Atomic Shelters: Coping with Multi-core Fallout

Zachary Ryan Anderson
David Gay

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-39
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-39.html

April 9, 2010

Copyright © 2010, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Atomic Shelters: Coping with Multi-core Fallout

Zachary Anderson
University of California, Berkeley
zra@cs.berkeley.edu

David Gay
Intel Labs Berkeley
david.e.gay@intel.com

Abstract

In this paper we introduce a new method for pessimistically ensuring the atomicity of atomic sections. Similar to previous systems using locks, our system allows programmers to annotate the types of objects with the *shelters*—an alternative to locks inspired by the Jade programming language—that protect them, and indicate the sections of code to be executed atomically with atomic sections. A static analysis can then determine from which shelters protection is needed for the atomic sections to run atomically. Unlike previous systems, our shelter-based atomic sections are implemented such that they can provide atomicity and deadlock-freedom without the need for whole-program analyses or transactional memory, while imposing only a small annotation burden on the programmer and making only a straightforward change to the underlying type-system. We have implemented shelters-based atomic sections for C, and applied our implementation to 11 benchmarks totaling 100k lines of code including the STAMP benchmark suite, and three multithreaded programs. Our system's performance is competitive with both explicit locking and a mature software transactional memory implementation.

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 March 2010, according to their bugzilla databases, there were 62 known, outstanding race conditions in the Linux kernel, and 23 in Firefox; and there were 30 known, outstanding deadlocks in the Linux kernel, and 46 in Firefox [22, 28].

Atomic sections 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 statements in atomic sections become visible to other threads all at once when execution leaves the atomic section, much like a database transaction. Because they simply declare that code is to be run atomically, rather than fully specifying how code is to be made atomic, the use of atomic sections is less error prone than explicit locking. In particular, atomic sections make it much easier to compose independently written code because there is no confusion created by having different locking disciplines in different modules.

As multi-core processors have become more prevalent, newly developed languages have begun to include atomic sections instead of explicit locking [2, 10, 13]. Furthermore, attempts have been made to add atomic sections to preexisting languages such as C and C++ [11, 19, 27, 34]. Atomic sections may be implemented either optimistically, as in software transactional memory(STM) [36] systems, or pessimistically, as in Autolocker [27].

STM systems execute atomic sections optimistically, at least in part. Atomic sections are allowed to execute concurrently, but when two or more threads make conflicting accesses, transactions must be rolled-back and retried. Many STM implementations achieve good performance. However, if transactions are large, or if data is highly contended, rollbacks may be frequent and expensive. Furthermore, rollback may not be possible if, for example, any I/O was performed during a failed transaction.

The relative merits of optimistic and pessimistic concurrency control have been investigated by the database community. The consensus of this work seems to be that optimistic approaches are desirable in the presence of abundant resources, so that the cost of roll-back/replay is not significant, whereas pessimistic approaches are desirable when resources are scarce [1]. In the future, when resources may become abundant, production quality STM systems may overcome these practical difficulties. In the meantime, while resources are scarce, a pessimistic implementation of atomic sections can avoid these problems, while giving comparable performance in the cases where STM performance does scale.

In this paper, we focus on a pessimistic method for implementing atomic sections for C. Previous pessimistic approaches have relied on whole-program analysis for determining a global lock order [27], or for inferring fine-grained lock hierarchies [11, 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.

We avoid these problems by implementing atomic sections using *shelters*. Shelters are an alternative to locks inspired by the deadlock-free synchronization strategy used in the Jade programming language [33]. Like locks, shelters are first-class objects. Upon entry into an atomic section, threads must "register" for all of the shelters protecting the objects that are accessed in the atomic section. Registration is an atomic operation that places the thread on the queue of each shelter, and acquires a globally unique, increasing sequence number. Before accessing a sheltered object, a thread must then wait unless it is the thread on the shelter's queue having the smallest sequence number. When a thread exits an atomic section, it removes itself from the queues of the shelters for which it had registered. In effect, the sequence number picked at registration specifies the apparent serial order of the atomic sections. In Section 3 we prove that this mechanism guarantees atomicity and prevents deadlock.

Using our system, programmers write code with atomic sections and `sheltered_by(s)` annotations to specify which objects are protected by which shelters. We discover for which shelters registration is required through a backwards dataflow analysis over atomic sections. We then use the results of the analysis to translate the program with atomic sections into a program using the register and wait calls mentioned above. Where the analysis is imprecise, we make use of coarser-grained shelters based on the types of the objects in question. An overview of our system's operation is given in Section 2 along with an example.

We have implemented compiler transformations and a runtime for the C language, described in Section 4, and applied our implementation to 11 programs including the STAMP benchmark suite [8], and a few representative applications totalling over 100k lines of code. On average, the runtime overhead incurred by shelter-based atomic sections remains between 0 and 20% with respect to explicit locking for programs using between 1 and 32 threads on a 32 core machine. In Section 5, we present a thorough comparison of the runtime and programming cost of shelter-based atomic sections with the Intel STM implementation [21] 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 sections that requires no whole-program analysis.
- We present the results or our experiments comparing the runtime performance of shelter-based atomic sections with both explicit locking and an STM system. We also show that acquiring all locks at the entry to the atomic section, an obvious alternative to our system, has worse performance than shelters.
- We formalize our design to show that shelters enforce atomicity and prevent deadlock.

2. Overview

In this section we describe our extensions to C, and explain how these extensions are translated into calls into our system's runtime. We also present a small example that demonstrates many of the features of our system.

In addition to atomic sections, our system adds annotations to C for describing what objects require protection from concurrent access, and what the sheltering needs of functions are when called from inside of an atomic section. The resulting extensions to the C language are:

- `atomic {...}` — Atomic statements indicate that the effects of the statements on shared objects in the indicated block should not become visible to other threads until execution exits the block.
- `shelter_t` — Shelters are first-class objects in the language, and can be declared as variables, structure fields, or arguments to functions.
- `sheltered_by(s)` — This is an annotation on types indicating that an object of the annotated type can only be accessed in an atomic section (or a function called within an atomic section), and that concurrent access is mediated by the shelter `s`.
- `needs_shelters(s1,s2,...)` — This is an annotation on function types that contains a list of the C expressions of type `shelter_t` in terms of formal parameters and global variables. The list must contain expressions for shelters that protect objects read and written by the function. The list may be empty if no sheltered expressions are accessed. Only functions called from within atomic sections require an annotation.

