Composable, Nestable, Pessimistic Atomic Statements

Zachary Anderson
ETH Zürich
zachary.anderson@inf.ethz.ch

David Gay
Intel Labs Berkeley
dgay@acm.org

Abstract

In this paper we introduce a new method for pessimistically implementing composable, nestable atomic statements. Our mechanism, called *shelters*, is inspired by the synchronization strategy used in the Jade programming language. Unlike previous lock-based pessimistic approaches, our mechanism does not *require* a whole-program analysis that computes a global lock order. Further, this mechanism frees us to implement several optimizations, impossible with automatically inserted locks, that are necessary for scaling on recent multicore systems. Additionally we show how our basic mechanism can be extended to support both open- and closed-nesting of atomic statements, something that, to our knowledge, has not yet been implemented fully-pessimistically in this context. Unlike optimistic, transactional-memory-based approaches, programmers using our mechanism do not have to write compensating actions for open-nesting, or worry about the possibly awkward semantics and performance impact of aborted transactions.

Similar to systems using locks, our implementation requires programmers to annotate the types of objects with the shelters that protect them, and indicate the sections of code to be executed atomically with atomic statements. A static analysis then determines from which shelters protection is needed for the atomic statements to run atomically. We have implemented shelter-based atomic statements for C, and applied our implementation to 12 benchmarks totaling over 200k lines of code including the STAMP benchmark suite, and the sqlite database system. Our implementation's performance is competitive with explicit locking, Autolocker, and a mature software transactional memory implementation.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages

Keywords Atomic statements, Locks, Concurrency

1. Introduction

Given recent advances in hardware, writing multithreaded programs that manipulate shared state is an increasingly important task. However, it is also very challenging, even for experienced programmers. Though explicit locking can yield highly efficient code, its use is prone to errors such as data-races and deadlocks. Indeed, as of June 2011, according to their bugzilla databases, there were 38 known, outstanding race conditions in the Linux kernel, and 136 in Firefox; and there were 39 known, outstanding deadlocks in the Linux kernel, and 107 in Firefox [22, 27].

Atomic statements are a convenient language construct for controlling access to shared state in multithreaded programs. They ensure that the statements within them execute atomically. That is, the effects of operations in atomic statements become visible to other threads only all at once when execution leaves the atomic statement, much like a database transaction. Because they simply declare what to run atomically, rather than fully specifying how to ensure atomicity, user studies have shown that the use of atomic statements is less error prone than explicit locking [34]. In particular, atomic statements make it much easier to compose independently written code as there is no confusion created by having different locking disciplines in different modules.

In this paper we introduce an implementation of atomic statements suitable for use in systems-level C code. Our mechanism, called *shelters*, is inspired by the synchronization strategy used in the Jade programming language [33]. Our approach is pessimistic, able to support multiple semantics for nested atomic statements, allows atomic statements to coexist with explicit locks, and does not require whole-program analysis. These benefits come at the cost of only a small number of simple annotations on types and functions. Further, our implementation achieves performance competitive with explicit locking on benchmarks and a few applications, including the sqlite database system.

Existing optimistic and pessimistic systems have significant drawbacks. STM systems execute atomic statements optimistically, at least in part. Atomic sections are allowed to execute concurrently, but when two or more threads make conflicting accesses, transactions must be aborted, rolled-back, and retried. Some TM implementations achieve good performance. However, if transactions are large, or if data is highly contended, conflicts may be frequent and expensive. Furthermore, roll-back may not be possible if, for example, a thread performed I/O during a failed transaction. STM systems support multiple semantics for nested transactions: closed-nesting, in which the effects of nested transactions become visible only at the end of the outer transaction; and open-nesting, in which the effects of nested transactions become visible at their respective ends. However, in order to use open-nesting, a programmer may have to write compensating actions, and reason directly about the semantics of roll-back [26]. We view this requirement as overly-burdensome.

These considerations lead us to prefer a pessimistic approach. However, previous pessimistic approaches have relied on whole-program analysis for determining a global lock order [25], or for inferring fine-grained lock hierarchies [13, 19]. But whole-program analysis is often problematic in practice. First, it is often expensive for large programs. Second, the source code for the whole program may not always be available. Furthermore, previous pessimistic, lock-based approaches have only supported closed-nesting semantics.

Our approach avoids these problems, thereby making it suitable for systems-level C programs. In Sections 2 and 3 we give a detailed description of our design, which we formalize and prove correct in Section 4. Further, we have implemented compiler transformations and a runtime for the C language, described in Section 5, and applied our implementation to 12 programs including the STAMP benchmark suite [11], and a few representative applications, including the sqlite database system, totaling over 200k lines of code. In Section 6, we present a thorough comparison of the runtime and programming cost of shelter-based atomic statements with the Intel STM implementation [21] and Autolocker [25] for the above mentioned benchmarks.

In summary, we make the following contributions:

- We present the design and implementation of *shelters*, a pessimistic method for implementing atomic statements, including straightforward semantics for both open- and closed-nesting, and interaction with explicit locking, that requires no whole-program analysis.
- We formalize our design to show that shelters enforce useful concurrency guarantees and are deadlock-free.
- We present the results or our experiments comparing the runtime performance and annotation overheads of shelter-based atomic statements with explicit locking, Autolocker, and an STM system. In summary, both our annotation burden and our performance burden are competitive with other systems. We also show that acquiring all locks upon entering an atomic statement, an obvious alternative to our system, has worse performance than shelters.

2. Overview

In this section we give an overview of what shelters are, how they work, and our extensions to C. Then in Section 3, using a few examples, we explain in detail the features of our system.

2.1 Extensions to C

We add to the C language a few basic constructs: atomic statements, shelters, a type annotation, and a function annotation.

Atomic statements in our extension have the usual semantics: the effects of a "closed" atomic statement on shared objects, including those in any nested atomic statements, do not become visible to other threads until the atomic statement is finished. We also provide "open" atomic statements, in which the effects of any nested atomic statements on shared objects become visible at their respective ends. In our system, the programmer identifies what constitutes a single shared object through a combination of *shelters* and type annotations.

Shelters, like locks, are first-class objects, and can be declared as variables, structure fields, and function arguments. Further, as in Autolocker [25], the programmer specifies the shelter that protects a location by writing a type annotation. Thus, what constitutes an object for concurrency-control purposes are those locations in memory whose C types, at the static source-code level, are annotated with a `sheltered_by(s)` qualifier using the same shelter `s`. Shelters, then, are simply a mechanism for storing all of the concurrency-control metadata in one place for a collection of locations that may not be accessed concurrently. From here on, we simply refer to such a collection of locations as a single object identified by a single shelter.

Since our approach is fully pessimistic, at the beginning of an atomic statement, we must know what objects may be touched within it. We discover these objects by performing a backwards dataflow analysis over atomic statements. For the implementation of open atomic statements, the analysis also notes in which nested atomic statement an object is accessed. Our compiler transforms a program such that at the beginning of an atomic statement, the collected shelters are passed to functions of our runtime, which are described below.

Unlike previous pessimistic approaches, one of our design gaols is to avoid whole-program analysis. Since we must know what objects may be accessed in an atomic statement we require the programmer to write function annotations describing the objects a function will access. These take the form of `needs_shelter()` annotations on function types. Our implementation performs a per-module call-graph analysis so that programmers must only annotate the module boundaries and function pointers of those functions that may be called with in an atomic statement. Further, at compile time, we provide warnings and errors as appropriate when `needs_shelters()` annotations are missing. We describe the conditions under which warnings and errors are emitted in Section 3.

The programmer is responsible for placing the `sheltered_by(s)` annotation on shared objects protected by shelter `s`. The programmer is also responsible for using the atomic statement with the correct semantics for his situation. Though our implementation provides errors if annotated objects are accessed outside of atomic statements, if type annotations are missing, then data-races, but not deadlocks, may result. Through a straightforward combination with our earlier sharing-checker work [5], we can provide compile-time and run-time errors for these missing-annotation mistakes.

Next, we give a brief description of how the semantics of our atomic statements are implemented by our runtime.

2.2 Runtime Semantics

Shelters are inspired by Jade in the sense that both are based on timestamp-based concurrency control as developed in the database community [8]. In databases with *basic* timestamp-based concurrency control, each transaction receives a timestamp when it begins. Each object in the database also has an associated timestamp that records the timestamp of the transaction that most recently accessed it. The timestamps are needed to implement optimistic concurrency control—the decision to abort a transaction (and maintain consistency) is made by comparing transaction timestamps with the object timestamps. If a transaction would read an object that was written by a newer transaction, or overwrite a value read or written by a newer transaction, then it must be aborted, rolled-back, and restarted. This approach is akin to how transactional memory systems operate.

The approach we use with shelters—and the approach used in Jade—is more similar to *strict* timestamp-based concurrency control, in which transactions that began later must block before accessing objects that may be accessed by transactions that began earlier, and must remain blocked until these other transactions finish. As one might expect, because transactions may block, additional logic is needed in the transaction manager to avoid deadlock. So, in essence, shelters solve the problem of using strict timestamp-based concurrency control for deadlock-free, systems-level programming.

In our system and Jade, the timestamp takes the form of a distinct value obtained by a thread from a strictly increasing thread-safe counter at the beginning of each atomic statement. The implementation details of this counter are discussed in Section 5.

At the beginning of an atomic statement, a thread atomically "registers" its timestamp with all of the shelters protecting the objects it will touch in the atomic statement, as found by the dataflow analysis described above. This registration operation is the function `shelter_register()` in our runtime (register() in our formalism). Atomically, in the register operation, a thread gets a timestamp and records its identifier and timestamp in each of the shelters it is registering for.

Then, to maintain atomicity, threads block before accessing an object unless they have the smallest timestamp among all threads recorded by the shelter that protects the object. This timestamp check is the function `shelter_wait(s)` in our runtime, and is captured by the transition semantics in our formalism. Here a thread examines all of the other threads registered for the shelter to see if it has the smallest timestamp.

Since threads in our system may block in an atomic statement, we must ensure that deadlock is not possible. In the presence of only closed atomic statements, this is straightforward. A timestamp is only acquired for the outermost atomic statement, and if two threads will touch the same objects in concurrently running atomic statements, the thread with the larger timestamp blocks right before accessing the first object it would touch that is shared by both atomic statements, while the thread with the smaller timestamp runs its atomic statement to completion. Thus, the apparent serial order of the atomic statements is given by the timestamps of the threads that run them. In other words, a thread that begins an atomic statement earlier never sees the effects from an atomic statement that started later.

