A Runtime System for Software Lock Elision

Amitabha Roy
University of Cambridge
amitabha.roy@cl.cam.ac.uk

Steven Hand
University of Cambridge
steven.hand@cl.cam.ac.uk

Tim Harris
Microsoft Research, Cambridge
tharris@microsoft.com

Abstract

The advent of multi-core processors means that exploiting parallelism is key to increasing the performance of programs. Many researchers have studied the use of `atomic` blocks as a way to simplify the construction of scalable parallel programs. However, there is a large body of existing lock-based code, and typically it is incorrect to simply replace lock-based critical sections with `atomic` blocks. Some problems include the need to do IO within critical sections; the use of primitives such as condition variables; and the sometime reliance on underlying lock properties such as fairness or priority inheritance.

In this paper we investigate an alternative: a software runtime system that allows threads to speculatively execute lock-based critical sections in parallel. Execution proceeds optimistically, dynamically detecting conflicts between accesses by concurrent threads. However, if there are frequent conflicts, or if there are attempts to perform operations that cannot be done speculatively, then execution can fall back to acquiring a lock. Conversely, implementations of `atomic` blocks must typically serialise all operations that cannot be performed speculatively.

Our runtime system has been designed with the requirements of systems code in mind: in particular it does not require that programs be written in type-safe languages, nor does it require any form of garbage collection. Furthermore, we never require a thread holding a lock to wait for a thread that has speculatively acquired it. This lets us retain any useful underlying properties of a given lock implementation, e.g. fairness or priority inheritance.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques Concurrent Programming

General Terms Design, Algorithms, Performance

EuroSys'09, April 1–3, 2009, Nuremberg, Germany.

1. Introduction

In the multi-core era, processor evolution is characterised by increasing numbers of cores from one year to the next, rather than faster uni-processor clock speeds. Software must therefore be structured to make good use of these cores if it is to continue to improve in performance on new hardware.

However, there is a large existing body of code that has been written in languages such as C/C++ using abstractions such as mutexes, condition variables, and barriers. There is also, of course, a large number of programmers with experience in using these language features.

We are investigating the extent to which we can improve the performance of these existing workloads and language features on multi-core hardware. We are aiming to develop new implementations of existing features that will (*a*) provide better scalability for the applications that use them, and (*b*) encourage the use of simple programming idioms so that programs are more obviously correct – free from deadlocks, race conditions, and so on.

In this paper we focus on the implementation of lock-based critical sections. We introduce a runtime system that allows multiple threads to execute critical sections via software-implemented *speculative lock elision* [Rajwar 2001] (SLE). We introduce the SLE programming model in Section 2. The idea is that multiple threads can execute critical sections protected by the same lock, so long as there are no data conflicts between the threads. If there are frequent data conflicts, or if a thread attempts an operation that cannot be done speculatively, then the implementation falls back to a non-speculative code path. SLE can improve scalability for workloads where lock contention is high, but actual data conflicts are rare. So long as data conflicts are unlikely, SLE can let the programmer develop scalable shared-memory data structures using coarse-grained locking, rather than needing to use complicated fine-grained locking schemes.

We introduce a manual interface for using our SLE runtime system in Section 3, and we describe the implementation of the runtime system in Section 4. Supporting SLE introduces a number of challenges, primarily because of the need to control the three-way interaction between speculative use of a lock, non-speculative use of the same lock, and ordinary code in the application. We make a number of design choices which guide many implementation decisions:

- First, we are careful to ensure that any non-speculative thread is not delayed by a speculative one. For example, if a thread attempts to acquire a lock non-speculatively, then we do not want it to have to wait for a thread that has speculatively acquired the lock. This design choice mitigates the negative effect that speculation can have in workloads where it is not effective, reminiscent of design choices in systems such as CILK [Blumofe 1995].
- Second, we wish to avoid any changes to the implementation of code outside critical sections. This is necessary to make SLE transparent, for example to allow the use of ordinary application-specific data formats, memory management schemes, and so on.

In ongoing work – and along with many other researchers – we have been studying the design and implementation of `atomic` blocks built over software transactional memory (STM) or hardware transactional memory (HTM). Programming with SLE is not as simple as programming with `atomic` blocks. With SLE it is still necessary to decide which lock protects which piece of data. However, while `atomic` blocks are attractive in the longer term, SLE has a number of attractions for today's languages. First, ordinary locking can be used if the runtime system does not support SLE, or if the workload does not perform well with speculation. Second, an implementation can revert to ordinary non-speculative execution if the critical section attempts an operation that cannot be performed speculatively; implementations of `atomic` blocks over STM typically revert to serialising *all* transactions, or at least all irrevocable ones [Spear 2008, Welc 2008]. In contrast, our implementation of SLE serialises only those operations that require the same lock; this enables us to benefit from additional parallelism as determined by the original programmer.

Section 5 evaluates the performance of our prototype implementation based on manual use of the SLE runtime system. We compare the performance of data-structure micro-benchmarks based on coarse-grain locking with SLE against an implementation based on STM. We also examine the performance of the STAMP suite [Cao Minh 2008]. In this paper we have manually added calls to the SLE runtime system. This has been a straightforward mechanical process for the programs we have studied.

In Section 6 we discuss an alternative interface to the SLE runtime system that can be targeted automatically, without requiring source-code changes to an application. We discuss how different design decisions impact the likely performance of the resulting automated system. We relate this to the significant body of existing work on compiler support for language constructs based on STM, and similar systems based on binary translation [Olszewski 2007]. In future work we plan to evaluate the performance of our prototype on a wider set of applications, using a fully automatic implementation based on a lightweight binary rewriter.

Sections 7–8 discuss related work and conclusions.

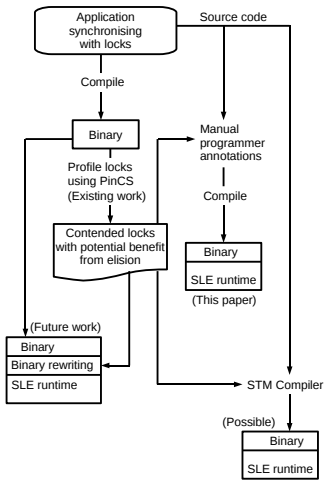


Figure 1. Overall system design.