Our system issues warnings and errors at compile-time as appropriate when required `needs_shelters` annotations are missing from functions, and when objects with a `sheltered_by` annotation are accessed outside of an atomic section or annotated function. The annotation burden imposed by our system, and a comparison to similar systems is investigated in Section 5.

Our system takes as input a program written using these extensions to C, and outputs a program that makes calls into a runtime implementing shelters-based atomic sections, which we then pass to an off-the-shelf C compiler for compilation and linking. This translation takes place in two steps. First, we perform a backwards dataflow analysis over atomic sections in order to collect the shelters that protect objects read and written in the atomic section. Where the analysis is imprecise, we use coarser-grained shelters as described in Section 2.2 below. In the second step, we use the results of the dataflow analysis to instrument the program with calls into our system's runtime.

Since the backwards analysis is straightforward, we only describe the runtime calls that are inserted by the analysis and instrumentation:

```
1 typdef struct {
2    int sheltered_by(s) id;
3    float sheltere_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;
10 }
11
12 needs_shelters(a->s)
13 void withdraw(account_t *a, float w) {
14   a->balance -= w;
15 }
16
17 needs_shelters(to->s, from->s)
18 void transfer(account_t *to, account_t *from,
19               float a) {
20   atomic {
21     withdraw(from, a);
22     deposit(to, a);
23   }
24 }
```

Figure 1. Code using atomic sections for the atomic transfer of funds between two accounts.

- `shelter_register(s,...)` — This call is added at the beginning of an atomic section. The arguments are the shelters that were found by the backwards analysis. In one atomic action, the call acquires one globally unique, increasing sequence number, and adds the calling thread to queues for each of the shelters in the argument list. Our implementation uses lock-free algorithms for acquiring the sequence number and adding the thread to the shelters' queues.
- `shelter_wait(s)` — This call is added when an object of a type annotated with `sheltered_by(s)` is accessed. The call causes the calling thread to wait until it is the thread with the smallest sequence number on `s`'s queue.
- `shelter_release_all()` — This call is added at the end of an atomic section. It causes the calling thread to remove itself from the queues of the shelters for which it had registered.

We support nested atomic sections by ignoring nested calls to `shelter_register` and `shelter_release_all`.

2.1 Example

We now show how shelter-based atomic sections can avoid the pitfalls of explicit locking while achieving the convenience of atomic sections 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` for protecting the `balance` and `id` fields, as indicated by the `sheltered_by(s)` annotations. 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)`. In the `transfer` function an `atomic` block indicates that the withdraw from the `from` account and the deposit to the `to` account happen atomically. 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 section, because it calls functions that access data protected by those shelters.

```
1 void deposit(account_t *a, float d) {
2   shelter_wait(a->s);
3   a->balance += d;
4 }
5
6 void withdraw(account_t *a, float w) {
7   shelter_wait(a->s);
8   a->balance -= w;
9 }
10
11 void transfer(account_t *to, account_t *from,
12               float a) {
13   shelter_register(to->s, from->s);
14   withdraw(from, a);
15   deposit(to, a);
16   shelter_release_all();
17 }
```

Figure 2. The results of the transformation of code with atomic sections and shelter annotations to code with shelter registrations, waits, and releases.

Figure 2 shows the results of transformation of the program with atomic sections and annotations to code that registers for shelters, waits on them before accessing the protected objects, and releases them when finished. The beginning of the atomic section in the `transfer` function is translated into the `shelter_register()` call on Line 13, which atomically acquires a unique sequence number and adds the calling thread to the queues for shelters `to->s`, and `from->s`. These shelters are collected by the backwards analysis from the annotations on the `deposit` and `withdraw` functions.

Where the program 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 sequence number on the queue for `s`. These calls appear for the accesses to the `balance` field of account structures on Lines 2 and 7 of Figure 2.

Where the atomic section ends in Figure 1, we now have a call to `shelter_release_all()` on Line 16, which removes the calling thread from the queues of the shelters for which it had registered.

Now, consider that one thread calls `transfer(A, B)`, and a second thread calls `transfer(B, A)` 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 sections, however, deadlock is avoided automatically because the two threads will have distinct sequence numbers, 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 sequence number proceeds while the thread with the larger sequence number must wait until the first thread calls `shelter_release_all()`.

2.2 Shelter Hierarchy

The backwards dataflow analysis cannot always statically infer precisely what shelters are needed on entry to an atomic section. We must therefore make use of a hierarchy of shelters, with coarser grained shelters used in response to imprecision in the analysis. 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 section. In this situation, our current implementation 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.

A static analysis having shape or ownership information may be able to obtain a finer-grained hierarchy. This is the approach suggested by McCloskey et al. [27], and implemented by Cherem et al. [11], and Hicks et al. [19], however these refinements are largely orthogonal to our contribution. Furthermore, our design goals include avoiding whole-program analysis or an intricate system of annotations, one of which approaches like these likely require.

In order to implement our hierarchy, when a thread registers for a shelter, it must also place itself on the queues of its ancestors in the hierarchy. Furthermore, when checking to see if it must wait for a shelter, a thread must also wait if a thread with a lower sequence number has registered directly for one of its ancestor shelters. For example, suppose thread T_1 is registered for a shelter protecting a particular `account_t` structure, `a->s`, and has sequence number 3. Further suppose that thread T_2 is in the atomic section in the alternate `transfer` implementation, and is registered directly for the ancestor of `a->s`, `account_t.s`, with sequence number 2. Even if T_1 is the thread with the smallest sequence number on the queue of shelter `a->s`, it must wait because T_2 has a smaller sequence number and is on the queue 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 `a'->s`, then both threads would be able to proceed.

We also provide syntax for adding a shelter higher up in the hierarchy to the list of shelters in the `needs_shelters` annotation.

2.3 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) {
  stmts;
}
```

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 statements `stmts` atomically. This is accomplished by our analysis treating the shelter wait block as an atomic section and collecting the shelters necessary for protecting both the block and the condition `e`. Then, the above construct can be translated as follows.