In the presence of open atomic statements, the situation is more complicated. In particular, since threads must acquire a timestamp for each nested atomic statement, it is possible that waits-for cycles will be created when threads registered for the same shelters have intersecting ranges of timestamps. To avoid this situation, we require threads to block in the register operation if the registration would create a cycle. This cycle avoidance relies on knowing for which shelters a thread will attempt to register in the future. Therefore, before registering for a shelter, a thread must declare its intent to do so, which we call a *reservation*. This approach is similar to the "Resource-Allocation-Graph" deadlock-avoidance algorithm as described by Silberschatz, et al. [37]. The reservation is carried out by the `shelter_reserve()` function of our runtime, (reserve() in our formalism). In the reserve operation, a thread adds the indicated shelters to its reservation set.

If a thread will never again register for a shelter within the scope of the current outermost atomic statement, then it no longer needs the reservation. The deletion of the reservation is performed by the `shelter_unreserve()` function of our runtime. In the unreserve operation, a thread removes the indicated shelters from its reservation set. This operation is also performed by the reserve() operation in our formalism.

The precise conditions under which a waits-for cycle may exist in our system are given in the formalism by the *impedes* relation between two threads, described informally in

Section 3 and formally in Section 4. Which threads impede each other depends on which shelters they are registered for, and which shelters they have reserved. We detect the potential for deadlock by looking for cycles in a graph in which threads are nodes, and the directed edges are given by this relation.

In open-nesting, a thread completes an atomic statement by unregistering only from the shelters for the immediately enclosing atomic statement. In closed-nesting, a thread completes an atomic statement by unregistering from all shelters when exiting only the outermost atomic statement. This unregistration is carried out by the `shelter_register_pop()` function of our runtime (pop() in our formalism). It causes the relevant shelters to forget about the calling thread, which allows previously blocked threads with larger timestamps to continue.

Now that we have given an overview of our system's extensions to C and the runtime library for these extensions, we briefly describe two features that are necessary for handling real code.

In our extensions to C, atomic statements are translated into calls into the runtime library described briefly above. This translation is guided by a static analysis that discovers what shelters must be registered for at the start of each atomic statement. Any static analysis will be imprecise somewhere. Where ours is imprecise, we automatically introduce a hierarchy of shelters, in which shelters higher in the hierarchy subsume lower-level shelters. Our hierarchy is type-based, and we use a two-level hierarchy in which shelters in the upper level subsume shelters that are fields of structures. An example of this situation is described in Section 3.

Our compiler transformations detect where the shelter-hierarchy may be needed and add calls to set it up when the program begins, as well as to add new shelters when they are created dynamically. All a programmer must do is declare the "leaf" shelters. If the static analysis is imprecise, a thread will register directly for one of these ancestor shelters. Then, in the `shelter_wait(s)` function, a thread must wait not only when a thread with a smaller timestamp is registered for `s`, but also when a thread with a smaller timestamp is registered for a descendant or ancestor of `s`.

Finally, shelters may coexist with explicit locking. In order to avoid deadlock, our runtime must be informed of the existence of any explicit locks that may be used within atomic statements, in addition to their runtime state. To achieve this, we introduce "shadow" shelters, which capture the state of explicit locks for use in our deadlock avoidance algorithm.

3. Examples

In this section we explain each of the features described in Section 2 using examples. In the course of explaining these examples, we also give more detail about the operation of our system.

```
 1 typedef struct {
 2    int sheltered_by(s) id;
 3    float sheltered_by(s) balance;
 4    shelter_t s;
 5 } account_t;
 6
 7 needs_shelters(a->s)
 8 void deposit(account_t *a, float d) {
 9   a->balance += d; // shelter_wait(a->s);
10 }
11
12 needs_shelters(a->s)
13 void withdraw(account_t *a, float w) {
14   a->balance -= w; // shelter_wait(a->s);
15 }
16
17 needs_shelters(to->s, from->s)
18 void transfer(account_t *to, account_t *from,
19               float amount) {
20   atomic {
21     //shelter_reserve((W,to->s), (W,from->s))
22     //shelter_register((W,to->s), (W,from->s));
23     //shelter_unreserve((W,to->s), (W,from->s));
24     withdraw(from, amount);
25     deposit(to, amount);
26   } //shelter_register_pop()
27 }
```

Figure 1. Code using atomic statements for the atomic transfer of funds between two accounts. Statements inserted by the shelters compiled are in comments.

3.1 Example

In this example we show how shelter-based atomic statements avoid the pitfalls of explicit locking while achieving the convenience of atomic statements for the example of the atomic transfer of funds between two bank accounts.

Consider the type and function declarations in Figure 1. The `account_t` structure type contains a `balance` field, and an `id` field. The structure also contains a `shelter_t` field `s`. In our design, shelters, like locks, are first class objects, and can be declared as variables, structure fields, and function arguments. The types of the `balance` and `id` fields are annotated with `sheltered_by(s)`. This indicates that concurrent access to these structure fields is mediated by the shelter `s` in the same structure.

In the `transfer` function an `atomic` block indicates that the withdraw from the `from` account and the deposit to the `to` account happen atomically. That is, no other thread will see some intermediate inconsistent state.

The beginning of the atomic statement in the `transfer` function is translated into the `shelter_register()` call on Line 22. Here a thread registers for shelters `to->s` and `from->s`, which are indicated by the results of the static analysis.

In this call, W indicates that the shelters are needed for write access. The calls to reserve (and unreserve) shelters needed to support open nesting of atomic sections are discussed further in Section 3.3.

Before accessing a sheltered object, a thread must wait unless it is the thread registered for the shelter with the smallest timestamp. Therefore, where the program in our example accesses objects with types annotated with sheltered_by(s), calls to shelter_wait(s) are added that cause the calling thread to wait until it is the thread with the smallest timestamp on s. These calls appear for the accesses to the balance field of account structures on Lines 9 and 14.

When a thread exits an atomic statement, it unregisters itself from the shelters associated with that atomic statement. Accordingly, we add a call to shelter_register_pop() on Line 26, which unregisters the calling thread from shelters to->s and from->s.

The deposit and withdraw functions adjust the balance of an account. They must be annotated as needs_shelter(a->s) because they access fields of a that have been annotated as sheltered_by(s). Additionally, the transfer function must be annotated with needs_shelters(to->s,from->s) if it is ever going to be called from within an atomic statement because it calls functions that access data protected by those shelters.

Whether an error or a warning is emitted for missing needs_shelters() annotations depends on where a function is defined and called. If a function may be called within an atomic statement, lacks the correct annotation, is called in the same module it is defined in, and our system cannot deduce the annotation automatically (for example, when a shelter is the result of a function call), then an error is emitted. However if the called function is defined in a different module, then we infer that no shelters are required by the function, but emit a warning asking for an annotation in the header file. We found this arrangement to be the most expedient in converting our benchmarks because very few library calls touch sheltered objects.

Now, consider that one thread calls transfer(A, B, 0.10), and a second thread calls transfer(B, A, 0.10) at the same time. If the example in Figure 1 were written with explicit locking, care would have to be taken when implementing or calling the transfer function in order to avoid deadlock. With shelter-based atomic statements, however, deadlock is avoided automatically because the two threads will have distinct timestamps, one smaller than the other. If two threads are registered for the same shelter s, and both arrive at a shelter_wait(s) call, the thread with the smaller timestamp proceeds while the thread with the larger timestamp must wait until the first thread calls shelter_register_pop().

In the examples that follow, we show how this basic mechanism is extended to support both open- and closed-nesting of atomic statements, something that, to our knowledge, has not yet been implemented fully-pessimistically. Finally, we show how our mechanism can coexist with software using other synchronization mechanisms such as file locking. First, though, we explain how our design uses the shelter hierarchy to cope with imprecision in our static analysis.

3.2 Shelter Hierarchy

Consider the following alternate version of the transfer function from the example:

```
void idTransfer(int toId, int fromId, float a) {
  atomic {
    account_t *to = accountLookup(toId);
    account_t *from = accountLookup(fromId);
    withdraw(from, a);
    deposit(to, a);
} }
```

In this example the function accountLookup takes the account ID, looks up the account_t structure in some data structure implementing a map, and returns a pointer to it. It might be preferable to write the function like this in case, for example, accounts may be deleted from the system. Depending on how the map data structure is implemented, it may not be possible for a static analysis to determine exactly what shelters are needed at the beginning of the atomic statement. In this situation, a thread registers for a coarser grained shelter protecting *all* account_t structures, which subsumes the shelters in the individual account_t structures. We call these coarser-grained shelters *type-shelters*, and will refer to particular type-shelters as T.s, where T is the structure type name, and s is the shelter field, for example account_t.s. In our implementation they are only needed to subsume the shelters that are fields of structure types.[1]

Our system transforms programs such that calls to shelter_wait() are always made on leaf shelters. A thread must then examine each ancestor of this shelter to determine whether or not it must wait. For example, suppose thread T_1 is registered for a shelter protecting a particular account_t structure, a->s, and has timestamp 3. Further suppose that thread T_2 is in the atomic statement in the alternate transfer implementation, and is registered directly for the ancestor of a->s, account_t.s, with timestamp 2. Even if T_1 is the thread with the smallest timestamp registered for shelter a->s, it must wait because T_2 is registered with a smaller timestamp for account_t.s, an ancestor of a->s. On the other hand, if T_2 had only been registered for the shelter for another account_t structure, say b->s, then both threads would be able to proceed.

[1] The precision of the static analysis will affect the granularity of the hierarchy. An analysis that goes beyond our type-based aliasing assumptions will probably be finer-grained than the two-level hierarchy described here, and may achieve better performance at the cost of a whole-program pointer analysis. Our runtime implementation is able to support finer-grained hierarchies, but they are not generated by our static analysis.

3.3 Open Atomic Statements

Now consider a function that processes a list of transfers. The list of transfers is shared among threads, but a thread processing a list of transfers requires that the list does not change during processing. However, a thread only requires exclusive access to a pair of accounts when executing a particular transfer. It is safe with respect to the semantics of the program for other threads to access the other accounts while the rest of the list is being processed.

```
 open_atomic {
   for (t = l->head; t; t = t->next) {
    atomic {
      withdraw(t->from, a);
      deposit(t->to, a);
} } }
```

Open atomic statements have the same semantics as regular atomic statements with the exception that the effects occurring in any nested atomic statement becomes visible as soon as the nested atomic statement finishes. Here, the `open_atomic` statement indicates that accesses to `l->head` will not be interfered with during the execution of the enclosed statements. This would also be true if an `atomic` statement had been used. The `open_atomic` statement differs because it allows the effects of the transfers performed within the nested `atomic` statements to become visible at their respective ends.

Typically in implementations of open-nesting, it is the inner atomic statements that specify the semantics. In our design it is possible to use either convention. However, we felt it would be clearer if, by default, outer atomic statements could constrain the semantics of atomic statements nested within them. Consider that the inner atomic statement in the above example could be replaced by a call to the `transfer` function. If, by default, inner atomic statements specified the nesting semantics, then we would need two versions of this function, one with an open atomic statement, and the other with a closed atomic statement. When the default is for the outer atomic statement to specify the semantics, only one version is needed.