2. Speculative Lock Elision

Figure 1 shows the structure of the complete system we are building. We envisage that existing multithreaded programs that use locks for synchronisation will hit bottlenecks as they are run on platforms with increasing numbers of hardware threads: as a result of lock contention, threads will wait on locks rather than doing useful work.

One solution to this scalability problem is to exploit any disjoint-access parallelism [Israeli 1994] that may be present: i.e. if two operations access distinct memory locations, then they may run in parallel, even though they may contend for the same lock.

Much recent research has focused on transactional memory (TM), and `atomic` blocks implemented over it. These blocks of code appear to execute atomically and in isolation, either through a software runtime system or some form of hardware support. However, typical semantics for `atomic` blocks mean that they are not a drop-in replacement for lock-based critical sections. For example, some implementations of `atomic` blocks support "single global lock atomicity", in which their behaviour is defined in terms of that of a program using a single process-wide lock [Larus 2007, Menon 2007; 2008]. With these semantics, programs that work correctly with locks may not work correctly with `atomic` blocks. Consider the following example using locks `l1` and `l2`:

```
// Example 1
// Initially: x=0

// Thread 1                    // Thread 2
lock(l1);                      do {
  lock(l2);                      lock(l2);
  x = 100;                         t2 = x;
  unlock(l2);                    unlock(l2);
  do {                         } while (t2 != 100);
    lock(l2);                  lock(l2);
      t1 = x;                    x = 200;
    unlock(l2);                unlock(l2);
  } while (t1 != 200);
unlock(l1);
```

This example is correctly synchronized; all accesses to `x` are protected by lock `l2`. When implemented with locks, Thread 2 is able to see Thread 1's write of 100 to `x`, and Thread 1 is subsequently able to see Thread 2's write of 200. However, when implemented with `atomic` blocks, or with a single global lock, these writes are not visible because Thread 1's outer critical section is serialised with Thread 2's critical sections.

As a consequence, researchers have studied forms of "lock elision" as a way to implement lock-based critical sections [Rajwar 2001, Rossbach 2007, Ziarek 2008]. These implementations retain the original semantics of the language, but allow threads to speculate past a lock-acquire operation and execute the critical section using TM-like techniques.

In this paper we assume that, as in the previous example, the original lock-based program is correctly synchronized and free from data races – i.e. when implemented using locks, the same location is not subject to multiple concurrent accesses, at least one of which is a write. We assume "catch fire" semantics for programs with data races; i.e. a correct implementation may behave arbitrarily for these programs. Recent work on memory models for C/C++ has taken this kind of approach, given the difficulty of constraining exactly how an optimised program may behave in the presence of data races [Boehm 2008].

The core of an SLE runtime system is similar to that of an STM: both must track the memory accesses made by concurrent threads and use some mechanism to detect and resolve conflicts. However, an SLE runtime system can be "best effort" in the sense that execution can always fall back to acquiring the lock and executing non-speculatively. This may be done if an implementation limit is reached (e.g. the amount of speculative state overflows a cache [Rajwar 2001]), if an operation is attempted that cannot be performed speculatively (e.g. waiting on a condition variable), or if the program uses locks in a way that the implementation cannot support (e.g. some forms of nesting, as in our previous example).

Conversely, an SLE runtime system faces a number of challenges that are not present in typical STMs:

Identifying profitable opportunities for SLE. In general, lock elision can only improve scalability when (*a*) there is little contention between the work different threads do in a critical section (e.g. speculation cannot speed up a critical section incrementing a single shared counter), and (*b*) there is contention for the lock protecting the critical section (e.g. a recent study showed little opportunity for using speculation in the Linux kernel: many data structures had already been designed to minimise lock contention [Rossbach 2007]).

Keeping this in mind, we believe that speculation should only be attempted on selected locks and some of their associated critical sections. We have constructed a tool (PinCS [Roy 2009]) that profiles unmodified x86 binaries at runtime, identifies critical sections, and predicts possible speedups assuming an ideal TM-based implementation. We have profiled a number of applications using this tool and seen possible speedups varying from 30% for some SPLASH applications to almost zero for well tuned applications such as Apache. For this reason we show this profiling step in Figure 1 as feeding into the elision process.

Integration with locks. The SLE runtime system must support the interaction between speculative execution and lock-based non-speculative execution of a critical section. Ideally, to support maximal parallelism, read-only lock-based critical sections should be able to run in parallel with read-only speculation. In addition to handling IO, system calls, condition variables, and so on, this is essential for performance when handling critical sections that have a high probability of conflict.

For example, consider the case of a hash table supporting a read-mostly workload. Ordinarily, lookups and updates will be able to proceed in parallel. However, an occasional rehashing operation needs to read every element of the hash table in a long-running critical section. This long-running section is unlikely to succeed using speculation since it will repeatedly see conflicts from the updating threads. Acquiring a read lock on the whole table allows the rehashing thread to scan entries while also letting concurrent lookups make progress. We examine this kind of workload later in the paper in Section 5.

Sandboxing speculative work. We must ensure that speculative work cannot affect non-speculative work. In particular, we must consider the execution of "zombie" speculative threads [Dice 2006] – that is, threads still performing speculative work that is doomed to be rolled back. For example, we must avoid access-after-free problems where a zombie thread accesses an object that has already been deallocated by a non-speculative thread: the memory that held the object may have been reused for another object, or even freed to the operating system.

Further problematic cases arise when pending updates from successful speculation are being written back to a block of memory that is being used non-speculatively. Consider the following example from a multithreaded program that uses linked lists:

```
// Example 2
// Thread 1                    // Thread 2
deleteNode(...) {              updateNode(...) {
  ListNode *node;                ListNode *node;
  lock(list);                    lock(list);
  ...                            ...
  node = ...;                    node = ...
  ...                            ...
  unlock(list);                  node->value = ...
  free(node);                    unlock(list);
}                              }
```

Without care, a TM-based implementation of these critical sections may allow Thread 2's speculative update to the `value` field to be written back *after* Thread 1's critical section has finished. It may even occur after Thread 1 has freed the node, leading to memory corruption. Variants of this

kind of "privatization" idiom are widely studied in work on TM [Spear 2007, Abadi 2008, Wang 2007, Dice 2006].