```
shelter_register(S);
shelter_wait(S);
while(!e) {
  ll_shelter_release_and_wait(S,scv);
  shelter_register(S);
  shelter_wait(S);
}
stmts;
shelter_release_all();
```

Here, `S` is the set of shelters found by the analysis, and `ll_shelter_release_and_wait(S,scv)` atomically releases

the shelters in S, and puts the thread to sleep waiting for a signal on scv. We leave as future work an extension to our system, like the one in Autolocker, that ensures that condition variables are signaled when appropriately specified state is updated.

2.4 Library Calls and Polymorphism

If a library call does not invoke any callbacks, 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 read and write 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 `shelter_wait` calls ahead of the library calls. Library calls invoking callbacks that access shelter protected state are not currently supported by our system.

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 5 due to the limited use of polymorphism in C programs. However, more modern languages may require increased support of polymorphism to support code-reuse, and other good software engineering practices. We leave support for polymorphism as future work.

2.5 Other Synchronization Strategies

It is not realistic to assume that all shared data will be protected by shelters and accessed within atomic sections. For instance, some shared data will be readonly 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. 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.

Three issues must be considered in the resulting programs:

- The programmer must ensure that data is shared correctly and using consistent mechanisms. Our own recent sharing checker work [4] uses sharing annotations on all types, and Martin et al. [26] use dynamic ownership assertions to detect where such rules are violated.
- The programmer must ensure that the mix of synchronization mechanisms does not cause deadlock. Such deadlocks are possible even when each mechanism is used safely: for instance, holding a lock when calling a barrier-synchronization function can cause deadlock if it prevents another thread from reaching its own barrier call. Similarly, calling a barrier-synchronization function inside a shelter-based atomic section will likely cause deadlock. Holding any lock when a `shelter_wait(s)` call occurs (i.e. when a sheltered object is accessed within an atomic section) is also likely to cause deadlock. However, this still allows many safe uses of locks with our system. Locks can be freely used outside atomic sections. Functions using locks can be freely called from within an atomic section as long as they release all acquired locks before returning and make no calls to shelter-using functions — this should be true of the typical library that uses locks to protect its internal state.
- Atomicity can be violated in some cases. For instance, atomic sections that access both lock-protected and shelter-protected data are not guaranteed to be atomic. However atomic sections that access only thread-private, readonly and shelter-protected data do remain atomic.

Trace	T	::=	$d_1, \ldots, d_n, (t_1, s_1), \ldots, (t_m, s_m)$
Statement	s	::=	$\text{atomic}_n(\sigma_1, \ldots, \sigma_m) \mid \text{end}$
		|	$v := n \mid v := v_1 + v_2$
Declaration	d	::=	int v sheltered by s
Shelter	σ	::=	$v_\sigma \mid s$
Identifiers	v, s	Integers	t, n, m

Figure 3. Traces of shelter-based programs.

3. Atomicity

We show that atomic sections implemented using shelters do indeed provide atomicity. That is, execution with shelters is equivalent to a sequential interleaving of atomically executed sections. Formally, we prove that a valid execution trace of a shelter-based program is equivalent to the corresponding trace where the atomic sections are executed atomically. Our approach is fairly different to that used for Jade [32] as we are proving a different property (atomicity vs. equivalence to a sequential program).

A trace T, defined in Figure 3, captures the essential aspects of execution in an imperative language with shelters. A trace starts with the declaration (d_i) of the global integer variables that are used in the trace. Each variable v is protected by its own shelter v_σ and a global shelter s (possibly shared with other variables), mirroring the hierarchical shelters in our system. The trace itself is a sequential interleaving of statements from multiple threads, where each thread is identified by a distinct integer t. The statements, executed atomically, are either the start of an atomic statement requiring shelters $\sigma_1, \ldots, \sigma_n$ (atomic), the end of an atomic statement, or assignments of an integer or computed value to a variable v.[1] An atomic statement has a sequence number n which must be greater than all atomic sequence numbers found earlier in the trace.

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

$$\frac{\text{access}(A(t), v, \Sigma) \qquad \text{access}(A(t), v_1, \Sigma) \qquad \text{access}(A(t), v_2, \Sigma)}{M, A, a, \Sigma : (t, v := v_1 + v_2) \to M[v \to M(v_1) + M(v_2)], A, a, \Sigma} \qquad \frac{A(t) = 0 \qquad n > a \qquad \Sigma' = \Sigma[\ldots, \sigma_i \to \Sigma(\sigma_i) \cup \{n\}, \ldots]}{M, A, a, \Sigma : (t, \text{atomic}_n(\sigma_1, \ldots, \sigma_m)) \to M, A[t \to n], n, \Sigma'}$$