On the other hand, in some situations, it is always safe for the effects of an atomic statement to become visible immediately, regardless of where it is nested, for example for a `malloc` call. To handle these situations, we include a "forced-open" atomic statement, written `force_open_atomic {...}`. They have the same semantics as open atomic statements with the exception that the effects occurring in them become visible as soon as they finish, regardless of where they are nested.

To prevent deadlock in the presence of open nesting, the system needs to know an approximation of all shelters a thread will register for before the end of the outermost atomic statement (see Section 3.4). Recall that our static analysis tracks not only the shelters needed for the present atomic statement, but also about the shelters needed for nested atomic statements, and the atomic statement nesting structure where they are needed. This information is used to construct calls to `shelter_reserve()` and `shelter_unreserve()`. Our implementation instruments the above example as follows:

```
// open_atomic {
shelter_reserve((R,l->s), (W,account_t.s));
shelter_register((R,l->s));
shelter_unreserve((R,l->s));
shelter_wait(l->s);
for (t = l->head; t; t = t->next) {
 // atomic {
 shelter_register((W,t->from->s), (W,t->to->s));
 withdraw(t->from, a);
 deposit(t->to, a);
 shelter_register_pop();
 // }
}
shelter_unreserve((W,account_t.s));
shelter_register_pop();
// }
```

The call to `shelter_reserve()` expresses the fact that forthcoming nested atomic statements will register for read-only access to `l->s` and for write access to `account_t` structures. However, no actual registration occurs. The first call to `shelter_register()` registers the thread on shelter `l->s` with a first timestamp. In the second call to `shelter_register()` for the nested atomic statement, the thread registers itself on the indicated `account_t` shelters with a *second* timestamp. The calls to `shelter_unreserve()` indicate that the thread will perform no new registrations (until the end of the outermost atomic statement) on the specified shelters. In the first call to `shelter_register_pop()` the thread only unregisters from `t->from->s` and `t->to->s`, but does not unregister from `l->s` or forget that it may still in the future attempt to register for an `account_t` shelter.

3.4 Deadlock Avoidance

As described above, to prevent deadlock in the presence of open atomic statements, the static analysis ensures that we reserve with `shelter_reserve()` all the shelters that may be needed for the current and for nested atomic statements. We explain the reason for this by way of the artificial example of function `foo`.

In `foo`, deposits are made into two accounts, one in an outer atomic statement, and one in a nested atomic statement with open semantics. Now suppose that Thread 1 calls `foo(x,y)`, and Thread 2 calls `foo(y,x)` at the same time. Further suppose that Thread 1 obtains timestamp 1 when making its first registration call for `x->s`. Without any sort of deadlock avoidance, Thread 2 could then succeed at its registration with timestamp 2 for `y->s`. Thread 1 would then proceed through the first call to `deposit` on account `x`, and

```
void foo(account_t *a, account_t *b) {
  open_atomic {
    //shelter_reserve((W,a->s),(W,b->s));
    //shelter_register((W,a->s));
    //shelter_unreserve((W,a->s));
    deposit(a,0.10);
    atomic {
      //shelter_register((W,b->s));
      //shelter_unreserve((W,b->s));
      deposit(b,0.10);
    } //shelter_register_pop();
  } //shelter_register_pop();
}
```

obtain timestamp 3 when it registers for shelter y->s, and Thread 2 would proceed to obtain timestamp 4 when it registers for shelter x->s. Then, when both threads make their second call to deposit, they will be stuck. Registered for y->s, Thread 2 has an earlier timestamp than Thread 1, and registered for x->s, Thread 1 has an earlier timestamp than Thread 2.

However, when Thread 2 reaches its first registration call, it can observe that Thread 1 is already registered for shelter x->s and may try to register for y->s in a nested atomic statement, and that if it were to complete its own registration, a cycle like the one described above could be created. We formalize this notion of a cycle, and specify precisely our deadlock avoidance algorithm in Section 4, but here we give an informal description.

First, we say that two shelters may *interfere* with each other if they are the same shelter or if one is an ancestor of the other. Next, we say that thread A *impedes* thread B if thread A has a shelter registration that may cause thread B to block. Thread A may cause thread B to block if they are registered for shelters that *interfere* and thread A is registered with a smaller timestamp than thread B. Thread A is also said to *impede* thread B if thread A is registered for a shelter that *interferes* with a shelter that thread B has only reserved—that is, thread B could block waiting for thread A after it registers one of its reserved shelters. If there is a cycle in this relation, for example if we have *impedes(A,B)*, *impedes(B,C)*, and *impedes(C,A)*, then a deadlock *may* occur. We find cycles in the relation by constructing a graph in which threads are nodes, and in which there is a directed edge between a pair of nodes when the relation is true of the corresponding pair of threads.

It may seem superfluous for reservations to occur even before the beginning of a closed-atomic statement. However, this is necessary because we must perform deadlock avoidance not only before registration for an open atomic statement, but also for closed ones, as well.

3.5 Coexisting with Explicit Locking

In low-level systems code it is often necessary to retain some explicit locking, for example file locking in a database system. To see why this might be problematic, consider the following example.

```
void accountFileLock(account_t *a, int mode) {
  fcntl(a->file, mode);
}
void account_file_open(account_t *a) {
  atomic {
    accountFileLock(a, WRITE);
} }
void account_file_close(account_t *a) {
  atomic {
    accountFileLock(a, UNLOCK);
} }
```

In this example we have added a file field to our account_t structure, which is also protected by the account structure's shelter. The atomic statements here protect account structures from concurrent access while an account's file is being opened and locked, and closed and unlocked. But the possibility for deadlock exists when, for example, the thread holding the file lock must wait on the shelter protecting the account structure before releasing the lock on the file. When code like this is exposed as library calls, as in sqlite, it is not possible to fix the problem by adding additional atomic statements.

We address this problem by introducing "shadow" shelters that follow the state, and track the owning thread, of an explicit lock in the program. Shadow shelters are declared with type shadow_shelter_t. The shadow shelters change state when explicit locking calls are made based on programmer supplied annotations on the explicit locking functions. The annotation shadow_change(s,c) on a function indicates that the function changes the state (e.g. locked, unlocked) of an explicit lock. Here, s is an expression for the shadow shelter for the lock, and c is an integer expression for the new state, both in terms of formal parameters and global variables. Our static analysis determines that an atomic statement requires shadow shelter s' when a function annotated with shadow_change(s,c) is called within it, where s' is s with actual arguments substituted for formal arguments.

In the above example, we would add a shadow_s field to the account_t structure of type shadow_shelter_t, and annotate the accountFileLock function with shadow_change(a->shadow_s, mode). Our implementation then instruments calls to accountFileLock with calls to a function in our runtime called shadow_change_state(a->shadow_s,mode), which updates the shadow shelter a->shadow_s to reflect the state of the explicit lock, mode. In our implementation, we assume that the external lock is unlocked when the mode is zero, and locked otherwise. Further, shadow_change_state(a->shadow_s,mode) must block when a thread is attempting to set the mode to a locked mode, but the thread does not have the smallest timestamp on the shadow shelter. We leave as future work extending this mechanism to support other locking modes.

$$
\begin{array}{llcl}
\text{Declaration} & d & ::= & \text{int } v \text{ sheltered by } x \\
\text{Trace} & T & ::= & (t_1, s_1), \ldots, (t_m, s_m) \\
\text{Statement} & s & ::= & \text{reserve}(\sigma_1, \ldots, \sigma_m) \\
 & & | & \text{register}(\sigma_1, \ldots, \sigma_m) \\
 & & | & \text{pop} \\
 & & | & v := v_1 + v_2 + n \\
\text{Shelter } (\Sigma) & \sigma & ::= & v_\sigma \mid x \\
 & & & \\
\text{Identifiers} & v, x & & \text{Integers} \quad t, n, m
\end{array}
$$

Figure 2. Traces of shelter-based programs.

Finally, a thread may only complete a registration when the shadow shelter says that the file lock is unlocked, or when the current thread is the file lock's owner. Thus, the call to `accountFileLock` will never block and cause deadlock. This strategy will also work for explicit locking calls where both the lock and unlock call are in the same atomic statement.

3.6 Discussion

This preponderance of annotations undoubtedly makes our design seem no less complex and confusing than simply using explicit locks. In particular, our insistence on avoiding whole-program analysis, and the inclusion of open-nesting, and shadow shelters, requires us to add several constructs beyond simple `atomic` statements and the `sheltered_by` type qualifier. However, this increase in complexity comes only in response to the complex protocols and invariants that we see in existing software. If we removed some of these annotations for the sake of simplicity, our design would be demonstrably less efficient, or less complete.

4. Atomicity

We give a trace-based formalism for our design in order to clarify the requirements on a shelter compiler and show that our system provides two useful guarantees: *progress* (no deadlock) and *partial atomicity* (atomic statement effects are visible once committed to those statements that started later). Our formalization is fairly different to that used for Jade [32] as we are proving a different property (progress and atomicity vs. equivalence to a sequential program).

A trace T (Figure 2) captures the essential operations executed by a shelter-based program. A trace is executed in the context of declarations $d_1, \ldots, d_n$ of the global integer variables used in the trace. Each variable v is protected by its own fine-grained shelter v_σ and a coarse-grained shelter s (possibly shared with other variables). These declarations implicitly define a partial order on shelters ($v_\sigma \le v_\sigma, v_\sigma \le s, s \le s$). This order mirrors our implementation's hierarchy, with the fine-grained shelters matching shelter_t objects and the coarse-grained shelters matching the type-based shelters like account_t.s. We say that a shelter σ_1 *subsumes* a shelter σ_2 if $\sigma_2 \le \sigma_1$. For simplicity, we omit the distinction between read and write shelter access.

The trace itself is a sequential interleaving of statements from multiple threads, with each thread identified by a distinct integer t. The statements, executed atomically, are either assignments of a computed value to a variable v,[2] or one of the three low-level operations used to implement atomic statements: reserve,[3] register and pop. The reserve operation allows future registration for shelters, or shelters subsumed by, $\sigma_1, \ldots, \sigma_m$, while the register operation begins an atomic statement, registering for shelters, $\sigma_1, \ldots, \sigma_m$. Finally, the pop operation releases the shelters registered for by the most recent, still active reserve statement. For instance, a possible trace for `atomic { j := 3 }` is

$$(0, \text{reserve}(j_\sigma)), (0, \text{register}(j_\sigma)), (0, \text{reserve}()),$$
$$(0, j := 3), (0, \text{pop})$$

Thread 0 simply reserves then registers for the required shelter, performs the atomic statement body, then releases all its shelters. The reserved shelters are released as soon as they will no longer need to be registered for, to avoid impeding other threads (see below).