Our implementation of SLE must handle this kind of usage, because the example is correctly synchronized in the original lock-based implementation.

Preventing non-speculative threads waiting for speculative threads. We do not want the SLE runtime system to require a thread acquiring a lock non-speculatively to wait for a thread that currently holds the lock speculatively. This design decision limits the negative impact that speculation can have in workloads where it is not effective. Of course, this is also a generally desirable dynamic property since undue waiting while holding a lock can lead to convoys of threads. For example, consider this shortened fragment of code from the Apache web server:

```
// Example 3
apr_proc_mutex_lock(accept_mutex);
...
if (rv == APR_EGENERAL) {
  // E[NM]FILE, ENOMEM, etc
  resource_shortage = 1;
  signal_threads(ST_GRACEFUL);
}
...
apr_proc_mutex_unlock(accept_mutex);
```

In this fragment, a thread accepting a connection acquires a lock and, while holding this lock, may force the server to exit due to lack of resources. The runtime system must be careful not to let a speculative thread hold up the execution of the shutdown request, as this would unnecessarily delay the subsequent restart of the server.

Supporting application-specific lock types. In some situations, an application uses customised kinds of locks that are tailored to its own requirements. For example, lock implementations may busy-wait on the assumption that the lock will become available soon, or locks may be integrated with the thread scheduler to implement a priority inheritance protocol or other strategy to avoid priority inversion.

To avoid compromising these uses, the SLE runtime system acts as a wrapper above an existing lock so that, when speculation is not effective, the existing lock can be used along with whatever queueing or blocking semantics it is built to provide. We evaluate such a scenario in Section 5.

3. Manual SLE API

In this section we describe the interface to our prototype SLE runtime system. This is designed for manual use by a programmer; we describe the implementation (Section 4) and evaluate its performance (Section 5) before returning to consider a lower-level interface for automatic use by a compiler or binary translator (Section 6).

The manual API differs from the automatic one in two key respects. First, with the manual API, we do not wish to require the programmer to write separate code paths for speculative and non-speculative implementations of each critical section – this kind of duplication is straightforward to do automatically (e.g. as in STM implementations [Harris 2003, Wang 2007]), but error prone when done by hand. Consequently, our API is based on function calls that will perform fine-grained concurrency control operations during speculative execution, but will become (almost) no-ops during non-speculative execution. Second, we assume that the programmer will be able to identify the points in the program at which objects change from being shared via critical sections to being private to a given thread – e.g. where privatization idioms occur, such as Example 2 from Section 2. This seems straightforward in the practical examples we have studied, and earlier STM-based work has shown that it may give a scalability advantage over supporting such idioms directly in the runtime system [Menon 2007; 2008].

We illustrate the manual API using a running example shown in Figure 2(a). The example maintains a set of event counters protected by a common lock `event_lock` and accessed via a function `count_event`. Depending on the number and frequency of events the lock might be heavily contended, even when the dynamic instances of the critical section may access disjoint memory locations. This is the kind of scenario where SLE can prove useful for scalability. The function `dump_to_log` performs external IO, and so an SLE implementation would need to serialise the critical sections protected by `event_lock` while performing the IO. In contrast, a direct implementation of `count_event` with STM would typically serialise these operations with any other IO-performing operations in unrelated parts of the program.

The API for manual SLE contains 7 operations:

```
// Declare an elidable lock (Section 3.1):
SLE_ELIDABLE(base_lock_type, flags);

// Acquire/release elidable locks (Section 3.1):
SLE_LOCK(elidable_lock, base_lock_operation);
SLE_UNLOCK(elidable_lock, base_lock_operation);

// Access objects protected by a lock (Section 3.2):
ptr = sle_open_ro(ptr, size, version_ptr);
ptr = sle_open_rw(ptr, size, version_ptr);

// Prevent speculative access to objects  (Section 3.3):
sle_finish_sharing(ptr, size);

// Revert to non-speculative execution  (Section 3.4):
sle_fail_speculation();
```

Figure 2(b) shows how all but one is used in the example. We discuss the use of these seven operations in turn in Sections 3.1–3.4.

3.1 Locks

The `SLE_ELIDABLE` macro declares a new elidable lock. It specifies the underlying lock type, and flags for tuning the SLE implementation. In Figure 2(b), the global variable `event_lock` is an elidable lock based on an ordinary `pthread_mutex`. We explain possible tuning parameters in more detail in Section 4.

The `SLE_LOCK` and `SLE_UNLOCK` macros are used to acquire and release an elidable lock. These operations take an elidable lock as a parameter, along with the function to ac-

```
typedef struct event_counter {
 int count;
 ...
} ev_ctr;
ev_ctr counters[EVENTS];

pthread_mutex_t event_lock;

void count_event(int event)
{
 // local variable initialisation
 l1: int threshold = 0;
 l2: pthread_mutex_lock(event_lock);
 // Write to global shared state
 // protected by the lock
 l3: counters[event].count++;
 ... // other counter-specific code




 if(event_threshold(counters[event].count))
   threshold = 1;
 // Possibly do IO
 if(threshold)
   dump_to_log(event);


 pthread_mutex_unlock(event_lock);
}
```

```
typedef struct event_counter {
 int count;
 ...
} ev_ctr;
ev_ctr counters[EVENTS];
// SLE capable lock
SLE_ELIDABLE(pthread_mutex_t, FLAGS) event_lock;

void count_event(int event)
{
 ev_ctr *my_counter;
 int threshold;
 l2: SLE_LOCK(event_lock, pthread_mutex_lock);
 // move initialisation into critical section
 // to protect against restarts
 l1: threshold = 0;
 // Use shadow copy for changes
 l3_1: my_counter = sle_open_rw(&counters[event],
                          sizeof(ev_ctr), NULL);
 l3_2: my_counter->count++;
 ... // other counter-specific code
 if(event_threshold(my_counter->count))
   threshold = 1;
 // Possibly do IO
 if(threshold) {
   sle_fail_speculation();
   dump_to_log(event);
 }
 SLE_UNLOCK(event_lock, pthread_mutex_unlock);
}
```

Figure 2. Implementing event counters, (*a*) using coarse-grained locks, and (*b*) using the SLE runtime system.

quire/release the underlying lock if it needs to be used non-speculatively.

