block b2-b5 is $next != null$. If condition (i) does not hold, or no appropriate `TRUE` statements appear in the local block, its local condition is `true`.

A local block of the form `local` $lvar =$ `LL`($svar$) `in` {$stmt$; `TRUE`(`SC`($svar, val$));} is called a *LL-SC block* on $svar$.

THEOREM 5.5. *Suppose a shared variable svar is updated only by SC expressions in LL-SC blocks, and every LL-SC block* `local` $lvar =$ `LL`($svar$) `in` {$stmt$;`TRUE(SC`($svar, val$)`);`} *in the program has the same local condition p(lvar). Suppose a local block* `local` $lvar' = svar$ `in` $stmt'$ *has a local condition !p(lvar′). No transition in that local block can be executed inside any LL-SC block on svar, and no transition in any LL-SC block on svar can be executed inside that local block.*

Proof sketch. The conclusion follows from the fact that $svar$ is not updated by other threads during execution of the LL-SC block (because that would cause its SC to fail), so $p(svar)$ (note $svar$ is equal to $lvar$) holds during execution of the LL-SC block, and the fact that $svar$ is not updated during execution of $stmt'$, because $!p(svar)$ would still hold when the first successful SC interleaved in $stmt'$ happens, contradicting the SC's local condition, so $!p(svar)$ holds throughout execution of $stmt'$. For details, see Appendix A. □

The definition of LL-SC block and the above theorem can be generalized, so that the LL does not need to occur at the beginning of a local block, and the SC does not need to occur at the end of a local block. A similar theorem exists for CAS.

5.4 Atomicity Inference

To analyze atomicity of each procedure in a SYNL program, we identify pure loops, then replace each procedure with its exceptional variants. We compute atomicity types for all expressions and statements in the resulting program as follows:

- Step 1: Identify all local actions and lock actions. According to Theorem 3.1, all local actions have atomicity type B. According to Theorem 3.2, all lock acquires and releases have atomicity type R and L, respectively. A simple escape analysis is used to identify accesses to objects that have not escaped from the creating threads; those accesses are like accesses to unshared variables and have atomicity type B.
- Step 2: According to Theorem 5.3, if all updates on a variable v are done through SC, all successful SC(v, val) and VL(v) have atomicity type L, and their matching LL(v) have atomicity type R. The analogous theorem for CAS is used for successful CAS and their matching reads.
- Step 3: Infer local conditions for local blocks, as described in Section 5.3.
- Step 4: Using Theorems 5.1, 5.3, 5.4 and 5.5, for each read, check whether there is a write on the same variable that can happen immediately before/after it; for each write, check whether there is a read or write on the same variable can happen immediately before/after it. For access to variables on the heap, the analysis does a case split on whether two field accesses refer to the same location; we consider both cases, unless alias analysis shows one is impossible. Our current alias analysis just checks whether the references have the same type and whether the same field is being accessed. Assign atomicity types to the reads and writes based on Theorem 3.3. If some reads and writes were given atomicity types in previous steps, use the minimum of the atomicities based on the partial order discussed in Section 3.3.

- Step 5: For actions not given an atomicity type in previous steps, conservatively assign them atomicity type A.

- Step 6: Propagate atomicity types from the actions up through the abstract syntax trees of the procedures using the atomicity calculus in [4]. The atomicity type of a compound program construct is computed from the atomicity types of its parts using join, sequential composition, and iterative closure as appropriate.

- Step 7: For each procedure p in the original program, if every exceptional variant of p has a procedure body with atomicity type A, then by Theorem 5.2, p has atomicity type A.

6. APPLICATIONS

This section demonstrates the applicability of our analysis to four non-trivial non-blocking algorithms from the literature. Although in two cases we must modify the algorithm before applying our analysis, we consider the results encouraging, since we do not know of any other algorithmic (*i.e.*, automatic) analysis that can show atomicity of the same (or larger) code blocks in the modified or original versions.