An open atomic statement is more complicated. A possible trace for

```
open_atomic { p := p+1; atomic { q := p+q; } }
```

in a context where both p and q are sheltered by s is:

$$(0, \text{reserve}(s)), (0, \text{register}(p_\sigma)), (0, \text{reserve}(q_\sigma)),$$
$$(0, p := p + 1), (0, \text{register}(q_\sigma)), (0, \text{reserve}()),$$
$$(0, q := p + q), (0, \text{pop}), (0, \text{pop})$$

Thread 0 initially reserves the coarse-grained shelter s which protects both p and q. The first open atomic statement only registers for fine-grained shelter p_σ. The thread then refines its reservation to the q_σ shelter, which it will subsequently register for. To start the inner atomic statement, it additionally registers for the q_σ shelter and releases all its reservations as it no longer needs them. Finally, it ends by terminating both atomic statements, releasing all shelters.

Traces do not necessarily represent a valid execution of a shelter-based program. For instance, if p is sheltered by s, $(0, \text{p} = 2)$ accesses p without being registered for either its fine-grained shelter p_σ or its coarse-grained shelter s. This corresponds to a thread accessing a shelter-protected object outside of an atomic statement, which is an error in the program itself. We call this type of error a *single-threaded error*. Our trace operational semantics will send traces with this kind of error into an error state.

[2] The atomic execution of $v := v_1 + v_2 + n$ is not essential and could easily be relaxed with the addition of per-thread variables to the trace formalism.

[3] The `shelter_unreserve` function is represented in the trace by a reserve on the resulting reservation set.

$$\frac{\text{regfor}(H,t,v) \qquad \text{regfor}(H,t,v_1) \qquad \text{regfor}(H,t,v_2)}{R,H \models (t, v := v_1 + v_2 + n)} \qquad \frac{\text{access}(H,t,v) \qquad \text{access}(H,t,v_1) \qquad \text{access}(H,t,v_2)}{\begin{array}{c} M,R,H,a : (t, v := v_1 + v_2 + n) \rightarrow \\ M[v \rightarrow M(v_1) + M(v_2) + n], R, H, a \end{array}}$$

$$\frac{H(t) = \emptyset \vee \text{subsumed}(\{\sigma_1, \ldots, \sigma_m\}, R(t))}{R,H \models (t, \text{reserve}(\sigma_1, \ldots, \sigma_m))} \qquad \frac{R' = R[t \rightarrow \{\sigma_1, \ldots, \sigma_m\}]}{M,R,H,a : (t, \text{reserve}(\sigma_1, \ldots, \sigma_m)) \rightarrow M, R', H, a}$$

$$\frac{\text{subsumed}(\{\sigma_1, \ldots, \sigma_m\}, R(t)) \qquad m \geq 1}{R,H \models (t, \text{register}(\sigma_1, \ldots, \sigma_m))} \qquad \frac{\begin{array}{c} H' = H[t \rightarrow H(t) \cup \{(a, \sigma_1), \ldots, (a, \sigma_m)\}] \\ \neg\text{cycle}(\text{impedes}(R, H')) \end{array}}{M,R,H,a : (t, \text{register}(\sigma_1, \ldots, \sigma_m)) \rightarrow M, R, H', a+1}$$

$$\frac{H(t) \neq \emptyset}{R,H \models (t, \text{pop})} \qquad \frac{\begin{array}{c} b = \max\ \{a \mid (a, \sigma) \in H(t)\} \\ H' = H[t \rightarrow \{(a, \sigma) \mid (a, \sigma) \in H(t) \wedge a \neq b\}] \end{array}}{M,R,H,a : (t, \text{pop}) \rightarrow M, R, H', a}$$

$$\frac{R,H \not\models (t_1, s_1)}{M,R,H,a : (t_1, s_1), \ldots, (t_m, s_m) \rightarrow \bot, \bot, \bot, \bot : \epsilon} \qquad \frac{R,H \models (t_1, s_1) \qquad M,R,H,a : (t_1, s_1) \rightarrow M', R', H', a'}{\begin{array}{c} M,R,H,a : (t_1, s_1), \ldots, (t_m, s_m) \rightarrow \\ M', R', H', a' : (t_2, s_2), \ldots, (t_m, s_m) \end{array}}$$

$$\begin{array}{rcl} \text{regfor}(H,t,v) & = & \exists (a, \sigma) \in H(t). v_\sigma \leq \sigma \qquad\qquad \text{subsumed}(\Sigma_1, \Sigma_2) \;=\; \forall \sigma_1 \in \Sigma_1. \exists \sigma_2 \in \Sigma_2. \sigma_1 \leq \sigma_2 \\ \text{interferes}(\sigma_1, \sigma_2) & = & \sigma_1 \leq \sigma_2 \vee \sigma_2 \leq \sigma_1 \\ \text{access}(H,t,v) & = & \exists (a, \sigma) \in H(t). (v_\sigma \leq \sigma \wedge \forall t' \neq t. \forall (a', \sigma') \in H(t'). \text{interferes}(v_\sigma, \sigma') \Rightarrow a < a') \\ \text{impedes}(R,H)(t_1,t_2) & = & \exists (\sigma_1, a_1) \in H(t_1). (\exists (\sigma_2, a_2) \in H(t_2). a_1 < a_2 \wedge \text{interferes}(\sigma_1, \sigma_2) \vee \\ & & \quad \exists \sigma_2 \in R(t_2). \wedge \text{interferes}(\sigma_1, \sigma_2)) \end{array}$$

Figure 3. Trace Operational Semantics

A different kind of error is present in the following trace:

$$\begin{array}{l} (0, \text{reserve}(p_\sigma), (0, \text{register}(p_\sigma)), (0, \text{reserve}()), \\ (1, \text{reserve}(p_\sigma), (1, \text{register}(p_\sigma)), (1, \text{reserve}()), \\ (1, p := 1) \end{array}$$

Thread 1 accesses p, but it should have been blocked because thread 0 registered the shelter p_σ earlier in the trace. We call this kind of error a *multi-threaded error*. We will show that these kinds of errors are forbidden by our trace operational semantics; there will be an implicit `shelter_wait()` operation in front of each variable access.

Figure 3 gives an operational semantics for traces that makes a clear distinction between these two kinds of invalid traces. The state of the operational semantics is a four-tuple M, R, H, a where $M : \text{id} \rightarrow \mathbb{N}$ maps variables to their values, $R : \mathbb{N} \rightarrow \mathcal{P}(\Sigma)$ maps threads to their current shelter reservations, $H : \mathbb{N} \rightarrow \mathcal{P}(\mathbb{N} \times \Sigma)$ maps threads to the shelters they are registered for, and $a : \mathbb{N}$ is the timestamp of the last register statement. Registrations for a shelter are described by a pair (a, σ) of the register statement's timestamp and the shelter itself.

The single-threaded errors are explicitly checked for by the $\models$ relation: $R, H \models (t, s)$ holds if executing s is not a single-threaded error for thread t in state R, H — it is easy to verify by inspection of the rules in Figure 3 that violations of the $\models$ relation can only be caused by the thread itself. The $M, R, H, a : (t, s) \rightarrow M', R', H', a'$ transition models the atomic execution of a single statement s by thread t and the corresponding state change. This transition is only defined when it does not cause a multi-threaded error; the requirements of the $\rightarrow$ transition effectively represent the conditions for which a thread must wait until it can execute s.

The single-threaded errors formalize the requirements on (and freedoms of) the shelter compiler: it must never generate code that would violate the $\models$ relation. That is, in the trace semantics and our implementation, the $\models$ relation checks at runtime that the results of the static analysis that generated the calls to reserve() and register() was correct. The two assignment rules use the *regfor* relation to check that the thread is registered for a shelter that subsumes the variable's shelter. The reserve statement can reserve new shelters when the thread is not currently registered for any shelters ($H(t) = \emptyset$): this corresponds to the requirement discussed in the overview to conservatively estimate the shelters required by an atomic statement prior to entering it. The reserve statement can also release or refine its current reservation (checked by the *subsumed* relation); this effectively allows a shelter compiler to change the reservation at any time to its best approximation of the shelters that will still be registered for in the remainder of the outermost atomic statement. The register statement can only register for shelters that have been previously reserved, and the pop statement cannot occur when no atomic statement is active.

The $\rightarrow$ transitions perform straightforward updates to the current state. Only two rules include preconditions that could lead to a multi-threaded error: the two assignment rules use the *access* rule to check that no other thread has registered for a shelter for the variable with a smaller timestamp. The *access* rule encodes the conditions under which a thread does not need to block for a `shelter_wait()` call.

The rule for register requires that registering for the shelters will not create a cycle in the the directed graph formed by considering the threads to be nodes, and the edges given by the impedes(R, H) relation (we call this the *impediment-graph*). Essentially, impedes captures when a thread might block the progress of another thread sometime in the future: when a thread is registered before another for an interfering shelter, or when a thread is registered for a shelter that interferes with a shelter that another thread plans to register for later. The impedes relation itself depends on the current state of registrations and reservations, so we write it as a function of these mappings. Throuh a little abuse of notation, we write $\neg$cycle(impedes(R, H)) to indicate that there are no cycles in the impediment-graph given the registrations and reservations in R and H. This cycle-detection is necessary for us to avoid deadlock while supporting open atomic statements as described earlier. In particular, it deals with the fact that threads may be registered for different shelters with different timestamps.

Definition 1. *Trace evaluation.* In the context of declarations $d_1, \ldots, d_n$, trace:

$$(t_1, s_1), \ldots, (t_m, s_m)$$

evaluates to M, R, H, a if

$$M_0, R_0, H_0, 0 : (t_1, s_1), \ldots, (t_m, s_m) \overset{*}{\rightarrow} M, R, H, a : \epsilon$$

where the initial state has $M_0(v) = 0$ for all variables v in $d_1, \ldots, d_n$, and $R_0(t) = H_0(t) = \emptyset$ (no shelters registered for or reserved) for all threads t in $t_1, \ldots, t_m$.

Trace evaluation ends in the error state $(\bot, \bot, \bot, \bot)$ if the trace has a single-threader error, but is not defined in the presence of a multi-threaded error.

Theorem 1. *Atomic-statement progress.* If trace

$$(t_1, s_1), \ldots, (t_m, s_m)$$

evaluates to M, R, H, a in context $d_1, \ldots, d_n$ then the extended trace

$$(t_1, s_1), \ldots, (t_m, s_m), (t, s)$$

evaluates to M', R', H', a', where either:

- if $\forall t'.H(t') = \emptyset$, then t is any thread, s is any statement, or
- $H(t) \neq \emptyset$, and s is any statement

Proof: By induction over traces, using the fact the trace semantics ensures that $\neg$cycle(impedes(R, H)). As a consequence, some thread t with $H(t) \neq \emptyset$ has no incoming edge from any other thread in the impediment-graph. This thread can attempt to execute any statement s: either this statement will cause a single-threaded error, or by construction of t, $M, R, H, a : (t, s) \rightarrow M', R', H', a'$.