At runtime, an elidable lock is represented by a wrapper around an instance of the underlying lock type. The wrapper adds fields to control the interaction between threads holding the lock speculatively, and threads holding the lock non-speculatively. We discuss the implementation and behaviour of elidable locks in Section 4.1.

3.2 Data-access Operations

The functions `sle_open_ro` and `sle_open_rw` must be used when accessing objects protected by an elidable lock. The caller must pass in a pointer to the start of the object, and the size of the object in bytes. The SLE runtime returns a new pointer that the caller should use when making their actual accesses. For example, in Figure 2(b), the function `count_event` opens an element of the `counters` array for read-write access, and then makes accesses via the local variable `my_counter`. There are three main design choices:

The first is the granularity at which to track speculative work – primarily, whether to work on a per-object or per-word basis. For the manual API we support variable granularity by shadowing arbitrary byte-ranges, and leave the ultimate decision to the programmer. For example, in Figure 2(b), each `ev_ctr` is managed individually. Fine-grained speculation may reduce the number of false conflicts. Conversely, it may incur a higher book-keeping overhead.

The second design choice is how to associate conflict-detection metadata with the program's data structures. By default we follow the approach taken by Fraser and Harris [Harris 2003] and Dice *et al.* [Dice 2006] by using a hash function to map an address to a slot in an external metadata table. However, to improve spatial locality, a programmer may optionally place the metadata within the application's own data structures; if the programmer does this then they must pass in a pointer to the metadata as the final parameter to `sle_open_ro` and `sle_open_rw`. This is not done in the example in Figure 2(b), and so external metadata is used.

The third design choice is how to isolate speculative work from any threads that execute a critical section directly. As discussed in Section 2, we wish to ensure that speculative work does not affect the correctness of non-speculative work, and to also ensure that non-speculative threads do not need to wait for speculative threads. We do this by allowing a thread (T1) that acquires a lock non-speculatively to detect when another thread (T2) may speculatively access an object it requires and "revoke" access to it from the speculating thread. Revocation guarantees that any speculative threads will (*a*) not make further accesses to these objects, letting T1 access those objects directly, and (*b*) not delay the execution of T1. We discuss how we implement revocation in Section 4.4.

3.3 Memory Management

To avoid the problems posed by privatization-style idioms, or the corruption of freed memory by speculating threads, the programmer must call `sle_finish_sharing` before memory that can been accessed speculatively may be recycled for any other purpose (this includes thread-local usage, such as removing an element from a list and then accessing it directly outside a critical section).

To illustrate the use of `sle_finish_sharing`, we reproduce the linked list example from Section 2 with this call inserted:

```
deleteNode(...) {
  ListNode *node;
  SLE_LOCK(list, pthread_mutex_lock);
  ...
  node = ...
  ...
  SLE_UNLOCK(list, pthread_mutex_unlock);
  sle_finish_sharing(node, sizeof(ListNode));
  free(node);
}
```

The `sle_finish_sharing` call adds a synchronisation step that ensures that it is safe to access the block directly; it does not interact with the actual memory manager used in the application. For example, it works with the pre-allocated blocks used by the OSTM [Fraser 2003] and RSTM [Marathe 2006] test harnesses, as well as with the direct `malloc`/`free` calls used in STAMP.

3.4 Checkpoint and Rollback

The lock elision runtime system needs to checkpoint execution at the point where a lock is taken and roll back to that point if speculation fails (whether because of a conflict, or an attempt to perform an operation that cannot be done speculatively). Checkpointing can be done easily for C using the `setjmp`/`longjmp` facility. For C++ we have used the language's `try-catch` constructs. Of course, this basic kind of checkpointing does not affect all program state. There are two cases that need particular care:

The first is thread-local data held in the heap, and local variables outside the critical section that is being executed speculatively. Such changes are not rolled back when the stack is unwound. An example of this is the `threshold` variable in Figure 2. When using the manual SLE API, the programmer must ensure that updates to such state are either (*a*) tracked by the SLE runtime system, or (*b*) that they are idempotent. In the example, the initialisation of the `threshold` variable is moved into the critical section, as shown in Figure 2(b).

The second special case is the use of memory management functions. With the manual API, we handle `malloc` and `free` operations by undoing allocation requests made by speculative work that is rolled back, and by deferring `free` requests until speculative work commits. (Custom allocators may either be integrated with the SLE runtime system, or treated by falling back to non-speculative execution).

With the manual SLE API, the programmer must call `sle_fail_speculation` at any point when speculation needs to be cancelled – i.e. before any operation that is either irrevocable (e.g. invoking IO operations, waiting on a condition variable), or which affects state not tracked by the runtime (e.g. calls into a library that is not written to support SLE). Speculation must also be cancelled in cases where execution returns from a function containing the `SLE_LOCK` operation. This is because the checkpoint is then lost and insufficient information is available to roll-back execution. In all of these cases, execution reverts to acquiring the underlying lock.

As we discuss in Section 6, a compiler or binary rewriting engine can remove a lot of this burden from the programmer by tracking all updates to global state – not just those to shared data – and thus can automatically checkpoint extra state and insert elision aborts as required.

4. Runtime

In this section we describe the details of our implementation of SLE. We first describe the structure of an *elidable lock* (Section 4.1), which wraps the existing *application locks* declared in the program. Speculative execution uses thread-private logs of tentative updates ("lazy versioning" [Moore 2006]); this technique effectively sandboxes one thread's speculative work from that of another, using an object-versioning scheme to detect conflict (Section 4.2).

Locks are also used to mediate access to objects that are being accessed simultaneously by multiple speculative threads; these *fine-grained* locks are part of the SLE runtime system and are distinct from the application locks specified by the programmer. A form of commit-time-locking and version-number checks is used to determine whether or not speculation has been successful, and to apply any shadowed changes back to the heap (Section 4.3). In our implementation, non-speculative threads must check that the fine-grained locks of objects they are accessing are not held by speculative threads (Section 4.5). To ensure that non-speculative threads do not need to block on fine-grained locks, we use a revocable lock design (Section 4.4).