$$\frac{\text{access}(A(t), v, \Sigma)}{M, A, a, \Sigma : (t, v := n) \to M[v \to n], A, a, \Sigma} \qquad \frac{A(t) \neq 0 \qquad \text{dom}(\Sigma') = \text{dom}(\Sigma) \qquad \forall \sigma \in \text{dom}(\Sigma).\Sigma'(\sigma) = \Sigma(\sigma) \setminus \{A(t)\}}{M, A, a, \Sigma : (t, \text{end}) \to M, A[t \to 0], a, \Sigma'}$$

$$\frac{M, A, a, \Sigma : (t_1, s_1) \to M', A', a', \Sigma'}{M, A, a, \Sigma : (t_1, s_1), \ldots, (t_m, s_m) \to M', A', a', \Sigma' : (t_2, s_2), \ldots, (t_m, s_m)}$$

$$\frac{\forall v.M(v) = 0 \qquad \forall \sigma.\Sigma(\sigma) = \emptyset \qquad \forall t.A(t) = 0 \qquad \forall t.A'(t) = 0 \qquad M, A, 0, \Sigma : (t_1, s_1), \ldots, (t_m, s_m) \xrightarrow{*} M', A', a', \Sigma' :}{d_1, \ldots, d_n, (t_1, s_1), \ldots, (t_m, s_m) \to M'}$$

$$\begin{aligned} \text{access}(n, v, \Sigma) \quad = \quad & (n \in (\Sigma(v_\sigma) \cup \Sigma(G(v)))) \wedge \\ & (\forall m \in \Sigma(v_\sigma) \cup \Sigma(G(v)).m \geq n) \end{aligned}$$

Figure 4. Trace Operational Semantics

Traces do not necessarily represent a valid execution of a shelter-based program. For instance,

int x sheltered by f, $(0, \text{atomic}_{42}(\text{g})), (0, \text{x} = 2), (0, \text{end})$

accesses x without holding either its individual shelter x_σ or its global shelter f. In Figure 4 we give an operational semantics for traces that computes a trace's effects as long as the trace is valid. The state of the operational semantics is a four-tuple M, A, a, Σ where $M : \text{id} \to \mathbb{N}$ maps variables to their values, $A : \mathbb{N} \to \mathbb{N}$ maps threads to their current atomic statement, $a : \mathbb{N}$ is the sequence number of the last initiated atomic statement and $\Sigma : \sigma \to \mathcal{P}(\mathbb{N})$ maps shelters to the set of active atomic statements that have requested that shelter. Finally, we write $G(v)$ to represent the global shelter specified in v's declaration.

The two assignment rules use the *access* function to check that thread t's current atomic statement $A(t)$ has either requested access to variable v's shelter v_σ or its global shelter $G(v)$, and that no atomic statement with an earlier sequence number currently has access to either of these two shelters. The rules then update the memory with the assignment's result. The atomic statement rule has three parts. First, the current thread must have ended any previous atomic statement ($A(t) = 0$). Second, the atomic statement's sequence number n must be greater than that of the previous (from any thread) atomic statement a. Finally, it computes Σ', the new state of shelter requests, by adding the atomic statement sequence number n to the requested shelter sets. Ending an atomic statement is symmetric: the thread must be executing an atomic statement ($A(t) \neq 0$), then it computes the new shelter state by removing the atomic statement's sequence number $A(t)$ from Σ and sets its atomic statement number to 0. The last two rules evaluate a complete trace from the initial state $M(v) = 0$ (all variables initialized to zero), $A(t) = 0$ and $a = 0$ (no active atomic statement), and $\Sigma(\sigma) = \emptyset$ (no shelters requested by any thread). Note that we also require that at the end of the trace all threads have ended their atomic statements ($A'(t) = 0$).

Definition 1. Trace $d_1, \ldots, d_n, (t_1, s_1), \ldots, (t_m, s_m)$ is *valid* with results M if $d_1, \ldots, d_n, (t_1, s_1), \ldots, (t_m, s_m) \to M$.

Definition 2. A trace is *serial* if it is of the form

$$\begin{aligned} & d_1, \ldots, d_n, \\ & (t^1, \text{atomic}_{n^1}(\ldots)), (t^1, v^1_1 := \ldots), \ldots, (t^1, v^1_{m^1} := \ldots), (t^1, \text{end}), \\ & \ldots \\ & (t^k, \text{atomic}_{n^k}(\ldots)), (t^k, v^k_1 := \ldots), \ldots, (t^k, v^k_{m^k} := \ldots), (t^k, \text{end}) \end{aligned}$$

Such traces execute the body of an atomic statement without any interference by other threads, so are obviously atomic.

Definition 3. *ops (t, T)* is the subsequence of statements in trace T executed by thread t.

Definition 4. *atomicorder (T)* is the subsequence of atomic statements of T.

Theorem 1. *Atomicity.* For every valid trace T with results M there exists a serial trace T' with results M. Furthermore, $\text{atomicorder}(T) = \text{atomicorder}(T')$ and for every thread t, $\text{ops}(t, T) = \text{ops}(t, T')$.

Less formally, the execution of T is equivalent to the clearly atomic execution obtained by moving all the assignments in each atomic section so that they occur immediately after the atomic statement.

Proof: The proof proceeds by a step-wise transformation of T into a serial trace T'. The detailed proof appears in Appendix A

4. Implementation

Our system is written in about 2200 lines of OCaml using the CIL [30] library, with a runtime library written in about 2500 lines of C. We use a combination of lock-free algorithms and other optimizations to ensure that our implementation scales.