In essence, the above theorem states that programs that have no single-threaded errors and wait for the preconditions of $\rightarrow$ can always make progress, and, in particular, it can never be the case that all threads in atomic statements are blocked.

Theorem 2. *Partial Atomicity.* In a trace

$$(t_1, s_1), \ldots, (t_n, s_n)$$

that evaluates to M, R, H, a, let M_i, R_i, H_i, a_i be the state of M, H, R, a before the ith step of the trace. If

$$s_i = \text{register}(s_1, \ldots, s_m)$$

and $s_j =$ pop is the end of this atomic statement then, for all variables v such that $\neg$regfor$(H_i, t_i, v) \wedge$ regfor(H_{i+1}, t_i, v) (the atomic statement gave access to v) the effects of the writes by thread t_i to v between s_i and s_j are visible to exactly those atomic statements of other threads that started after s_i **and** ended after s_j.

We prove Theorem 2 in Appendix A. It essentially states that updates to objects protected by an atomic statement become visible to other threads only after the end of the atomic statement. Note that it is safe for threads with concurrently running atomic statements to access the same objects, but the atomic statement that started later may not access these objects until the atomic statement that started earlier ends. This theorem applies to both open and closed atomic statements.

5. Implementation

Our implementation is written in about 3600 lines of OCaml using the CIL [29] library, with a runtime library written in about 4000 lines of C. We use a combination of lock-free algorithms and other optimizations to ensure that our implementation scales.

In this section we describe optimizations. We also discuss issues that arise when using shelters in existing C programs, such as condition variables, library calls, polymorphism, and interaction with synchronization strategies not based on atomic statements.

5.1 Optimizations

In practice, a number of optimizations are required for shelters to scale. In particular, the following optimizations were critical to achieving performance similar to, and in some cases, better than explicit locking. Without any one of these optimizations, at least one of our benchmarks will fail to exceed sequential performance when explicit locks do.

First, we track shelter registration using a queue on each shelter of registered threads, sorted by timestamp. Furthermore, threads are also placed on the queues of a shelter's ancestor, with an indication that their initial registration was for a descendant. This allows us to efficiently implement `shelter_wait()` (and the cycle-detection for deadlock-avoidance in `shelter_register()`); we only ever need to inspect a short initial segment of the queues at each level of the hierarchy.

When a shelter is acquired for read-only access, waiting for a shelter can be optimized: a thread must only wait until it has a timestamp smaller than threads on the shelter's queue that have registered for write access. This mechanism is essentially equivalent to explicit read/write locking, however our shelter-based atomic statement implementation invokes it automatically wherever possible.

It is often the case that atomic statements are used to protect access to objects for which there is not much contention. To take advantage of this, instead of adding itself to a shelter's queue, our runtime allows a thread to acquire a spinlock—or to increment a counter in the case of read-only access when there are no writers—during the registration phase in the case that there are no other threads on the shelter's queue. This also reduces contention for the global timestamp counter.

Furthermore, not all programs will use the various levels of the shelter hierarchy, and those that do will not be using them at all times. Therefore, our runtime includes a mechanism to activate shelters higher up in the hierarchy only when they are needed—that is, when a thread attempts to register for them directly. When a shelter is inactive, threads registering for its children must simply read a flag that indicates inactivity, and check that its queue is empty to see that no further action is required. Threads also record for which inactive shelters they have registered. When a thread wishes to register directly for a shelter with children, it registers as usual, but during a wait call, it must also wait until there are no threads registered for a child shelter that make use of the inactivity of the parent shelter. This check is made by examining the inactive shelters that each thread has registered for. When there are no more such threads, the thread unsets the inactive flag in the parent shelter, and proceeds when the usual conditions for a `shelter_wait()` are met.[4]

As the number of threads and cores increases, contention for the global sequence number counter increases. To address this, when the compare-and-swap operation used to increment the counter fails, after retrying immediately a small number of times, we use a binary exponential back-off algorithm [17]. In our experiments, this approach reduced by several orders of magnitude the number of compare-and-swap failures without compromising performance.

Potential Optimizations Our static analysis can tell when a shelter will no longer be used before the end of an atomic statement, when a shelter will no longer be used for write access, and when a child shelter can be held instead of a parent shelter. These optimizations are all sound in the absence of open atomic statements, but we have not yet determined the full conditions under which they can be used with nested atomic statements (see Appendix A), so we leave them for future work.

Another potential optimization that we leave for future work involves the way that timestamps are allocated to threads. If it can be determined that two threads will never use the same shelter, then their timestamps need not be distinct. An analysis supporting such an optimization would likely rely on some form of a must-not-alias analysis [28].

Finally, another potential optimization for timestamp allocation proceeds as follows. Each shelter could be randomly placed in one of a fixed number of sets of shelters. Each set would have a separate timestamp counter. When a thread registers for a set of shelters at the start of an atomic statement it would atomically acquire timestamps from each of the sets the shelters belong to. That is, for each registration, a thread would be granted a set of timestamps. `shelter_wait` would proceed as before, using the appropriate timestamp for each shelter.

We have not yet implemented these techniques for reducing contention on the timestamp counter because the backoff optimization was effective enough on our benchmarks that this contention was no longer the performance bottleneck.

5.2 Condition Variables

Our implementation includes support for condition variables. That is, threads may send signals and wait on condition variables based on shared state that is protected by shelters. Shelter condition variables are much like traditional condition variables. They are declared like pthread condition variables, e.g. `shelter_cond_t scv`, and are signaled in the same way, e.g. `shelter_cond_signal(scv)`. However, a conditional wait on shelter protected state is slightly different. We introduce the following construct: `shelter_cond_wait(scv,e) s`.

The meaning of this statement is as follows. The thread waits on the shelter condition variable `scv` while the condition, `e`, is false. If the thread is then signaled, and the condition is true, it executes the statement `s` atomically. This is accomplished by our analysis treating the shelter wait block as an atomic statement and collecting the shelters necessary for protecting both the block and the condition `e`. Then, the above construct can be translated as follows. The thread registers for the shelters found by the analysis, waits on them, and then checks `e`. If `e` is false, it atomically releases the shel-

[4] It may seem simpler for threads registering for a child shelter to simply acquire read-access to the inactive parent shelter by way of a counter, as in a reader-writer lock. However, in practice, contention for this counter can incur an unacceptably high overhead. We shift the cost to the uncommon operation of waiting on a non-leaf shelter.

ters and sleeps on scv. If e is true, then the thread proceeds as in a normal atomic statement. We leave as future work an extension to our implementation that ensures that condition variables are signaled when appropriately specified state is updated.

5.3 Library Calls and Polymorphism

If a call is made into a library that does not use shelters, and does not use any callbacks, then it will not cause a thread to register for any shelters. Therefore, it is only necessary to know what objects such a library call will access in case any of these locations are protected by a shelter. We allow programmers to indicate this by providing annotations that summarize the read and write behavior of library calls, so that our implementation can automatically place the appropriate wait calls ahead of them. Library calls invoking callbacks that access shelter protected state are not currently supported by our system, though we believe this issue could be addressed by allowing the programmer to describe the behavior of callbacks with an annotation.

In our current implementation we do not support type-qualifier polymorphism for the `sheltered_by(s)` annotations. This sort of feature has not been needed in the benchmarks we analyze in Section 6 due to the limited use of polymorphism in C programs. However, more modern languages may require increased support of polymorphism to allow code-reuse, and other good software engineering practices. We leave support for polymorphism as future work.

5.4 Other Synchronization Strategies

It is not realistic to assume that all shared data will be protected by shelters and accessed within atomic statements. For instance, some shared data will be read-only and need no synchronization, while other data will be protected by other means: barrier synchronization, data obtained from work queues and worked on exclusively by a single thread, etc. The programmer must ensure that data is shared correctly using consistent mechanisms. Our own previous work in SharC uses sharing annotations on all types [5], while Martin et al. [24] use dynamic ownership assertions to detect where such rules are violated. Furthermore, external libraries may already use locks to protect their own data — converting these libraries to use shelters may not be desirable, practical or even possible. For these cases in our sqlite benchmark, our shadow shelters mechanism has proved adequate.

6. Evaluation

We have modified a number of programs to use shelters as the mechanism for enforcing atomic statements, and we have measured the runtime performance of these programs on realistic inputs. The purpose of this evaluation is to investigate the convenience of using shelter-based atomic statements, and to compare the runtime performance of our implementation with five other mechanisms for enforcing atomicity, namely explicit locking, Autolocker, software transactional memory, a single global lock, and shelters implemented with pthread reader-writer locks. We chose these implementations for comparison because they cover existing techniques for C not requiring special hardware or by-hand instrumentation. The STM implementation allows us to compare against an optimistic approach, and Autolocker and the others allow us to compare against the various ways of implementing atomic statements pessimistically.

In the alternate implementation of shelters using reader-writer locks (RWLocks) each shelter contains a lock. When registering for shelters at the beginning of atomic statements, the locks are sorted by address to avoid deadlock before being acquired. When a fine-grained shelter is registered, the shelter's lock is acquired in read mode if the section only reads the sheltered data, and in write mode otherwise. When a coarse-grained shelter is registered, the lock is acquired in write mode. A shelter's ancestors' locks are always acquired in read mode.

6.1 Experimental Setup

All of our experiments were performed on a 2.27GHz Intel Xeon X7560 machine with four processors each with eight cores having 32GB of main memory running Linux. Hyperthreading was turned off. We chose to compare against an Intel compiler for C/C++ that includes an STM implementation [35]. Other STM implementations may give better performance [11], but the Intel STM compiler has an annotation burden that is similar to that of our implementation, and requires no additional real or simulated hardware. We also used the Intel compiler with the STM features disabled as the back-end of our implementation, and to compile the other versions of the benchmarks. The compile-time analyses used for our implementation did not add significantly to compilation time.

6.2 Benchmarks

Our benchmark programs consist of the STAMP STM benchmark suite [11], along with: pbzip2, a parallel version of bzip2; pfscan, a parallel file scanning tool; ebarnes, an n-body simulation using the Barnes-Hut algorithm [6]; open-atomic, a micro-benchmark that uses open atomic statements; and the sqlite database system. Table 1 shows the size of each of the benchmark programs, the number of atomic statements in each, and the number of other annotations that were needed to use Intel's STM, shelters, and Autolocker, respectively. The other annotations for Intel's STM are the `tm_callable` annotations that must be placed on functions called from transactions. The other annotations for our system are the `sheltered_by()` annotations and the `needs_shelters()` function annotations. Autolocker's only annotations are its `protected_by` annotations.