6.1 Michael and Scott's Non-Blocking FIFO Queue Using LL/SC/VL

Figure 1 contains code for a non-blocking FIFO queue (NFQ) that uses LL/SC/VL [11]. It is similar to the well-known CAS-based algorithm in [13]. It uses a singly-linked list whose head and tail are pointed to by global variables `Head` and `Tail`. Enqueue consists of three main steps: create a node, add it to the end of the list, and update `Tail`. A blocking implementation would use a lock around the second and third steps to achieve atomicity. In the non-blocking algorithm, if a thread gets delayed (or killed) after the second step, other threads may update `Tail` on its behalf; in that case, if the delayed thread later tries to update `Tail`, its SC will harmlessly fail. To avoid blocking, the dequeue operation also updates `Tail`. A dummy node is used as the head of the queue to avoid degenerate cases. The code for `Deq` in [11, 13] stores the value of `LL(Tail)` in a local variable; the code in Figure 1 does not. This does not affect the correctness or performance of the algorithm but makes it easier to analyze.

We would like to show that NFQ is linearizable, using the two-step approach described in Section 1. An obstacle is that the loops in `Enq` and `Deq` are not pure, because of the updates to `Tail`. Therefore, we modify the program to make the loops pure before applying our analysis algorithm; specifically, we consider the modified program NFQ′ in Figure 2 and argue that the modification preserves linearizability (the proof is relatively easy). In NFQ′, all updates to `Tail` are performed in a separate procedure `UpdateTail`. `UpdateTail` may be invoked (by the environment) at any time, so NFQ′ is effectively more non-deterministic than NFQ.

In other words, NFQ′ can simulate all behaviors of NFQ. It is not difficult to show that linearizability of NFQ with respect to any specification (of the kind defined in [8]) follows from linearizability of NFQ′ with respect to that specification augmented freely with calls to `UpdateTail`. A more precise statement of this theorem and a proof appear in [16]. Our analysis algorithm can show that the procedures in NFQ′ are atomic. To conclude NFQ′, and hence NFQ, are linearizable with respect to a sequential specification of FIFO queues, we also need to show that NFQ′ executed sequentially satisfies that specification. One approach is to use a powerful verification tool such as TVLA [18] which is a model checker based on static analysis. With our approach, TVLA only needs to consider sequential executions of NFQ′, so the verification will be much faster and use much less memory than the verification in [18], where TVLA was used to show directly that NFQ satisfies some complicated temporal logic formulas.

```
void Enq(int value)
 node = new Node();
 node.value = value;
 node.next = null;
 loop
   local t = LL(Tail) in
   local next = LL(t.Next) in
     if (!VL(Tail)) continue;
     if (next != null)
       SC(Tail,next);
       continue;
     if (SC(t.Next,node))
       // optional
       [SC(Tail,node);]
       return;
```

```
int Deq()
  loop
    local h = LL(Head) in
    local next = h.Next in
      if !VL(Head) continue;
      if (next == null)
        return EMPTY;
      if (h == LL(Tail))
        SC(Tail,next);
        continue;
      local value = next.Value in
        if (SC(Head,next))
          return value;
```

Figure 1: Non-Blocking FIFO Queue (NFQ). Names of global variables start with an uppercase letter. The declarations of global variables `Tail` and `Head` are not shown.

To evaluate the speedup that our atomicity analysis can provide for subsequent verification, we used TVLA to verify several correctness properties of NFQ′, similar to the properties in [18, Table 2]. We analyzed the correct program with two different environments: in the first one, the number of threads that concurrently call `AddNode` is unbounded (there is only one thread performs dequeues, and there is only one `UpdateTail` thread, since it contains a non-terminating loop); in the second one, the number of threads that concurrently perform dequeues is unbounded (the threads that perform `AddNode` and `UpdateTail` are single). We also checked the properties for an incorrect version of NFQ′; specifically, we deleted the statement `if (next.ref != null) continue` in the `AddNode` procedure; TVLA catches this error. We performed all experiments twice: once with each procedure body declared as atomic, as inferred by our analysis algorithm, and once without those declarations. The atomicity declarations had little effect on the time needed for TVLA to find an error in the incorrect program, but it reduced the time and space needed to verify the correct versions by a factor of 100 or more. The experimental results appear in Table 2.