A key restriction in the case of nested critical sections is that once a critical section is elided, all enclosed ones must also be. SLE cannot allow the nested critical sections to be executed using non-speculative execution (consider the case where one lock is elided but an enclosed lock is not: changes made to objects protected by the enclosed lock cannot be rolled back). There is no problem with locks being held while starting elision, and hence the programmer does not need to reason about possible entry paths. However such locks cannot be released until all enclosed critical sections are completed.

4.1 Elidable Lock Metadata

The SLE runtime system must track the use of application locks in order to detect whether or not speculative and non-speculative uses of the lock have conflicted. In particular, to support multi-reader/single-writer locks, this means that a thread that acquires a coarse-grained application lock non-speculatively can run concurrently with read-only speculative threads. We add metadata fields to a lock to track both exclusive-mode non-speculative lock and unlock operations as well as to track the number of threads that currently hold the lock non-speculatively in read-only mode.

```
elidable_lock<T> {
  T base lock;          // underlying lock
  int version;          // inc on xlock/xunlock
  int read_lockers;     // counts #readers
  int occ_hashsize;     // tuning (Sec 4.2)
  int occ_speculations, occ_waitfor; // tuning (Sec 4.5)
  int check_freq;       // Must be >=1: (Sec 4.2)
};
```

Threads acquiring the underlying lock non-speculatively in exclusive mode increment the `version` field both on acquisition and on release, while threads non-speculatively acquiring the underlying lock in read-only mode atomically increment the `read_lockers` field on acquisition and decrement it on release.

Consequently, a thread attempting to acquire a lock in speculative mode must check whether or not the `version` field contains an even value. At commit time, a speculating thread must check that the version field has not changed (to ensure that it has not overlapped with a non-speculative exclusive-mode operation). Unless the speculation was read-only, a speculative thread must also check that the `read_lockers` count is zero to ensure that writes from speculative threads do not overlap with a read-only acquisition of the same lock.

4.2 Objects

As discussed in Section 3.2, runtime metadata can either be managed by an external lock table (using a hash function to map the address of a data item to its metadata), or metadata can be explicitly embedded within the application's data structures (passing in the metadata's address to calls to `sle_open_ro` and `sle_open_rw`). The metadata comprises a fine-grained lock and a version number. The fine-grained lock is used for concurrency control when committing speculative work back to the heap. The version number is updated on every such update, to signal a conflict to any concurrent speculative readers of the data.

Speculating threads need to execute in isolation. This involves logging memory locations that are read and buffering any changes until the speculating thread has finished executing the critical section and can successfully commit: the structure of the logs used is shown in Figure 3. The `sle_open_ro` and `sle_open_rw` operations take a thread-private snapshot of an object as follows:

```
snapshot(object, size, version_ptr, log_copy, log_version)
{
  log_version = *version_ptr;
  memcpy(log_copy, object, size);
  if(log_version != *version_ptr) {
    // something changed --- abort
    sle_fail_speculation();
} }
```

The version number of the object is changed by any thread that updates it (Section 4.3), thus letting threads take a consistent snapshot, as well enabling them to detect if their snapshots become out of date.

When held, a fine-grained lock points to the logged copy of the object it is currently protecting. This lets a thread

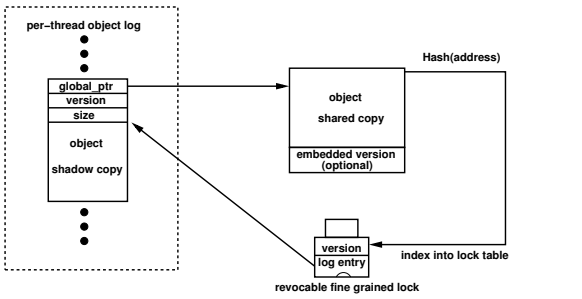


Figure 3. Log structure.

examining a fine-grained lock to quickly determine exactly which object in memory it is currently associated with. This is critical to the revoke operation we describe in Section 4.4. In our current implementation, if the same thread needs to update two objects protected by the same fine grained lock, it aborts elision. This limitation could be lifted by a more involved implementation, but this has not been indicated by our experience to date.

In order to efficiently determine whether or not a given object has already been logged, we use a chained hash table to index the object log. The hash table is implemented as a fixed (compile time) sized array, which is partitioned between the hash buckets and overflow chains. The partitioning is dynamic on a per elidable lock basis (through the tuning parameter `occ_hashsize`). A larger hash table means more single-probe lookups, but also results in more cache misses due to its random access nature, especially when it does not fit into the processor cache. The hash bucket includes a generation number that is set to the generation counter of the thread on allocation. The generation counter of the thread is incremented every time a new elision is begun. This allows us to reuse the hash table across elisions without reinitialising it. Running out of logspace or space in the hashtable is never catastrophic in our design: we fall back to acquiring the lock and executing non-speculatively.

Finally, SLE provides mechanisms to allow the programmer to ensure that the snapshot of memory taken by a speculating thread is consistent: i.e. that a thread reading from multiple objects sees a mutually-consistent view of them. This ensures that speculating threads do not enter infinite loops or suffer other fatal behaviour as a result of conflicting accesses from other speculative as well as non-speculative threads. We use two approaches:

The first approach, as in TL2 [Dice 2006], is to use a global version number. A thread observes the current global version number when it starts speculating (its *base version number*). A thread increments the global version number if it successfully finishes speculating, and it writes the resulting number into any fine-grained locks that it has held. This means that a thread is guaranteed to see a mutually-consistent view of memory provided that the version number in each fine-grained lock that it accesses is older than its base

```
void commit_speculation() {
  cas(tx_state, EXECUTING, UNDECIDED);
  if (!read_only)
    foreach obj in dirty_list
      if (trylock(lock(obj)) == FAIL)
        goto cleanup;
    if (!verify_snapshot())
      goto cleanup;

    BEGIN_RESTARTABLE();
    foreach obj in dirty_list
      if (held(lock(obj)))
        version(obj) = log(obj).version + 1;
    END_RESTARTABLE();

  // Linearisation point here

  if (!verify_snapshot())
    goto cleanup;
  if (!cas(tx_state, UNDECIDED, SUCCESS))
    goto cleanup;

  BEGIN_RESTARTABLE();
  foreach obj in dirty_list
    if (held(lock(obj)))
      copy(shared(obj), private(obj));
      version(obj) = log(obj).version + 2;
    unlock(lock(obj));
  END_RESTARTABLE();

  cas(tx_state, SUCCESS, EXECUTING);
  return; // Success

cleanup:
  undo object version changes
  release object locks
  // tx_state may have changed to FAILED
  cas(tx_state, tx_state, EXECUTING);
}
```

Figure 4. Commit.

version number. We use this approach when benchmarking SLE against TL2 and RSTM in Section 5.