The STAMP benchmarks are distributed with the `tm_callable` annotations already placed, some of which are redundant. We also made redundant annotations when adapt-

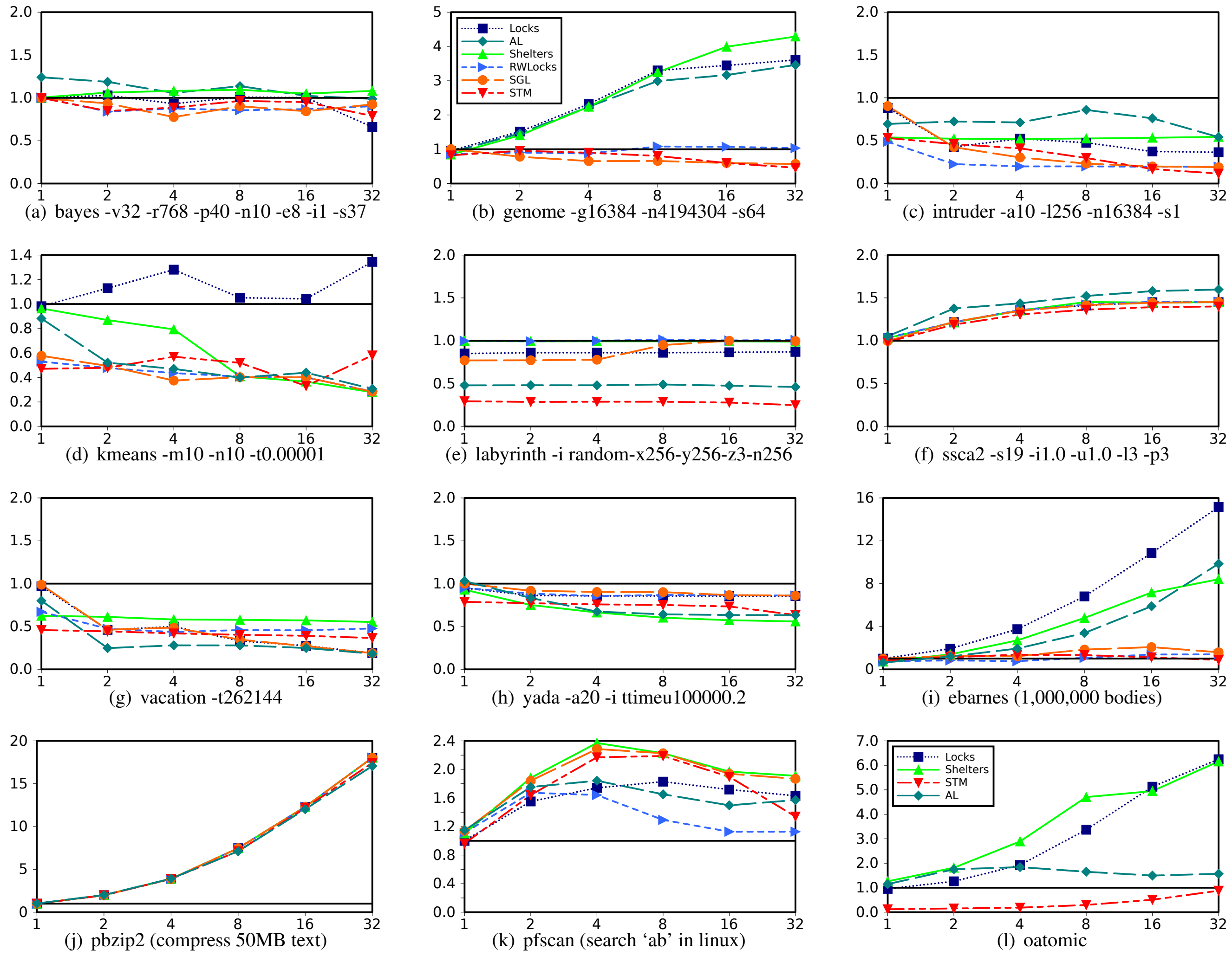

Figure 4. Graphs (a) - (k) show speedup over sequential runs versus the number of threads used for our benchmark programs when run with explicit locking (Locks), Autolocker (AL), shelters (Shelters), Intel STM (STM), a single global lock (SGL), and shelters implemented with reader-writer locks (RWLocks). Higher is better.

ing the application benchmarks for shelters as we also found the annotations to be useful for documentations purposes. The annotation counts in the table include these redundant annotations. This is why in the STAMP benchmarks the annotation count for STM is sometimes higher than it is for shelters. The annotation count for shelters on the sqlite application would have been as high as the count for STM, but our call-graph analysis allowed us to avoid making module-local function annotations. A similar analysis would benefit STM to the same extent.

The results of all our experiments except sqlite are given by the graphs in Figure 4. The graphs show speedup over sequential runs (i.e. $T_{sequential}/T_{parallel}$) versus the number of cores used. Each reported result is the average of 50 runs. Command line arguments passed to the STAMP benchmarks are also given in the captions. Below, we describe all the results we observe before discussing the conclusions we draw from them.

6.2.1 STAMP Benchmarks

The intended usage of the STAMP benchmark suite is to compare different transactional memory implementations. In addition, we believe that it is also a suitable benchmark suite for the more general task of comparing different implementations of atomic statements. The algorithms used in some of the benchmarks have synchronization strategies that are very challenging to handle well with any implementation.

We note that the suite covers variation across the following dimensions: contention, length of time in atomic statements, and the number of shared memory accesses in atomic statements. Because it attempts to cover this space, on several of the benchmarks, none of the automatic software-

Name	Size (kloc)	Atm. stms.	STM Anns.	Shelt. Anns.	AL Anns.	Seq. Time
bayes	12.0	15	47	42	11	9.97s
genome	10.0	5	16	25	11	8.58s
intruder	11.3	3	61	64	9	2.26s
kmeans	3.9	3	4	7	6	9.17s
labyrinth	8.2	3	50	46	13	3.02s
ssca2	9.2	1	5	12	9	9.73s
vacation	11.0	3	159	122	6	1.53s
yada	13.4	6	105	86	12	4.19s
ebarnes	13.4	3	8	9	7	16.07s
pbzip2	10.0	10	4	14	12	10.46s
pfscan	2.8	6	4	11	12	2.56s
oatomic	0.3	4	0	9	8	8.56s
sqlite	131.0	97	2229	161	156	-
total	236.5	149	2692	608	272	

Table 1. Program size, number of atomic statements, number of annotations for Intel STM, Shelters, and Autolocker, and sequential runtime for our benchmark programs.

based approaches we investigate here manage to scale. Scaling on these benchmarks seems to require either specialized hardware, or by-hand instrumentation along with a knowledge of implementation details [11].

For the bayes, intruder, labyrinth, vacation, and yada benchmarks contention is high, and threads make many shared memory accesses in atomic statements. The genome benchmark includes lightly contended atomic statements that make many shared memory accesses. The kmeans benchmark includes highly contended atomic sections that make many updates to shared memory. The ssca2 benchmark includes lightly contended atomic statements that make only a few shared memory accesses. Scaling of ssca2 is limited by the sequential portion of the benchmark.

The bayes, vacation, and yada benchmarks required activation of our shelter hierarchy. The others did not. Autolocker initially reported deadlock for these benchmarks due to its inability to compute a global lock-order. To address this, following the technique outlined by Autolocker's authors, we added global locks by hand until it accepted the programs.

6.2.2 Open Atomic Statements

To show the benefit of open nesting, we constructed a micro-benchmark to compare our implementation of open-nesting with explicit locking, Autolocker, and Intel STM. The form of this micro-benchmark is taken from a pattern that appears a number of times in the Linux kernel, the MySQL database server, and sqlite. The micro-benchmark resembles the open-nesting example in Section 3.

Occasionally one thread computes a summary of data contained in a collection of linked lists. While this thread is computing, other threads are permitted to access individual linked-lists, but not to add or remove lists from the collection:

```
lock_acquire(S.lock);
for (i = 0; i < S.len; i++) {
  list *l = S.lists[i];
  lock_acquire(l->lock);
  list_ops(l);
  lock_release(l->lock);
}
lock_release(S.lock);
```

A system having only closed atomic statements would be unable to capture this safe way of increasing concurrency.

In our oatomic benchmark, one thread computes a summary of the collection of linked lists, while a number of other threads modify randomly selected lists from the collection. The results of this experiment are presented in Figure 4(l).

6.2.3 Application Benchmarks

Our three multithreaded C applications that use threads and locks were chosen to compare the real-world performance of shelters with the other implementations. The pbzip2 benchmark is a parallel implementation of the popular block-based compression algorithm. Atomic statements are short, lightly contended, and make only a small number of shared memory accesses. For this benchmark, a 50MB text file was compressed. The pfscan benchmark is a tool that searches for a string in all files under a given directory tree. For this benchmark, we searched for the string "ab" in the Linux source code tree. The atomic statements are short, lightly contended, and make only a few shared memory accesses. Calls into the operating system and limited workload size impeded performance gains beyond 8 cores for each of the synchronization mechanisms. The ebarnes benchmark is an n-body simulation adapted from the Barnes-Hut Splash2 benchmark [38]. Performance of its oct-tree building phase is very sensitive to the synchronization strategy. Contention is low, but many shared memory accesses are made in atomic statements. For this benchmark, we simulated 1 million bodies so that building the oct-tree in parallel would give a significant performance advantage.

6.2.4 Sqlite

To show that our implementation is capable of scaling to large pieces of systems-level software, we ported the sqlite database system to use shelters instead of explicit locking with mutexes. Sqlite is composed of about 130k lines of code, and uses mutexes to protect database connections, btrees, a shared page cache, the memory-management subsystem, and its pseudo-random number generator. Additionally, it uses file locks and locking of pages in btrees to ensure the atomicity of database transactions. These file and page locks are frequently held across API calls.

In porting sqlite, we exercised all of the features of our implementation. In particular, we used open-nesting for the memory, page-cache, and random number generator subsystems. Furthermore, shadow shelters were required to prevent locks on files and btree pages from causing deadlock, as indicated by the example in Section 3. This deadlock problem was not created by our use of atomic statements and shelters, though in the absence of our shadow shelters, it is exposed more readily when imprecision in our static analysis requires the use of the shelter hierarchy. The default explicit locking version of sqlite deals with this issue by requiring application code to manually roll-back database transactions when certain lock acquisitions fail. Our shadow shelters hide this issue from client applications, however it is likely sometimes useful for applications to deal directly with contention of this sort. We leave the addition of customizable contention management to our shelters-based implementation for future work.

Statistics for the sqlite benchmark appear in Table 1. We evaluated the performance of our implementation by running sqlite's multithreaded regression suite, which uses between 2 and 12 threads, and comparing only against the original explicit locking version; attempting to use Autolocker, the Intel STM, or a single global lock resulted in deadlock due to the file locking problem.