All exceptional variants for the procedures of NFQ′ are listed in Figure 3. The left side column shows line numbers and the atomicity type of the code on each line. A line may contain multiple actions; we refer to the sequential composition of their atomicity types as the atomicity type of the line. We show how the atomicity analysis algorithm in Section 5.4 works on these procedures.

In step 1, a1, a2, a3, a7, a9, b4, b6, c4, c5, d4 and d8 are classified as both-movers because they access local variables.

program	without atomic		with atomic	
	states	time	states	time
unbounded `AddNode` threads	4500	> 19hrs	13	3.0s
unbounded `Deq'` threads	1285	88 min	10	1.7s
incorrect `AddNode`	13	5 sec	13	3.0s

Table 2: Experimental results for verification of NFQ′ with TVLA.

```
void AddNode(int value)
 local node = new Node() in
 node.Value = value;
 node.Next = null;
 loop
   local t = LL(Tail) in
   local next = LL(t.Next) in
     if !VL(Tail)
       continue;
     if (next != null)
       continue;
     if SC(t.Next,node)
       return;

 int Deq'()
   loop
     local h = LL(Head) in
     local next = h.Next in
       if (!VL(Head))
         continue;
       if (next == null)
         return EMPTY;
       if (h == LL(Tail))
         continue;
       local value = next.Value in
         if (SC(Head,next))
           return value;
```

```
void UpdateTail()
  loop
    local t = LL(Tail) in
    local next = t.Next in
      if !VL(Tail)
        continue;
      if (next != NULL)
        SC(Tail,next);
        return;
```

Figure 2: NFQ′, a modified version of NFQ

In step 2, a4, a5, b1, c1 are d1 are classified as right-movers, because they are matching LLs for successful SCs or VLs; a6 (which is reclassified as a both-mover in step 4), a8, b5, c3, and d7 are classified as left-movers because they are successful SCs or VLs; b3 and d3 are classified as right-movers, and then reclassified in step 4 as both-movers because they are between matching LLs and successful SCs.

In step 3, the local condition for a5-a8 and c2-c4 is $next == null$. The local condition for b2-b5 and d2-d8 is $next! = null$.

Now consider step 4. Let $\mathtt{t}_a$ and $\mathtt{t}_u$ denote the local variable `t` in `AddNode` and `UpdateTail`, respectively. If $\mathtt{t}_a$`.Next` of the LL-SC block in `AddNode` is aliased with $\mathtt{t}_u$`.Next` of the local block in `UpdateTail`, then according to Theorem 5.5, the update on `Tail` (*i.e.*, b5) cannot happen between a6 and a7, so a6 is a both-mover. a8 cannot happen between b2 and b3, so b2 is a right-mover. Suppose $\mathtt{t}_a$`.Next` is not aliased with $\mathtt{t}_u$`.Next`; this implies $\mathtt{t}_a$ is not aliased with $\mathtt{t}_u$, *i.e.*, $\mathtt{t}_a \neq \mathtt{t}_u$, so even if a8 happens between b2 and b3, b2 is a right-mover by Theorem 3.3. $\mathtt{t}_a \neq \mathtt{t}_u$ implies that the `Tail` in `AddNode` is not equal to `Tail` in `UpdateTail`. Thus, even if b5 happens between a6 and a7, a6 is still a both mover by Theorem 3.3. For d2, if `h.Next` is aliased with `t.Next` of

```
void AddNode(int value)
a1:B  local node = new Node() in
a2:B    node.Value = value;
a3:B    node.Next = null;
a4:R    local t = LL(Tail) in
a5:R      local next = LL(t.Next) in
a6:B        TRUE(VL(Tail));
a7:B        TRUE(next == null);
a8:L        TRUE(SC(t.Next,node));
a9:B    return;

void UpdateTail()
b1:R  local t = LL(Tail) in
b2:R    local next = t.Next in
b3:B      TRUE(VL(Tail));
b4:B      TRUE(next != NULL);
b5:L      TRUE(SC(Tail,next));
b6:B  return;

int Deq'1()
c1:R  local h = LL(Head) in
c2:A    local next = h.Next in
c3:L      TRUE(VL(Head));
c4:B      TRUE(next == null);
c5:B  return EMPTY;

int Deq'2()
d1:R  local h = LL(Head) in
d2:R    local next = h.Next in
d3:B      TRUE(VL(Head));
d4:B      TRUE(next != null);
d5:A      TRUE(h != LL(Tail));
d6:B      local value = next.Value in
d7:L        TRUE(SC(Head,next));
d8:B        return value;
```