```
 1 typedef struct {
 2   qentry_t Q[SIZE];
 3   int front, back, level;
 4   shelter_t *parent;
 5 } shelter_t;
 6
 7 typdef struct {
 8   uint64_t *thread;
 9   int level;
10 } qentry_t;
11
12 uint64_t global_counter;
13 __thread uint64_t T; // T is thread-private
14
15 void shelter_register(shelter_t *s) {
16 retry:
17   uint64_t old = global_counter;
18   addToQueues(s, &T);
19   T = old + 1;
20   if (CAS(&global_counter, old, T) != old) {
21     T = 0;
22     rmFromQueues(s, &T);
23     goto retry;
24   }
25 }
26
27 void shelter_wait(shelter_t *s, int level) {
28   int wait, front;
29   do {
30     wait = 0; front = s->front;
31     while (front != s->back) {
32       if (Q[front].thread == &T) break;
33       else if (*Q[front].thread &&
34                level == s->level &&
35                *Q[front].thread < T) {
36         wait = 1; break;
37       }
38       else if (*Q[front].thread &&
39                s->level < level &&
40                Q[front].level < level &&
41                *Q[front].thread < T) {
42         wait = 1; break;
43       }
44       front = (front + 1) % SIZE;
45     }
46   } while (wait);
47   if (s->parent) shelter_wait(s->parent, level);
48 }
```

Figure 5. Psuedo-code for the shelter wait functions.

4.1 Shelter Registration and Waiting

In order to avoid threads being serialized in acquiring a unique sequence number and adding themselves to shelter queues, we use lock-free algorithms for the `shelter_register` and `shelter_wait` operations. Figure 5 gives sketches for these functions. The intention is to illustrate the key features of the algorithms rather than to show a complete implementation.

We first define two structures, one for the shelters themselves, and the other for the entries in the shelters' queues. The `shelter_t` type includes the shelter's queue, `Q`; indexes for the front and back of the queue, `front` and `back`; the shelter's level in the hierarchy, `level`, where a smaller number indicates that the shelter is higher in the hierarchy; and a pointer to the shelter's parent in the hierarchy, `parent`. The entries in the queue, with type `qentry_t`, hold pointers to the sequence numbers of the registered threads in the `thread` field, and the levels of the shelters that those threads have initially registered for in the `level` field.

Two global variables keep track of the global sequence number counter, `global_counter`, and the thread-private sequence number, `T`. We use 64-bit unsigned integers so that overflow is unlikely.

The goal of shelter registration is to atomically acquire a unique sequence number, and place the calling thread in the right place on the indicated shelters' queues. The thread must be added to all shelter queues atomically in order to avoid deadlock. The function works as follows. First the global sequence number is read (Line 17). Then the thread adds itself to the queues for the indicated shelter and its ancestors (Line 18). We omit the pseudo-code for the queue because it is a standard lock-free queue implemented with a circular buffer. Next, the thread's sequence number is assigned to the next available sequence number (Line 19) before an attempt is made to increment the global sequence number counter with an atomic compare-and-swap (Line 20). If that fails, the thread zeroes out its sequence number (Line 21), and removes itself from the shelters' queues (Line 22) before retrying (Line 23).

The global sequence number is read before adding the thread to the shelter queues so that when the compare-and-swap succeeds, we are guaranteed that any unsorted-ness ahead of the thread in the queue will be only temporary. It is also the case that entries in the queue may temporarily be zero (that is, before assigning `T` after adding a thread to the queue, or after zeroing it when the compare-and-swap fails.) It is safe for the `shelter_wait` function to ignore these zeroed entries.

The shelter wait function works by checking to see if it must wait for a shelter by inspecting its thread queue, and then making a recursive call to see if it must wait for any of that shelter's ancestors. It takes two arguments, the shelter currently being inspected, and the level in the hierarchy of the shelter that the thread initially called `shelter_wait` on. The call mentioned in Section 2, which only took the shelter argument, is a simple wrapper for this call.

First, the thread will look through the shelter's queue, either for its own entry or for an entry that means it must wait. This loop begins on Line 31. In the loop, the thread checks to see if it has found its own entry (Line 32). Since

a thread can never see its own entry out of order in the queue (shelter registration only completes when an in-order enqueue succeeds), when a thread finds its own entry, it does not need to wait. If a thread has not yet found itself, and if the the shelter currently being inspected is the one the thread initially called `shelter_wait` on, then the thread must wait if it finds an entry in the queue that has a smaller sequence number (Line 36). If the shelter currently being inspected is an ancestor of the shelter that the thread initially waited on, then the thread must only wait if it finds a queue entry for a thread that initially waited on a shelter higher in the hierarchy that *also* has a smaller sequence number (Line 42). On Line 47, the thread checks to see if it must wait for a parent shelter by making a recursive call to `shelter_wait`.

In this pseudo-code, when a thread discovers that it must wait, it simply retries the check. However, it is straightforward to implement other options, such as sleeping, or waiting on a traditional condition variable. In our experiments, we simply retry the check again immediately after an invocation of the scheduler, however in the future we intend to experiment with the futex [12] system calls provided by Linux.

4.2 Optimizations

In practice, a number of optimizations are required for our system to scale. In particular, the following optimizations were critical to achieving performance similar to, and in some cases, better than explicit locking.

4.2.1 Important Optimizations

It is often the case that a shared object is only ever read in a atomic section. Our static analysis is able to determine when a shelter is only required by an atomic section for read access. This has been omitted from the runtime description above, however in these cases, a thread may indicate that it will only ever read objects protected by a shelter when registering for it. Then, when determining whether or not it should wait before reading a sheltered object, a thread must only have a sequence number 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 section implementation invokes it automatically wherever possible.

It is also often the case that atomic sections 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.

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 `shelter_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, we unset the inactive flag in the parent shelter, and proceed.[2]

As the number of threads and cores increases, contention for the global sequence number counter increases. To address this, when a compare-and-swap operation on the counter fails, after retrying immediately a small number of times, we use a binary exponential backoff algorithm [18]. In our experiments, this approach reduced by several orders of magnitude the number of compare-and-swap failures without compromising performance.

4.2.2 Other Optimizations