The unmodified sqlite runs the multithreaded regression suite in roughly 4 minutes, whereas the shelters version took around 8 minutes. Because of the imprecision of our static analysis, namely the type-based aliasing assumptions, many of the atomic statements acted effectively as a single global lock. We believe that an automatic approach to acquiring finer-grained locks or shelters in this context may only be possible in the presence of at least some information about the shape of data structures.

It is also interesting to note that in the process of converting sqlite to use atomic statements, we were able to remove a few hundred lines of code dedicated to deadlock avoidance in its btree implementation.

6.2.5 Effects of Workload Size

We also did an experiment in which the number of bodies simulated in the ebarnes benchmark was varied from 100k bodies, which fit comfortably in the CPU caches, to 8 million bodies, which exceeded the capacity of the caches. We ran the simulations with each implementation, and on 4, 8, 16, and 32 cores. We did not observe any changes in relative overhead with respect to the explicit locking version as workload size increased.

6.2.6 Discussion

Our implementation obtains performance comparable to explicit locking while having much of the convenience of a mature STM implementation. The graph in Figure 5 shows the average percent slowdown over all of the benchmarks (excluding sqlite and oatomic) of the Autolocker, shelters, single global lock, reader/writer locks, and STM runs with respect to the locking runs versus the number of threads used. Our implementation scales up where possible as threads are

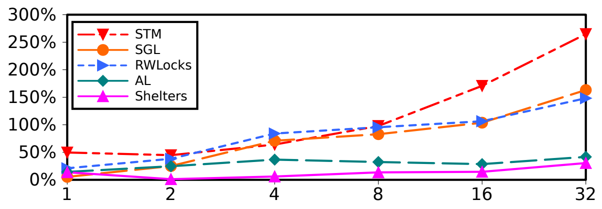

Figure 5. Average percent slowdown with respect to explicit locking over all benchmarks versus the number of threads. Lower is better.

added in many cases where the the other implementations fail to do so.

In particular, the Intel STM scales well when atomic statements are short or touch little shared memory, but suffers significant performance penalties when atomic sections are long and make many accesses to shared memory due to the cost of rollback and the cost of the instrumentation of shared memory accesses. Autolocker has performance similar to explicit locking and our implementation, however it is occasionally necessary to add locks by hand in order for it to accept a program. This could be avoided if it also used a hierarchy like our implementation does. However, we have shown that the performance of a hierarchy built out of locks would be poor (RWLocks). Furthermore, Autolocker fails to allow open-nesting. On the other hand, Autolocker requires a smaller annotation burden due to its use of whole program analysis. Finally, our implementation outperforms the use of a single global lock, and a naive implementation of shelters with a hierarchy of reader/writer locks, demonstrating that our techniques help achieve good performance in this space.

On the other hand, our implementation does not scale as well as explicit locking in the cases where contention is very low, but in which the ratio of synchronization calls to actual work is somewhat high. This overhead is exposed by the ebarnes benchmark, but is not an issue in the pbzip2 benchmark, for example.

7. Related Work

Many researchers have investigated the implementation of atomic statements, and more generally, language constructs enabling correct concurrency. Fortress [3], Chapel [12], and X10 [14] are recent languages intended to be used for writing highly scalable, high-performance code. Each of them includes atomic statements for protecting shared state.

The relative merits of optimistic and pessimistic concurrency control have been investigated by the database community. The consensus seems to be that optimistic approaches are desirable in the presence of abundant resources, so that the cost of roll-back and retry is not significant, whereas pessimistic approaches are desirable when resources are scarce [2].

A few projects are more closely related to our own. We have already discussed Autolocker in detail [25]. Cherem et al. [13] present a system in which the locks required for an atomic statement are inferred from the structure of expressions accessed in the atomic statement. The granularity of the locks may vary depending on the precision of the underlying analyses. Like Autolocker, this approach requires a whole program analysis for calculating pointer aliasing and for refining the expressions accessed in an atomic statement. The approach of Hicks et al. [19] is somewhat similar, requiring a whole program alias analysis to acquire a coarse-grained lock for the possible targets of a pointer when a precise target cannot be determined. The coarsening strategy for our system is similar, however instead of performing a whole-program alias analysis, we simply assume that pointers of the same type may alias. A more precise analysis could be integrated into our implementation to obtain a finer-grained shelter hierarchy, however, due to the practical problems it presents, we chose to avoid whole-program analysis.

In the Jade [33] programming language, programmers make annotations to describe how concurrent tasks will access shared state so that the Jade compiler can then automatically extract concurrency. We use the shelter mechanism to enforce atomic statements in an explicitly parallel program rather than as a way to help a compiler extract parallelism in an implicitly parallel language. Our system's implementation also expands on this mechanism by introducing explicit shelter objects that allow the programmer to declare what objects need protection, eliminating the need for the programmer to make annotations at each atomic block, and by introducing a shelter hierarchy.

There also exist several transactional memory systems that can be used to implement atomic statements, both hardware [4, 31] and software [16, 23, 36] based. We believe that the STM system for C most similar to our own in terms of programmer convenience is the Intel STM implementation [21]. Other STM implementations for C give better performance [11], but they require by-hand instrumentation of reads and writes of shared memory. The Intel STM implementation incurs overhead from the instrumentation of all shared memory reads and writes inside of transactions, and from the rollback of transactions during which conflicts are detected. Because our system is pessimistic, it does not incur these overheads. Boehm argues that transactional memory should be viewed as a mechanism for providing atomicity rather than a programming interface [9]. Some researchers have suggested that atomic statements with transactional semantics are for concurrency what garbage collection is for memory management [18]. The nesting semantics of transactions and atomic sections has been explored by both TM [1, 26, 30] and database [7] researchers. Open-nesting in a pessimistic setting is desirable because there is no need to write compensating actions, or to worry about complicated roll-back semantics.

Type systems with annotations describing locking rules have been used to prevent data races [15], in particular for Java [10]. Boyapati's ownership type-system [10] allows the expression and checking of sophisticated locking schemes. Our system could be extended with similar ownership or region types as an alternate way to arrive at a finer-grained shelter hierarchy. We leave these extensions for future work. Also for Java, Hindman and Grossman [20] translate programs with atomic statements to ones which acquire locks just before they are needed. Deadlock is avoided at runtime through rollback.

8. Conclusion

We have implemented a system in which atomic statements in C programs are implemented with *shelters*. We avoid the current problems associated with optimistic implementations of atomic statements by using a pessimistic approach, and unlike previous pessimistic approaches, our design allows us to avoid whole-program analysis, and provides for open-nesting. A wide range of benchmarks shows that our techniques perform well.

In the future, we plan to investigate the use of lightweight shape specifications that may allow our analysis to make use of a finer-grained shelter hierarchy to further improve usability and performance. Further, recording the differences between the timestamps obtained by threads, may provide an efficient method for deterministically replaying threaded programs. Additionally, we have not yet thoroughly investigated the fairness properties of our design. Fairness and liveness issues did not arise in any of our benchmark programs, but in the future we wish to obtain more rigorous guarantees.

9. Acknowledgements

We would like to thank the anonymous reviewers for their valuable feedback and suggestions, and our shepherd Michael Hicks for guidance in improving the clarity and organization of this paper.

References

[1] Agrawal, K., Leiserson, C. E., and Sukha, J. Memory models for open-nested transactions. In *MSPC'06*.

[2] Agrawal, R., Carey, M. J., and Livny, M. Concurrency control performance modeling: alternatives and implications. *ACM Trans. Database Syst. 12*, 4 (1987), 609–654.

[3] Allen, E., Chase, D., Luchangco, V., Jr., J.-W. M. S. R. G. L. S., and Tobin-Hochstadt, S. *The Fortress language specification version 1.0*, 2008. http://research.sun.com/projects/plrg/fortress.pdf.

[4] Ananian, C. S., Asanovic, K., Kuszmaul, B. C., Leiserson, C. E., and Lie, S. Unbounded transactional memory. In *HPCA'05*, pp. 316–327.

[5] Anderson, Z., Gay, D., Ennals, R., and Brewer, E. SharC: checking data sharing strategies for multithreaded C. In *PLDI'08*, pp. 149–158.

[6] Barnes, J., and Hut, P. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature 324* (Dec. 1986), 446–449.

[7] Beeri, C., Bernstein, P. A., and Goodman, N. A model for concurrency in nested transactions systems. *J. ACM 36*, 2 (1989), 230–269.

[8] Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[9] Boehm, H.-J. Transactional memory should be an implementation technique, not a programming interface. In *HotPar'09*.

[10] Boyapati, C. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT.

[11] Cao Minh, C., Chung, J., Kozyrakis, C., and Olukotun, K. STAMP: Stanford transactional applications for multi-processing. In *IISWC'08*.

[12] Chamberlain, B., Callahan, D., and Zima, H. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl. 21*, 3 (2007), 291–312.

[13] Cherem, S., Chilimbi, T., and Gulwani, S. Inferring locks for atomic sections. In *PLDI'08*.

[14] Ebcioglu, K., Saraswat, V., and Sarkar, V. X10: Programming for hierarchical parallelism and non-uniform data access. In *OOPSLA'04*.

[15] Flanagan, C., and Abadi, M. Object types against races. In *Conference on Concurrent Theory (CONCUR* (1999).

[16] Fraser, K., and Harris, T. Concurrent programming without locks. *ACM Trans. Comput. Syst. 25*, 2 (2007), 5.

[17] Goodman, J., Greenberg, A. G., Madras, N., and March, P. Stability of binary exponential backoff. *J. ACM 35*, 3 (1988), 579–602.

[18] Grossman, D. The transactional memory / garbage collection analogy. In *OOPSLA'07*, pp. 695–706.

[19] Hicks, M., Foster, J. S., and Pratikakis, P. Inferring locking for atomic sections. In *TRANSACT'06*.

[20] Hindman, B., and Grossman, D. Atomicity via source-to-source translation. In *MSPC'06*.

[21] Intel. *Intel C++ STM Compiler Prototype Edition 3.0*, 2008.

[22] kernel.org. Kernel bug tracker. http://bugzilla.kernel.org/.

[23] Marathe, V., Spear, M., Heriot, C., A.Acharya, Eisenstat, D., III, W. S., and Scott, M. Lowering the overhead of software transactional memory. In *TRANSACT'06*.

[24] Martin, J.-P., Hicks, M., Costa, M., Akritidis, P., and Castro, M. Dynamically checking ownership policies in concurrent C/C++ programs. In *POPL'10*, pp. 457–470.

[25] McCloskey, B., Zhou, F., Gay, D., and Brewer, E. Autolocker: synchronization inference for atomic sections. In *POPL'06*, pp. 346–358.