Following forms of DSTM [Herlihy 2003] and OSTM, the second approach that we support is to periodically revalidate the objects that a thread has accessed. This validation involves checking that:

- the versions of all logged objects are unchanged; and
- the version numbers of elided locks are unchanged.

The frequency of this validation is controlled by a programmer provided tuning parameter `check_freq`. A value of 1 leads to doing validation on every addition to the snapshot, which ensures a consistent snapshot at all times, but incurs a performance overhead. Larger values cause the SLE runtime to perform less frequent validation, which allows temporary inconsistencies, but can potentially reduce the performance overhead significantly. The latter option is preferable if the programmer is certain that execution on inconsistent snapshots will not be problematic. We use this approach with `check_freq` set to 10 when benchmarking against OSTM.

4.3 Commit

When a speculatively executing (outermost) critical section finishes, the corresponding `SLE_UNLOCK` on the elided lock triggers the commit operation shown in Figure 4. The commit operation applies changes atomically to the shared heap using a variation of the well known 2-phase commit protocol adopted by many STM implementations. There are three noteworthy aspects of our commit operation:

The first is that validating the snapshot means checking (*a*) version numbers on shadowed objects are unchanged, (*b*) version numbers of elided locks are unchanged, (*c*) either this critical section has not updated any objects, or the reader counts of elided locks are zero. The checks take into account versions that have already been locked by the committing thread.

The second point is that we need to manipulate both the external lock table as well as any per-object embedded version numbers. This leads to the extra validation pass shown in the figure compared to a standard 2-phase commit.

Finally, we must consider the possible revocation of the fine-grained locks. In our implementation, the revocation notification comes as a UNIX signal, which is ignored unless we are in a section marked by `BEGIN_RESTARTABLE` and `END_RESTARTABLE`. If we are in such a section, the signal handler executes a `longjmp` to transfer control flow to the checkpoint taken at the thread's most recent call to `BEGIN_RESTARTABLE`. Since we always check that the fine-grained lock is held before applying changes to the associated object this scheme ensures that once the notification is received, the thread that has lost the lock will no longer make any updates to the associated object.

4.4 Revocable Fine-Grained Locking

A key building block of the STM is revocable locks. As in earlier work [Harris 2005], the idea is to provide locks that can be revoked, displacing the original lock holder's execution to a special clean-up path. We implemented revocable locks in software for off-the-shelf microprocessors to serve as a basis for our SLE runtime system.

We added revocation support to our fine-grained locks by adding a version number and a pointer to the log entry of the object which is locked. We implemented a `trylock` call that attempts to acquire a lock, either succeeding, or returning without blocking if the lock is already held. This is implemented by incrementing the version number both on acquire and on release. Consequently, an odd version number indicates that a lock is held (a reuse of the scheme in Section 4.1). We also have a `revoke` operation that operates as shown in Figure 5. The thread revoking the lock notifies the previous lock holder to inform it that it has lost the lock. Once the notification is received by the other thread then the thread revoking the lock can assume that objects protected by the lock will no longer be touched by the previous lock holder (this includes the log-entry-pointer field within the revocable lock itself). The thread revoking the lock also takes the responsibility of copying back changes from the original holder if it has successfully committed.

For brevity, Figure 5 does not cover the case where multiple threads have acquired the application lock in reader

```
revoke(lock, global_ptr, size)
{
  version = lock.version;
  if(version is even) // unlocked
    return;
   logentry = lock.log;
   // Is lock associated with object ?
   if(logentry.global_ptr != global_ptr)
     return;
   memcpy(local_buffer, logentry.shadow, size);
   other_thread = logentry.thread;
   other_state = other_thread.tx_state;
   if(other_state == UNDECIDED) {
     // Push to a failed decision
     cas(other_thread.tx_state, UNDECIDED, FAILED);
     other_state = other_thread.tx_state;
   }
copyback:
   if(other_state == SUCCESS && lock.version == version)
     memcpy(global_ptr, local_buffer, size);
revoke:
   if(cas(lock.version, version, version + 2) == version) {
     // we have the lock
     // send revoke notification --- synchronous
     revoke_notify(other_thread);
     lock.log = NULL;
     cas(lock.version, version + 2, version + 3);
   }
 }
```

Figure 5. Fine-grained lock: revoke operation.

```
void revoke_notify(target) {
  int old_interrupt_count, cpu;
  signals_pending(target).restart = true;
  cpu = get_cpu(target);
  old_interrupt_count = interrupt_count(cpu);
  send_interrupt(cpu);
  while (old_interrupt_count == interrupt_count(cpu));
  // guaranteed signal delivery at this point
}
```

Figure 6. Synchronous signalling.

mode and then concurrently try to revoke the same fine-grained lock. In practise this is handled by further encoding the fine-grained lock version to indicate when it is being held by a revoking thread. In this case other threads do not attempt to further revoke it in the `revoke` stage. However we still allow them to perform the `copyback` stage concurrently (and not block on each other). This is safe since they are copying back the same values and no other writer can access the object (the application read lock is held).

One challenge we faced was to implement a non-blocking synchronous notification mechanism. We do this by providing a new system call that allows the revoking thread to send a synchronous signal to the thread previously holding the lock being revoked. The typical UNIX signal interface provided does not provide such a facility, and hence we have built a custom Linux kernel module which provides the synchronous signalling capabilities we require.

Figure 6 gives pseudocode for the new `revoke_notify` system call. It first marks the signal as pending in the target thread. It then determines the CPU that the target thread is running on (or last running on if not running currently). Finally, it sends a reschedule interrupt to the target CPU. It then waits for the per-interrupt counter on the target CPU to be incremented, indicating that the interrupt has been received. At this point the system call returns. Since signals are always delivered on a return to user-mode from kernel-mode in Linux, this guarantees that once an interrupt is received the signal will be delivered (if it has not been already).