We also implemented a few obvious optimizations that we believe to be important in general, but for which we observed no advantage in our experiments.

In particular, from the results of our backwards analysis, we can deduce a few interesting facts. First, we find places where it is safe to release a shelter before the end of an atomic section. Similarly, we find places where it is safe to downgrade a thread from write access to read access. Finally, we find places where it is safe to give up a parent shelter, and to instead acquire one of its children. These optimizations were left enabled for the results we present in Section 5, but because the atomic sections in our benchmark programs are placed very carefully, they had little effect on performance. In the future, we plan to investigate microbenchmarks that demonstrate in what situations these optimizations are most useful.

4.2.3 Proposed Optimizations

Another potential optimization that we leave for future work involves the way that sequence numbers are allocated to threads. If it can be determined that two threads will never use the same shelter, then their sequence numbers need not be distinct. An analysis supporting such an optimization would likely rely on some form of a must-not-alias analysis [29].

[2] 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.

5. Evaluation

We have modified a number of programs to use shelters as the mechanism for enforcing atomic sections, and we have measured the runtime performance of these programs on typical inputs. The purpose of this evaluation is to investigate the convenience of using shelter-based atomic sections, and to compare the runtime performance of our implementation with four other mechanisms for enforcing atomicity, namely explicit locking, software transactional memory, a single global lock, and shelters implemented with pthread reader-writer locks.

In the implementation of shelters using reader-writer locks, each shelter contains a lock. When registering for shelters at the beginning of atomic sections, 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.

5.1 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 2.6.18. We chose to compare against an Intel compiler for C/C++ that includes an STM implementation [34]. Other STM implementations may give better performance [8], but the Intel STM compiler has an annotation burden that is similar to that of our system, and requires no additional special hardware. We also used the Intel compiler with the STM features disabled as the back-end of our system, and to compile the other versions of the benchmarks. The compile-time analyses used for our system did not add significantly to compilation time.

Occasionally, slight modifications were made to programs to avoid activation of higher levels of the shelter hierarchy where possible. We believe this practice is consistent with how a working programmer would improve the performance of a program using atomic sections, and the STM and explicit locking versions also benefited from these modifications because transactions were smaller, and locks were held for shorter periods of time, respectively. In the explicit locking versions, the locking was made as fine-grained as possible without substantially rewriting the programs.

5.2 Benchmarks

Our benchmark programs consist of the STAMP STM benchmark suite [8], along with: pbzip2, a parallel version of bzip2; pfscan, a parallel file scanning tool; and ebarnes, an n-body simulation using the Barnes-Hut algorithm [5]. Table 1 shows the size of each of the benchmark programs, the number of atomic sections in each, and the number of other annotations that were needed to use Intel's STM and our system, 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. The STAMP benchmarks are distributed with the `tm_callable` annotations already placed, some of which are redundant. We also made redundant annotations when adapting the programs for our system, 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 our system.

Name	Size (kloc)	Atm. sects.	STM Annts.	Shelter Annts.	Seq. Time
bayes	12.0	15	47	42	9.97s
genome	10.0	5	16	25	8.58s
intruder	11.3	3	61	64	2.26s
kmeans	3.9	3	4	7	9.17s
labyrinth	8.2	3	50	46	3.02s
ssca2	9.2	1	5	12	9.73s
vacation	11.0	3	159	122	1.53s
yada	13.4	6	105	86	4.19s
ebarnes	13.4	3	8	9	16.07s
pbzip2	10.0	10	4	14	10.46s
pfscan	2.8	6	4	11	2.56s
total	105.2	48	463	438	

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

The results of our experiments are given in the graphs of Figure 6. 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 at least 50 runs. Command line arguments passed to the STAMP benchmarks are also given in the captions.

5.2.1 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 sections.

The bayes benchmark implements an algorithm used in learning Bayesian networks. It involves multiple threads concurrently modifying and searching in a graph. The bayes benchmark required activation of shelters higher in the hierarchy for the linked lists used to represent the graph adjacency list. The performance of the bayes benchmark is shown in Figure 6(a). None of the implementations were able to yield any significant parallel speedup. For our system and explicit locking, lock and shelter contention is high because exclusive access to the entire graph is acquired for

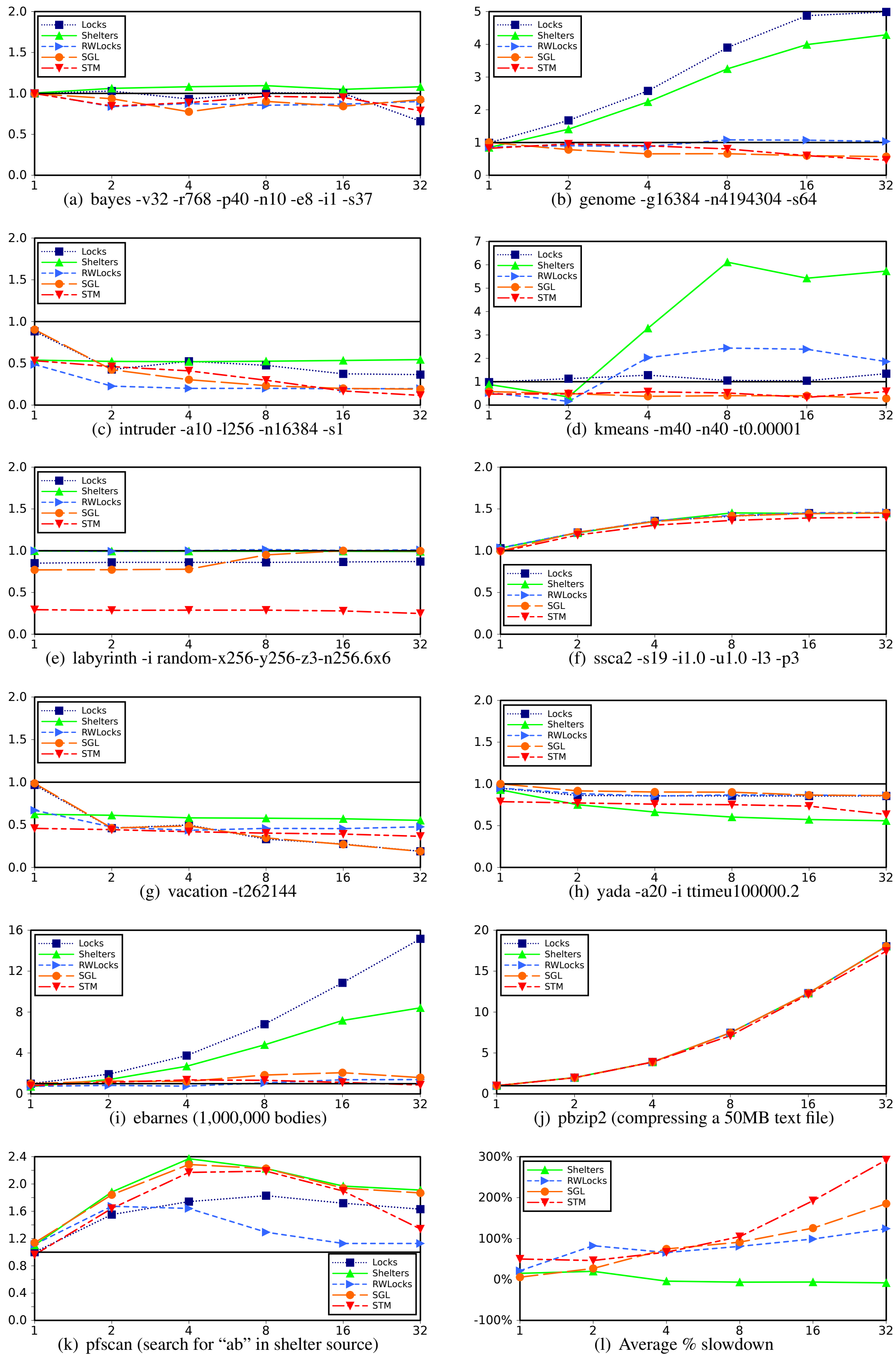

Figure 6. Graphs (a) - (k) show speedup over sequential runs versus the number of threads used for our benchmark programs when run with explicit locking (Locks), shelters (Shelters), Intel STM (STM), a single global lock (SGL), and shelters implemented with reader-writer locks (RWLocks). Higher is better. Graph (l) shows average percent slowdown with respect to explicit locking over all benchmarks versus the number of threads. Lower is better.

each atomic section. For the STM runs, data contention and big transactions caused many conflicts and rollbacks, limiting scaling.

The genome benchmark reconstructs a gene sequence from overlapping fragments. It involves multiple threads concurrently adding elements to hashtables, and searching in and modifying strings. The performance of the genome benchmark is shown in Figure 6(b). The explicit locking and shelters versions scale up as threads are added, but the STM version does not due to the overhead of memory barriers for memory reads and writes in atomic sections [9].

The intruder benchmark implements a signature-based intrusion detection algorithm. Packet streams are concurrently collected in a red-black tree, added to a queue when finished, and examined for a matching signature. The performance of the intruder benchmark is shown in Figure 6(c). None of the implementations achieve a parallel speedup due to being serialized by access to the red-black tree.

The kmeans benchmark implements a clustering algorithm. Threads concurrently read and modify several arrays used to calculate the means of, and membership in, the clusters.The performance of the kmeans benchmark is shown in Figure 6(d). Shelters and shelters implemented with reader-writer locks are able to achieve some parallel speedup. The STM implementation is penalized by the necessity of instrumenting memory reads and writes in transactions as for the genome benchmark. Explicit locks are penalized by the inability to provide concurrent read access, however this could be fixed by using reader-writer locks.