Figure 3: Exceptional variants for procedures of NFQ′.

`AddNode`, a8 cannot happen between d2 and d3 according to Theorem 5.5, hence d2 is a right-mover by Theorem 3.3; if `h.Next` is not aliased with `t.Next`, d2 is again a right-mover. Also in step 4, d6 is inferred to be a both-mover, because there is no write on the `Value` field of any shared object; the only write a2 on `Value` is on an object that has not escaped.

In step 5, the unclassified c2 and d5 are given atomicity type A. Step 6 infers that each procedure in Figure 3 has atomicity type A. Step 7 infers that all procedures in NFQ′ are atomic.

6.2 Herlihy's Non-Blocking Algorithm for Small Objects

Figure 4 shows Herlihy's algorithm for non-blocking concurrent implementation of small objects [7]. Suppose a small object (*i.e.*, small enough to be copied efficiently) is shared by a set of threads. The main steps on each thread in the algorithm are: (1) read the shared object reference using LL; (2) copy the data from the shared object into a private (*i.e.*, currently unshared) working copy of the object; (3) perform the requested computation on the private object; (4) switch the references between the shared object and the private object using SC and an assignment statement. Note that, the formerly shared object becomes a private copy, and the formerly private object becomes the current shared copy.

Before a thread t_1 switches the reference of the shared copy o with the private copy of t_1, another thread t_2 may read the reference to o using `LL(Q)`. Even though o becomes

```
void proc(Node Q) loop
  local m = LL(Q) in          a1:R local m = LL(Q) in
    copy(prv.data,m.data);    a2:B   copy(prv.data,m.data);
    if (!VL(Q)) continue;     a3:B   TRUE(VL(Q));
    computation(prv.data);    a4:B   computation(prv.data);
    if (SC(Q,prv))            a5:L   TRUE(SC(Q,prv))
      prv = m;                a6:B   prv = m;
      break;                  a7:B   break;
```

Figure 4: Herlihy's non-blocking algorithm for small objects. Left column: original code. Right column: the exceptional variant. `prv` is a thread-local variable, `Q` is a shared variable.

the private copy of t_1, t_2 may still hold the reference to o, though the SC of t_2 will fail later, causing t_2 to loop and read the current reference from `Q`. Thus, t_1 may write to o while t_2 copies data from o. If t_2 tried to perform a computation on a copy of the data that reflects only part of some update, it might suffer a fatal error, such as divide by zero. Line a3 prevents this: if *o.data* (accessed as `m.data`) is modified by another thread during the copy in line a2, the VL will fail.

Applying the uniqueness analysis in [16] to this algorithm shows that accesses to fields of `prvObj` are local actions, because `prvObj` effectively contains a unique reference. It is easy to check that the update to `prv.data` is a pure action with respect with the loop, and hence the loop is pure. The procedure has only one exceptional variant shown in Figure 4. According to Theorem 5.4, the LL-SC block a1-a5 is atomic. Both variables in a6 are local, so a6 is a both-mover. Combining these atomicities shows that the exceptional variant is atomic, and hence the original procedure is atomic.