On systems where the kernel module is not available, threads simply fall back to blocking when encountering held fine-grained locks.

```
check_access_safe(obj, version_ptr) {
  if (version_ptr != NULL) {
    unsafe = is_locked(version_ptr);
  } else {
    unsafe = is_locked(lock(obj));
  }
  if (unsafe) {
    revoke(lock(obj));
} }
```

Figure 7. Access check for non-speculative execution.

4.5 Non-Speculative Execution

If a critical section is executed non-speculatively then the `sle_open_ro` and `sle_open_rw` operations no longer need to track the data accesses made by a thread, and no longer need to build up a commit set to write back to shared objects at the end of the critical section. However, a non-speculative thread must still ensure that it sees any updates from speculative threads that are in their write-back phase.

We achieve this by having the non-speculative thread check the version number of any object being accessed. If the version number indicates that the object is currently locked, then the non-speculative thread revokes the fine-grained lock. Since our lock revocation system call cannot block, this means that threads holding a lock never need to wait on threads that have speculated past it, a cornerstone of our design. The pseudocode given in Figure 7 illustrates the procedure used by a non-speculating thread to determine that a particular access is safe.

Another area of interaction between non-speculative threads and speculative ones is memory management. As we have described, speculation can lead to the write-after-free problem, where a block is written to after it has been freed. Since we wanted a non-blocking solution, we eschewed the epoch based memory management [Kung 1980] schemes used in OSTM and RSTM. Instead we use a variant of the scheme in TL2, which treats the fine-grained lock as a hazard pointer [Michael 2004]. The call to `sle_finish_sharing` checks if the fine-grained lock on the object is held. If the lock is held then there may be pending speculative updates to the object, in which case the lock is revoked. The operation then returns indicating that it is safe to reuse the block.

Finally, when an elidable lock is encountered either when executing or after an aborted attempt to speculate past it, the runtime system needs to decide whether to elide it or fall

back to lock acquisition. Also, if the lock is currently held, the runtime system needs to decide whether to wait for it to become free and then try elision or to abandon elision altogether for this instance of the critical section. To control and tune runtime behaviour, we provide two further tuning parameters on a per-elidable-lock basis.

The first parameter is `occ_speculations`: this determines the the maximum number of attempts that the elision runtime system will make to elide this lock. On suffering that many failures in speculation, the implementation falls back to pessimistically acquiring the lock – a value of zero for this parameter means that elision is never attempted.

The second parameter is `occ_waitfor`. In the case that a lock is unavailable when attempting elision, this defines the number of acquisitions of the lock by *other* threads that the elision runtime will wait for before abandoning the attempt to speculate past it. If this limit is reached, the thread switches to pessimistic acquisition of the lock for this dynamic instance of the critical section.

Larger values for these two parameters pushes the runtime system towards more optimism, approximating `atomic` blocks. Smaller values result in a closer approximation of the conventional blocking behaviour on locks.

5. Evaluation

We evaluate our SLE runtime system by using the SLE API described in Section 3. The evaluation is driven by two motivating factors. The first is to confirm that the complexity of the commit operation in Figure 4 does not affect scalability. The second motivation is to show how our design decisions, such as non-blocking execution of non-speculative threads and non-blocking memory management, improve the robustness of applications using the runtime.

Our benchmarking methodology uses the test harnesses of OSTM and STAMP (both written in C) and RSTM (written C++). We use three different machines and two different architectures: a large ccNUMA Itanium-2 machine (Altix 4700, 52 Madison cores, 456 GB memory, 38 ccNUMA nodes); a 16-way x86 SMP (machine 4 sockets, 2 cores per socket, 2 hyperthreads per core, running in 64-bit mode); and a smaller x86 SMP machine with one hyperthreaded CPU running in 32-bit mode. We have deliberately used different architectures to verify that our implementation of SLE is portable across the stronger memory model of the x86 and the weaker one of the Itanium-2: these memory models roughly represent opposite ends of the memory consistency scale. We use the implementation of revocable locks only on the x86 machines (we do not have superuser access on the Altix machine).

All our experiments use the default STM of the test harness in question as a baseline. We emphasise here that our purpose is not to compare the absolute performance of SLE with the other STMs and hence have not tuned parameters for the elidable locks. Except where explicitly noted, all experiments use fixed settings of `occ_hashsize = 32`, `occ_speculations = 17` and `occ_waitfor = 17`, which we found during development to be adequate for reasonable performance from SLE for most of the benchmarks.

5.1 Scalability

The first set of scalability experiments use the highly scalable Altix machine. Figure 8 shows the results of running concurrent skip list and red-black tree benchmarks, using a lock-based version, OSTM and our SLE runtime system. The benchmarks perform a mixture of 75% lookups and 25% accesses split between updates and deletes. SLE shows no scalability bottlenecks even at 64 threads, keeping the execution time for the critical section constant, regardless of the number of threads. It is able to use all the disjoint access parallelism hidden behind the coarse-grained lock that we used to protect the data structure. The pure lock-only versions on the other hand are unable to scale, even the sophisticated fine-grained ones. This is because they serialise on exclusive access to the same cache line even for read locking. SLE performs worse than OSTM for skip lists by a constant factor since it needs to copy entire skip list objects for read and write access rather than just indirection pointers in the case of OSTM.

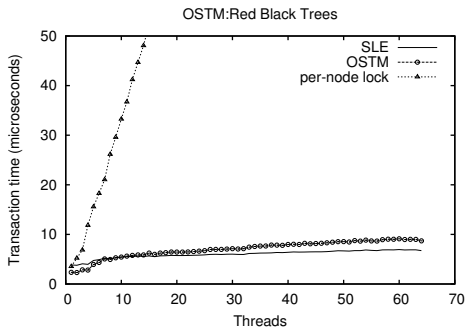
The next scalability experiment uses the 16-way x86 SMP. The experiment demonstrates the value of placing a version number in objects. We evaluated two variants of red black trees in OSTM using our SLE runtime. The first variant uses version numbers within objects while the other variant uses external metadata. As Figure 9 shows for this benchmark with 75% lookups, the read only critical sections benefit heavily (around 15%) due to the version number being on the same cache-line as the object itself.