The labyrinth benchmark implements an algorithm for navigating a maze. Threads concurrently update a two-dimensional array to add to it paths through the maze. The performance of the labyrinth benchmark is shown in Figure 6(e). None of the implementations are able to achieve a parallel speedup due to being serialized by access to the maze array. The STM version is again penalized by instrumentation of reads and writes.

The ssca2 benchmark implements a graph kernel used in a number of different algorithms. Threads concurrently update graph adjacency lists. The performance of this benchmark is shown in Figure 6(f). Each implementation is able to achieve a modest parallel speedup.

The vacation benchmark implements a travel reservation system. Threads concurrently access and modify relations stored as maps represented by red-black trees. The vacation benchmark required activation of the higher levels in the shelter hierarchy for linked lists stored in the maps. The performance of this benchmark is shown in Figure 6(g). None of the implementations are able to achieve any parallel speedup due to being serialized by access to the red-black trees.

The yada benchmark implements a mesh refinement algorithm. Threads concurrently add and remove nodes and edges from the mesh. As with the bayes benchmark, the yada benchmark required activation of higher levels in the shelter hierarchy for graph adjacency lists. The performance of this benchmark is shown in Figure 6(h). None of the implementations are able to achieve a parallel speedup. Even though the mesh is large and triangles in need of refinement begin distributed uniformly throughout the mesh, as the algorithm runs, the location of triangles in need of refinement becomes correlated. This increases both data and lock contention, which prevents scaling [23]. The performance of our system implementation degrades a bit less gracefully than the other implementations.

5.2.2 Application Benchmarks

In addition to the STAMP benchmarks, we chose two multithreaded C applications using explicit locking to protect shared objects. We chose these applications in order to compare the real-world performance of shelters with the other implementations. We also included a benchmark implementing an n-body simulation using the Barnes-Hut algorithm. We chose it because the parallel speedup in the oct-tree-building phase is very sensitive to the synchronization strategy.

The ebarnes benchmark is an n-body simulation adapted from the Barnes-Hut Splash2 benchmark [37]. In each phase of the simulation it builds an oct-tree in parallel and uses it to update the positions of the bodies. For this benchmark, 1 million bodies were simulated so that building the oct-tree in parallel would give a significant performance advantage. The performance of this benchmark is shown in Figure 6(i). Shelters and explicit locks allow scaling up as cores are added, whereas the other implementations fail to allow scaling.

The pbzip2 benchmark is a parallel implementation of the popular block-based compression algorithm. Threads take blocks to compress off of a shared queue. Separate threads add blocks to the queue, and write the compressed blocks to a file. For this benchmark, a 50MB text file was compressed. The file was small enough to fit into the operating system's file system caches. The performance of this benchmark is shown in Figure 6(j). Each implementation manages to obtain a parallel speedup.

The pfscan benchmark is a tool that searches for a string in all files under a given directory tree. One thread places paths to files to search on a queue while other threads takes paths off the queue and search for the string in the files. For this benchmark, we searched for the string "ab" in the Linux source code tree. The tree was small enough to fit into the operating system's file system caches. The performance of this benchmark is shown in Figure 6(k). Each implementation manages to obtain a parallel speedup until calls into the operating system and limited workload size impede further performance gains.

5.2.3 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 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.

5.2.4 Discussion

Our system's implementation manages to obtain performance comparable to explicit locking while having much of the convenience of a mature STM implementation. The graph in Figure 6(l) shows the average percent slowdown over all of the benchmarks of the shelters, single global lock, reader/writer locks, and STM runs with respect to the locking runs versus the number of threads used. Our system's implementation scales up where possible as threads are added in cases where the the other implementations fail to do so.

6. Related Work

Many researchers have investigated the implementation of atomic sections, and more generally, language constructs enabling correct concurrency. Fortress [2], Chapel [10], and X10 [13] are new languages intended to be used for writing highly scalable, high-performance code. Each of them includes atomic sections for protecting shared state.

A few projects are more closely related to our own. Autolocker allows programmers to annotate variables and fields as being protected by a particular lock [27]. Then, for each atomic section, it determines which locks must be acquired, and in what order they must be acquired. Determining lock order requires a whole-program analysis, which is often impractical for large projects written in C. Our system avoids the need to find a global lock order, and therefore allows C projects to retain separate compilation.

Cherem et al. [11] present a system in which the locks required for an atomic section are inferred from the structure of expressions accessed in the atomic section. 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 section. 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. Access to the shared state is then mediated with a mechanism that inspired our implementation of shelter-based atomic sections. In particular, shared objects each maintain a queue of tasks waiting to access them. When many tasks attempt to access the same object, the task with the smallest sequence number is permitted to proceed. We use the shelter mechanism to enforce atomic sections 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. Further, our lock-free implementation reduces the extent to which programs are serialized by accesses to queues.

There also exist several transactional memory systems that can be used to implement atomic sections, both hardware [3, 31] and software [35, 17, 25] 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 [8], 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 [6].

Type systems with annotations describing locking rules have been used to prevent data races [14, 15], in particular for Java [16, 7]. Boyapati's ownership type-system [7] 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 sections to ones which acquire locks just before they are needed. Deadlock is avoided at runtime through rollback.

7. Conclusion

We have presented a system in which atomic sections in C programs are implemented with *shelters*. We avoid the current problems associated with optimistic implementations of atomic sections by using a pessimistic approach, and unlike previous pessimistic approaches, our system's design allows us to avoid whole-program analysis. We used shelter-based

atomic sections in 11 benchmarks including the STAMP benchmark suite, and three interesting programs. We observed that the performance of our system is competitive with both explicit locking and the Intel STM system.