6.3 Gao and Hesselink's Non-Blocking Algorithm for Large Objects

For large objects, copying is the major performance bottleneck. Gao and Hesselink [5] proposed an algorithm to avoid copying the whole object. The fields of each object are divided into disjoint groups such that each operation changes only fields in one group. When copying data between the shared and private copies of an object, only the modified groups are copied. To efficiently detect modifications, a version number is associated with each group of fields of each copy of the object. The algorithm works as follows: (1) read the shared object reference using LL; (2) copy data and version numbers in all modified groups of fields of the currently shared copy of the object into the corresponding groups of fields of the current thread's private copy; (3) do the computation on the private copy, updating fields in some group and incrementing the corresponding version number; (4) switch the references between the shared object and the private object using SC. The algorithm is more complicated than the algorithm for small objects in Section 6.2 mainly because of the loops over groups of fields, the conditional behavior depending on which groups of fields changed, and the use of version numbers to efficiently detect changes.

Our analysis cannot directly show that the algorithm is atomic, due to the use of version numbers. Our analysis algorithm is able to show that a version of the algorithm does not use version numbers is atomic. We then show that the transformations that optimize the algorithm by introducing and using version numbers preserve atomicity; this is relatively easy.

```
void proc(Object SharedObj, int g)
a1     loop
a2         local m = LL(SharedObj) in
a3         local i = 1 in
a4         loop
a5             if (i>W) break;
a6             copy(prvObj.data[i],m.data[i]);
a7             if (!VL(SharedObj))
a8                continue a2;
a9             i++;
a10        if (!VL(SharedObj)) continue a2;
a11        compute(prvObj,g);
a12        if (SC(SharedObj,prvObj))
a13            prvObj = m;
a14            return;
```

Figure 5: Gao and Hesselink's Non-Blocking Algorithm for Large Objects: Simplified Program 1

We show that the non-blocking algorithm for large objects in Figure 7 is atomic. We use `continue` in the pseudo-code, even though SYNL does not have `continue`. It is easy to eliminate the `continue`, with no significant effect on the atomicity analysis. The procedure call `copy(prvObj.data[i], m.data[i])` copies the data in `m.data[i]` to `prvObj.data[i]`. The procedure `compute(prvObj,g)` does computation based on the data in `prvObj` and writes the result into `prvObj.data[g]`.

The algorithm in Figure 7 differs in some minor ways from the original algorithm in [5]. The program in Figure 7 simplifies the algorithm in [5] by removing redundant array `old`. Our version, like the program in Section 6.2, uses VL (line a13) to prevent errors due to inconsistent states of `prvObj` that may result from updates during copying (line a8). [5] simply assumed that such errors do not occur. Also, we omit the guard predicate used in [5] to optimize cases where compute is applied in a state in which it performs no updates.

Figure 5 shows a simplified version of the algorithm in which all data of the shared object (*i.e.*, `m`) are copied into the working object (*i.e.*, prvObj) of the current thread in every iteration of the outer loop. Applying the uniqueness analysis in [16] shows that all field accesses through `prvObj` are local actions, because `prvObj` effectively contains a unique reference. Moreover, `prvObj.data[i]` is dead at the end of the outer loop's body under all normal terminations. Therefore, the outer loop is pure. By the same reasoning as for the non-blocking algorithm for small objects in Section 6.2, the procedure in Figure 5 is atomic.

Figure 6 shows an improved version of the code in Figure 5 in which the copy is omitted from `m.data[i]` to `prvObj.data[i]` when those two locations already contain the same value. The program in Figure 6 clearly has the same behavior as the program in Figure 5. Therefore, the procedure in Figure 6 is atomic.

Figure 7 shows an improved version of the code in Figure 6 in which version numbers are used to efficiently and conservatively check whether `m.data[i]` and `prvObj.data[i]` are equal. "Conservatively" here means that the check might return false when they contain the same value (*e.g.*, because the values stored in `m.data[i]` and `prvObj.data[i]` happen to be equal), but this merely causes the code in Figure 7 to do an unnecessary copy (*i.e.*, the copy does not actually change the value of `prvObj.data[i]`). The last state-

```
void proc(Object SharedObj, int g)
a1     loop
a2         local m = LL(SharedObj) in
a3         local i = 1 in
a4         loop
a5             if (i>W) break;
a6             if (prvObj.data[i] != m.data[i])
a7                 copy(prvObj.data[i],m.data[i]);
a8                 if (!VL(SharedObj))
a9                    continue a2;
a10            i++;
a11        if (!VL(SharedObj)) continue a2;
a12        compute(prvObj,g);
a13        if (SC(SharedObj,prvObj))
a14            prvObj = m;
a15            return;
a16        //else continue a2;
```