[26] Moss, J. E. B. Open nested transactions : Semantics and support. In *Workshop on Memory Performance Issues (WMPI)* (2006).

[27] mozilla.org. Bugzilla@mozilla bug tracker. http://bugzilla.mozilla.org/.

[28] Naik, M., and Aiken, A. Conditional must not aliasing for static race detection. In *PLDI'07*, pp. 327–338.

[29] Necula, G. C., McPeak, S., and Weimer, W. CIL: Intermediate language and tools for the analysis of C programs. In *CC'04*, pp. 213–228. http://cil.sourceforge.net/.

[30] Ni, Y., Menon, V. S., Adl-Tabatabai, A.-R., Hosking, A. L., Hudson, R. L., Moss, J. E. B., Saha, B., and Shpeisman, T. Open nesting in software transactional memory. In *PPoPP'07*, pp. 68–78.

[31] Rajwar, R., Herlihy, M., and Lai, K. Virtualizing transactional memory. In *ISCA'05*, pp. 494–505.

[32] Rinard, M. C., and Lam, M. S. Semantic foundations of Jade. In *POPL '92*, pp. 105–118.

[33] Rinard, M. C., and Lam, M. S. The design, implementation, and evaluation of jade. *ACM Trans. Program. Lang. Syst. 20*, 3 (1998), 483–545.

[34] Rossbach, C. J., Hofmann, O. S., and Witchel, E. Is transactional memory programming actually easier? In *PPoPP'10*, pp. 47–56.

[35] Saha, B., Adl-Tabatabai, A.-R., Hudson, R. L., Minh, C. C., and Hertzberg, B. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP'06*, pp. 187–197.

[36] Shavit, N., and Touitou, D. Software transactional memory. In *PODC'95*, pp. 204–213.

[37] Silberschatz, A., Galvin, P. B., and Gagne, G. *Operating System Concepts*, 6th ed. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[38] Woo, S. C., Ohara, M., Torrie, E., Shingh, J. P., and Gupta, A. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA'95*, pp. 24–36.

A. Proofs

Here, we prove Theorem 2 from Section 4. Additionally we argue that releasing shelters early, before the end of an atomic statement, or downgrading a shelter from one higher in the hierarchy to one lower in the hierarchy may have unexpected semantics. This implies that though it is an attractive seeming optimization, it must be weighed against the desired semantics for atomic statements.

Theorem. *Partial Atomicity.* In a trace

$$(t_1, s_1), \dots, (t_n, s_n)$$

that evaluates to M, R, H, a, let M_i, R_i, H_i, a_i be the state of M, H, R, a before the ith step of the trace. If

$$s_i = \text{register}(s_1, \dots, s_m)$$

and $s_j = \text{pop}$ is the end of this atomic statement then, for all variables v such that $\neg\text{regfor}(H_i, t_i, v) \wedge \text{regfor}(H_{i+1}, t_i, v)$ (the atomic statement gave access to v) the effects of the writes by thread t_i to v between s_i and s_j are visible to exactly those atomic statements of other threads that started after s_i **and** ended after s_j.

Proof: In the proof, we say "the atomic statement at i" to stand for "the atomic statement started by the statement s_i = register(...)",

Let $(a_i, s) \in H_{i+1}(t) - H_i(t)$ be such that $v_\sigma \leq s$. Consider the following cases for the atomic statement at k in threads $t \neq t_i$ that can access v:

1. $k < i$ and $\neg\text{regfor}(H_k, t, v) \wedge \text{regfor}(H_{k+1}, t, v)$ (the atomic statement gave access to v). Then $\exists(a_k, s') \in H_{k+1}(t) - H_k(t).v_\sigma \leq s'$, and, obviously, $a_k < a_i \wedge \text{interferes}(s', v_\sigma)$. Thus thread t_i cannot modify v until the atomic statement started at s_k terminates.
2. $k > i$, and the atomic statement ended at statement s_l with $l < j$, and $\neg\text{regfor}(H_k, t, v) \wedge \text{regfor}(H_{k+1}, t, v)$ (the atomic statement gave access to v). Then this atomic statement never accessed v, as $\forall k < d \leq l.\neg\text{access}(H_d, t_k, v)$.
3. $k > i$, and the atomic statement ended at statement s_l with $l > j$, and $\neg\text{regfor}(H_k, t, v) \wedge \text{regfor}(H_{k+1}, t, v)$ (the atomic statement gave access to v). Then $\forall(a, s') \in H_{k+1}(t) - H_k(t).a = a_k \wedge (v_\sigma \leq s' \Rightarrow$ $interferes(s', v_\sigma)$. As $a_k > a_i$, the atomic statement can only access v after the atomic statement at s_i terminates, so sees the effects of that atomic statement's writes.
4. $\text{regfor}(H_k, t, v)$ (an enclosing atomic statement gave access to v): consider the (unique) enclosing atomic statement at l that gave access to v:

   $$\neg\text{regfor}(H_l, t, v) \wedge \text{regfor}(H_{l+1}, t, v)$$

   If $l < i$, then the atomic statement at i cannot access v until the atomic statement started at l ends, which must necessarily be after the atomic statement at k ends. Therefore the effects of the writes by the atomic statement at i are not visible to the atomic statement started at k, as required.

   If $l > i$, the same arguments as above can be applied to show that either the atomic statement at k does not access v, or it terminates after the atomic statement at k.

Corollary 1. *Atomicity.* In the trace of a program that does not use open-nested transactions, the effects of an atomic statement are visible to exactly those atomic statements that started after it.

Proof: In the absence of nested atomic statements, the condition $\neg\text{regfor}(H_i, t_i, v) \wedge \text{regfor}(H_{i+1}, t_i, v)$ holds for all variables v accessed in the atomic statement at i.

All atomic statements at j with $j > i$ that access any variable v accessed by the atomic statement at i will necessarily end after the atomic statement at i ends, so will all see the effects of the writes of the atomic statement at i. Conversely, all statements that start at j with $j < i$ will not see the effects of the atomic statement at i.

Finally, atomic statements that start at j with $j > i$ and end before s_i end cannot access any variable accessed by the atomic statement at i or, by induction, depend on any atomic statement that accesses such a variable, so the effects of the atomic statement at i can be viewed as "visible" to the atomic at j.

The requirement that the effects of an atomic statement not be visible to atomic statements that end before it is necessary to prevent surprising behaviors where the effects of an incomplete atomic statement become visible to inner atomic statements of other threads. Consider a slightly modified trace language where pop is replaced by refine$(\sigma_1, \ldots, \sigma_m)$ which allows a thread to release shelters at any time or replace a coarser shelter by a finer one. The semantics for refine allow a thread to refine any shelters it is registered for:

$$\frac{\exists(a, \sigma) \in H(t).a_i = a \wedge \sigma_i \leq \sigma}{R, H \models (t, \text{refine}((a_1, \sigma_1), \ldots, (a_m, \sigma_m))}$$

$$\frac{H' = H[t \rightarrow \{(a_1, \sigma_1), \ldots, (a_m, \sigma_m)\}]}{M, R, H, a : (t, \text{refine}((a_1, \sigma_1), \ldots, (a_m, \sigma_m)) \rightarrow M, R, H', a}$$

In this modified trace language, a variant of Theorem 2 where the "**and** ended ..." clause is removed can easily be proved. However, the following trace has the surprising behavior that thread 2's atomic statement *must* complete after thread 1's atomic statement (it accesses state shared with thread 1's atomic statement and started after it), but the effects of thread 2's atomic statement are visible to thread 1's inner atomic statement *before* thread 2's atomic statement completes:

$(1, \text{reserve}(a_\sigma, b_\sigma)), (1, \text{register}(a_\sigma)), (1, \text{reserve}(b_\sigma)),$
$(2, \text{reserve}(a_\sigma, c_\sigma)), (2, \text{register}(a_\sigma, c_\sigma)),$
$(2, \text{reserve}()), (2, c = c + 1), (2, \text{refine}(a_\sigma)),$
$(3, \text{reserve}(b_\sigma, c_\sigma), (3, \text{register}(b_\sigma, c_\sigma)), (3, \text{reserve}()),$
$(3, b = b + c)), (3, \text{refine}()),$
$(1, \text{register}(b_\sigma)), (1, \text{reserve}()), (1, b = b + a), (1, \text{refine}()),$
$(2, a = 7), (2, \text{refine}())$

The write in thread 2 to c affects the write in thread 3 to b, and therefore the write in thread 1's inner atomic statement to b. While one could argue that this behavior is acceptable (thread 1's inner atomic statement does occur after thread 2's atomic statement), we believe it would be very counter-intuitive for programmers and should hence be forbidden.

In essence, allowing early shelter release exposes an atomic statement's effects "too early". It is however worth noting that in the absence of nested atomic statements, early shelter release or refinement is sound (i.e. preserves atomicity).

We attempted to prove that shelters could be understood as providing a mutual exclusion property similar to locks:

Definition 2. A trace is *lock equivalent* if after every step,

$$\forall t \neq t'.\forall(a, \sigma) \in H(t), (a', \sigma') \in H(t').\neg\text{interferes}(\sigma, \sigma')$$

i.e. two threads are never simultaneously registered for interfering shelters.

Conjecture 1. *Lock equivalence.* For every trace T which evaluates to M, R, H, a there exists a lock equivalent trace T' which evaluates to M, R, H, a (in the same context). Furthermore, for every thread t, $\text{ops}(t, T) = \text{ops}(t, T')$, where *ops (t, T)* is the subsequence of statements in trace T executed by thread t.

However, this conjecture is invalid, as the following variation on the example above shows:

$(1, \text{reserve}(a_\sigma, b_\sigma)), (1, \text{register}(a_\sigma)), (1, \text{reserve}(b_\sigma)),$
$(1, a = 10),$
$(2, \text{reserve}(a_\sigma, c_\sigma)), (2, \text{register}(a_\sigma)), (2, \text{register}(c_\sigma)),$
$(2, \text{reserve}()), (2, c = c + 1), (2, \text{pop}),$
$(3, \text{reserve}(b_\sigma, c_\sigma), (3, \text{register}(b_\sigma, c_\sigma)), (3, \text{reserve}()),$
$(3, b = b + c)), (3, \text{pop}),$
$(1, \text{register}(b_\sigma)), (1, \text{reserve}()), (1, b = b + a),$
$(1, \text{pop}), (1, \text{pop}),$
$(2, a = a + 1), (2, \text{pop})$

Here thread 2's inner atomic statement must happen after the start of thread 1's outer atomic statement starts and before the start of thread 1's inner atomic statement. It is thus not possible to reorganize this trace so that thread 1 and thread 2 are not simultaneously registered for interfering shelters. This trace also contradicts a weaker form of conjecture 1 which only requires non-interference at assignment statements (the trace element $(1, b = b + a)$ cannot be made to satisfy this weaker conjecture).