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. Also, as the number of cores per socket increases, we will be able to do a more thorough investigation of the ability of our system to scale beyond the 32 cores used in the experiments presented here.

Additionally, we have not yet thoroughly investigated the fairness properties of our system. Fairness and liveness issues did not arise in any of our benchmark programs, but in the future we wish to obtain more rigorous guarantees.

A. Atomicity

THEOREM 1. *Atomicity.* For every valid trace T with results M there exists a serial trace T' with results M. Furthermore, atomicorder(T) = atomicorder(T') and for every thread t, $\mathrm{ops}(t, T) = \mathrm{ops}(t, T')$.

Proof: The proof proceeds by a step-wise transformation of T into a serial trace T'. Any non-serial trace T has a (possibly empty) prefix that matches a serial trace, i.e. T is of the form:

$$(t^1, \mathrm{atomic}_{n^1}(\ldots)), (t^1, v^1_1 := \ldots), \ldots, (t^1, v^1_{m^1} := \ldots), (t^1, \mathrm{end})$$
$$\ldots$$
$$(t^k, \mathrm{atomic}_{n^k}(\ldots)), (t^k, v^k_1 := \ldots), \ldots, (t^k, v^k_{m^k} := \ldots), (t^k, \mathrm{end}),$$
$$(t_1, s_1), \ldots, (t_n, s_n)$$

Furthermore s_1 must be an atomic statement, as assignments and end are not valid when no atomic statement is in progress. Therefore $(t_1, s_1), \ldots, (t_n, s_n)$ must match the following template:

$$(t, \mathrm{atomic}_{n_t}(\ldots), (t, v_1 := \ldots), \ldots, (t, v_{m_t} := \ldots),$$
$$(t', s'), \ldots, (t'', s''), (t, s), \ldots$$

with $n_t > n^k$, $t' \neq t$ and (t, s) the earliest statement of thread t after $v_{m_t} := \ldots$: there must be at least one such statement as a valid trace requires $A(t) = 0$ at termination so thread t must have at least one more end statement. Note also that s cannot be an atomic statement.

We show that the trace T'' produced by moving (t, s) one step left, i.e. swapping it with (t'', s'') is a valid trace with the same results M, atomicorder(T) = atomicorder$)(T'')$ and $\forall t.\mathrm{ops}(t, T) = \mathrm{ops}(t, T'')$. Repeated applications of this transformation clearly terminate in the desired serial trace T': we eventually move (t, s) until it is adjacent to $(t, v_{m_t} := \ldots)$. At that point we will start moving another statement left, either for thread t if $s \neq$ end or for some new statement for thread t'. We note that this approach is very similar, but not quite identical to Lipton's theory of reduction [24]: while in our particular case $(t, v := \ldots)$ can be moved left past (t'', end) (see below), in general assignments cannot be moved past end statements of other threads, i.e. are not general left-movers.

We show that moving (t, s) left produces T'' with the desired properties for the six possible combinations of statements s'' and s. We start by noting that as s is not an atomic statement and $t'' \neq t$ that atomicorder(T) = atomicorder$)(T'')$ and $\forall t.\mathrm{ops}(t, T) = \mathrm{ops}(t, T'')$. For each case we only need to show that the trace T'' is valid and has results M. We denote the operational state of T before (t'', s'') by M_0, A_0, a_0, Σ_0, the state after (t'', s'') by M_1, A_1, a_1, Σ_1 and the state after (t, s) by M_2, A_2, a_2, Σ_2. The state in T'' before (t, s) is also M_0, A_0, a_0, Σ_0, after (t, s) it is $M''_1, A''_1, a''_1, \Sigma''_1$ and the state after (t'', s'') it is $M''_2, A''_2, a''_2, \Sigma''_2$. We simply need to show that $(M''_2, A''_2, a''_2, \Sigma''_2) = (M_2, A_2, a_2, \Sigma_2)$.

First, we consider the case where s is end. If s'' is end or s'' is atomic the result follows from the obvious lemma that $u \neq u' \wedge A(u) \neq 0 \Rightarrow A(u) \neq A(u')$ applied to $t \neq t''$. The case where s'' is $v'' := \ldots$ also holds, as if $A_0(t'')$ is the smallest value in $\Sigma_0(v''_\sigma) \cup \Sigma_0(G(v''))$ it is also clearly the smallest value in $\Sigma''_1(v''_\sigma) \cup \Sigma''_1(G(v''))$ (the same argument applies to any other variable in the assignment).

Next, we consider the case where s is $v := \ldots$. We know that $A_0(t) = A_1(t) = n_t$. If s'' is atomic$_p(\ldots)$, then $p > n_t$ so $\mathrm{access}(A_1(t), v, \Sigma_1) \Rightarrow \mathrm{access}(A_0(t), v, \Sigma_0)$ so the result clearly holds. If s'' is $v'' := \ldots$ and the two statements access non-overlapping variables, the result is trivial. The case where they access overlapping variables cannot occur as T would not be a valid trace: $A_0 = A_1$ and $\Sigma_0 = \Sigma_1$, so if $\mathrm{access}(A_1(t), v, \Sigma_1)$ then $\neg\mathrm{access}(A_0(t''), v, \Sigma_0)$, or vice-versa. If s'' is end we note that $A_0(t'') > A_0(t) = A_1(t)$ as t'' must have started its atomic statement after t (all atomic statements prior to t's have already completed). Thus $\mathrm{access}(A_1(t), v, \Sigma_1) \Rightarrow \mathrm{access}(A_0(t), v, \Sigma_0)$ so the result clearly holds.

References

[1] AGRAWAL, R., CAREY, M. J., AND LIVNY, M. Concurrency control performance modeling: alternatives and implications. *ACM Trans. Database Syst. 12*, 4 (1987), 609–654.

[2] 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.

[3] ANANIAN, C. S., ASANOVIC, K., KUSZMAUL, B. C., LEISERSON, C. E., AND LIE, S. Unbounded transactional memory. In *HPCA'05*, pp. 316–327.

[4] ANDERSON, Z., GAY, D., ENNALS, R., AND BREWER, E. SharC: checking data sharing strategies for multithreaded C. In *PLDI'08*, pp. 149–158.

[5] BARNES, J., AND HUT, P. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature 324* (Dec. 1986), 446–449.

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

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

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

[9] Cascaval, C., Blundell, C., Michael, M., Cain, H. W., Wu, P., Chiras, S., and Chatterjee, S. Software transactional memory: why is it only a research toy? *Commun. ACM 51*, 11 (2008), 40–46.

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

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

[12] Drepper, U. Futexes are tricky, 2009. http://people.redhat.com/drepper/futex.pdf.

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

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

[15] Flanagan, C., and Abadi, M. Types for safe locking. In *ESOP'99* (1999).

[16] Flanagan, C., and Freund, S. N. Type-based race detection for Java. In *PLDI'00*, pp. 219–232.

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

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

[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] Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., and Chew, L. P. Optimistic parallelism requires abstractions. In *PLDI'07*, pp. 211–222.

[24] Lipton, R. J. Reduction: a method of proving properties of parallel programs. *Commun. ACM 18*, 12 (1975), 717–721.

[25] 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*.

[26] Martin, J.-P., Hicks, M., Costa, M., Akritidis, P., and Castro, M. Dynamically checking ownership policies in concurrent C/C++ programs. In *POPL'10*. Full version, preprint.

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

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

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

[30] 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/.

[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] 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.

[35] Shavit, N., and Touitou, D. Software transactional memory. In *PODC'95*.

[36] Shavit, N., and Touitou, D. Software transactional memory.

[37] 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.