Figure 6: Gao and Hesselink's Non-Blocking Algorithm for Large Objects: Simplified Program 2

```
void proc(Object SharedObj, int g)
a1     loop
a2         local m = LL(SharedObj) in
a3         local i = 1 in
a4         loop
a5             if (i>W) break;
a6             local newVersion[i] = m.version[i] in
a7                 if (newVersion[i] != prvObj.version[i])
a8                     copy(prvObj.data[i],m.data[i]);
a9                     if (!VL(SharedObj))
a10                        continue a2;
a11                    prvObj.version[i] = newVersion[i];
a12            i++;
a13        if (!VL(SharedObj)) continue a2;
a14        compute(prvObj,g);
a15        prvObj.version[g]++;
a16        if (SC(SharedObj,prvObj))
a17            prvObj = m;
a18            return;
a19        else
a20            prvObj.version[g] = 0;
```

Figure 7: Gao and Hesselink's Non-Blocking Algorithm for Large Objects: Full Program with Modification

ment `prvObj.version[g] = 0` is needed so that the update to `prvObj.version[g]` from line a15 will be discarded if the SC fails. The program in Figure 7 clearly has the same behaviors as the program in Figure 6. Therefore, the procedure in Figure 7 is atomic.

To evaluate the benefit of our atomicity analysis compared to a traditional partial-order reduction, we implemented the algorithm for large objects in the model checker SPIN [9]. We wrote a driver with 3 threads that concurrently invoke arithmetic operations on a shared object with 3 integer fields, each in its own group. The input files are available at the URL in [16]. The numbers of reachable states are: 4,069,080 with no optimization; 452,043 with SPIN's built-in partial-order reduction; 69,215 with the procedure body declared as atomic, as inferred by our analysis algorithm; and 4619 with both optimizations.

6.4 Michael's Lock-Free Memory Allocator

Michael designed an efficient and robust lock-free (*i.e.*, non-blocking) memory allocator that uses CAS [12]. We applied our analysis algorithm to the pseudo-code for `malloc` in Figure 4 of [12]. We assume that the auxiliary procedures for which code is not given in [12] are atomic. We inline all calls to other procedures. All the loops are pure, and all their exceptional slices are atomic. Moreover, all CAS-blocks (from each successful CAS back to the matching read) are atomic, because all actions in the CAS-blocks are either accesses to local variables, or accesses to read-only shared variables. Some CAS actions do not have matching reads; each one by itself is an atomic code block. The remaining actions are local actions; they can be combined with the previous or following atomic block. More details appear in [16]. In summary, the allocation routines contain 74 lines of pseudo-code (actual C code may be significantly longer), and our analysis classifies it into 15 atomic blocks.

7. CONCLUSIONS

This paper presents a static analysis to infer atomicity of code blocks in programs with non-blocking synchronization. Although we need to modify the program before applying our analysis in two out of the four examples in Section 6, we consider the results encouraging, since we do not know of any other algorithmic (*i.e.*, automatable) analysis that can show atomicity of the same (or larger) code blocks in the modified or original versions. Theorem-proving approaches, such as [5], can verify atomicity of the original programs but require much more manual effort than our approach, even if our analysis algorithm is applied manually. In programs where entire procedures are not atomic, such as the memory allocator example in Section 6.4, our analysis shows that many code blocks are atomic; this can significantly reduce the number of states considered during subsequent analysis and verification.

Acknowledgment

We thank Cormac Flanagan for helpful comments and Eran Yahav for help with TVLA.

8. REFERENCES

[1] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proc. of ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2004.

[2] C. Flanagan, S. N. Freund, and S. Qadeer. Exploiting purity for atomicity. In *Proc. ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 221–231. ACM Press, 2004. An extended version appeared as Technical Report 04-02, Williams College, 2004.

[3] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2003.

[4] C. Flanagan and S. Qadeer. Types for atomicity. In *Proc. ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*, pages 1–12. ACM Press, 2003.

[5] H. Gao and W. H. Hesselink. A formal reduction for lock-free parallel algorithms. In *Proceedings of the*

16th International Conference on Computer-Aided Verification (CAV), Lecture Notes in Computer Science, pages 44–56, 2004.

[6] M. P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[7] M. P. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, Nov. 1993.

[8] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[9] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[10] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[11] M. M. Michael. Private communication, 2004.

[12] M. M. Michael. Scalable lock-free dynamic memory allocation. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, June 2004.

[13] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 267–275. ACM Press, 1996.

[14] D. Peled. Ten years of partial order reduction. In A. J. Hu and M. Y. Vardi, editors, *Proc. 10th Int'l. Conference on Computer-Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 17–28. Springer-Verlag, 1998.

[15] S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *Proc. 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–255. ACM Press, 2004.

[16] L. Wang and S. D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. Technical Report DAR-04-17, SUNY at Stony Brook, Computer Science Dept., Oct. 2004 (revised Jan. 2005). Available at http://www.cs.sunysb.edu/~liqiang/nonblocking.html.

[17] L. Wang and S. D. Stoller. Run-time analysis for atomicity. In *Third Workshop on Runtime Verification (RV03)*, volume 89(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

[18] E. Yahav and M. Sagiv. Automatically verifying concurrent queue algorithms. In *Proc. Workshop on Software Model Checking (SoftMC'03)*, volume 89(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

APPENDIX

A. PROOF OF THEOREM 5.5

LEMMA A.1. *Suppose a shared variable svar is updated only by SC expressions in LL-SC blocks, and every LL-SC block* `local` $lvar$ = `LL(`$svar$`) in {`$stmt$`;TRUE( SC(`$svar, val$`));}` *in the program has the same local condition* $p(lvar)$. *Suppose a local block* S `local` $lvar' = svar$ `in` $stmt'$ *has a local condition* $!p(lvar')$. *Any successful SC(svar) in the LL-SC blocks cannot happen inside* S.

Proof. We prove the lemma by showing a contradiction. Suppose a successful SC($svar$) in a LL-SC block executed by another thread happens inside S. Without loss of generality, we consider the first such SC($svar$). According to the assumption, $!p(lvar)$ holds during $stmt'$. Because $svar$ is updated only by SC actions from LL-SC blocks, $lvar' == svar$ and hence $!p(svar)$ holds from the start of $stmt'$ until SC($svar$) happens. This implies that $!p(svar)$ holds when SC($svar$) happens. The LL-SC block has local condition $p(lvar)$, and $lvar == svar$ holds until the SC, because $lvar$ is not updated in the LL-SC block, and $svar$ is not updated before the first successful SC on it, so $p(svar)$ holds when SC($svar$) happens. This contradicts the previous conclusion. This concludes the proof of the lemma. □

Proof of Theorem 5.5. According to Lemma A.1, no successful SC($svar$) can happen inside S. Consider an execution of a LL-SC block on $svar$. There are two cases:

case 1: the successful SC happens before S. Thus, the whole LL-SC block happens before S. Obviously, the theorem holds in this case.

case 2: the successful SC happens after S. If the matching LL also happens after S, the whole LL-SC block happens after S. Hence the theorem holds. Suppose the matching LL happens inside S or before S. Similar to the proof of Lemma A.1, because $svar$ is updated only by SC actions from LL-SC blocks, and no successful SCs happen inside S or between the matching LL and the SC, $lvar' == svar$ and hence $!p(svar)$ holds from the start of $stmt'$ until the SC($svar$) happens. Thus, $!p(svar)$ holds when SC($svar$) happens. By the same reason of the proof of Lemma A.1, $p(svar)$ holds when SC($svar$) happens. This contradicts the previous conclusion. Therefore, when SC happens after S, the matching LL cannot happen inside S or before S. □