5.2 Robustness

In this section we evaluate the robustness of our design, and the impact of our design goals of supporting non-speculative execution, and avoiding the need for non-speculative threads to block waiting for speculative threads.

The first experiment uses the concurrent hashtable implementation in RSTM. The concurrent hashtable uses 256 buckets with overflow chains. The experiment uses the 16-way x86 SMP with 15 threads devoted to inserting or deleting entries from the hashtable. The last thread runs a long critical section reading every element in the table, simulating a hash table rehash. This thread finds it difficult to make progress when executing speculatively due to the other threads executing shorter critical sections that conflict with its extremely large read set.

We ran the experiment with two version of SLE: one with the default tuning parameters, and one which optimistically always speculates and never falls back to explicitly acquiring a lock. As expected, the optimistic version results with the long running thread being completely starved of CPU time, while the standard version of SLE adapts to the situation



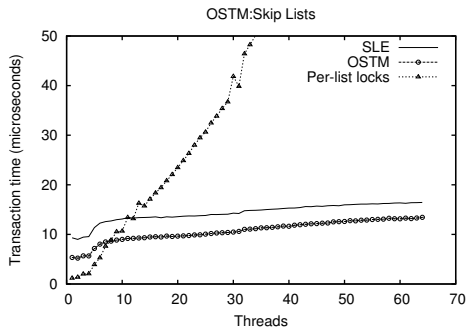


Figure 8. OSTM: red-black trees and skip lists

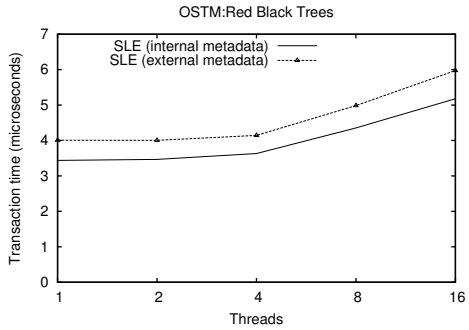


Figure 9. OSTM: external vs. internal metadata

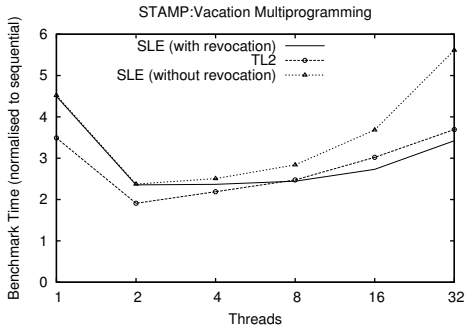


Figure 10. STAMP: Vacation under multiprogramming

and ensures liveness. We also ran this benchmark on RSTM using the default Polka contention manager; once more the long running thread is starved due to repeated conflicts with the short running critical sections.

The next two experiments use an x86 system with a single hyper-threaded processor. This lets us explore settings where there are more software threads than hardware ones. For brevity, we report results from just one benchmark from each of the STAMP and RSTM suites; the other benchmarks lead to similar conclusions.

We first run the Vacation benchmark from the STAMP suite and impose multiprogramming by launching upto 16 threads on the testbed (which has only two hardware threads) as well as tuning the benchmark for high conflict.

We measured two different SLE configurations: the first uses lock revocation, while the second does not. We used a POSIX-threads mutex for the coarse-grained lock in SLE. Figure 10 shows that as the degree of multiprogramming increases lock revocation is extremely useful to prevent situations where the non-speculative thread is forced to wait on threads that have uncompleted speculative work and have been de-scheduled while in their commit phase. The use of the heavyweight POSIX-threads mutex reflects the *intention* of the programmer to handle multiprogramming scenarios and avoid any priority inversion. When speculation causes busy waiting on fine-grained locks it effectively circumvents that intention. TL2 is also affected by multiprogramming: note that we are using the default conflict manager for TL2 and thus its performance in such circumstances could possibly be improved by tuning it. A non-blocking STM may be particularly suitable to this workload [Marathe 2008].

The final experiment in this section uses a version of RSTM where we deliberately cause one selected thread to be de-scheduled in its commit phase, both in RSTM as well as the SLE runtime system. Since RSTM is non-blocking and our SLE implementation is designed to isolate changes made by a speculating thread, this is not immediately fatal to the application. However in the case of RSTM a thread stuck in its commit phase means that the global epoch is unable to make progress and hence freed blocks cannot be returned for reuse. We tracked the number of free memory blocks in the system by means of the `vmstat` utility. RSTM was unable to reuse freed blocks, continually allocated blocks to meet its demand and ultimately crashed, being terminated by the operating systems when free system memory fell below a critical water mark. SLE on the other hand did not have any problem since the crashed thread cannot prevent memory management from making progress.

5.3 Performance

In this section, we evaluate the overall performance of SLE on the STAMP benchmarks. The eight STAMP benchmarks include a gene sequencing program (“Genome”), a bayesian learning network (“Bayes”), a network intrusion detection

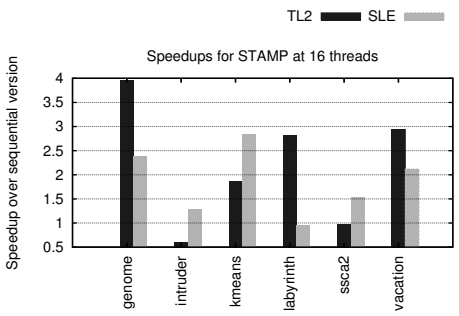


Figure 11. STAMP: speedups with 16 threads

algorithm ("Intruder"), a k-means clustering algorithm ("K-Means"), a maze routing algorithm ("Labyrinth"), a set of graph kernels ("SSCA2"), a client-server reservation system simulating SpecJBB ("Vacation") and finally a Delaunay mesh refinement algorithm ("Yada"). Of these Yada and Bayes failed to run out of the box on our hardware using the TL2 version available with it. Hence we do not report results for these two benchmarks. For the remaining benchmarks we report speedup on the 16-way x86 SMP over the sequential version of the benchmark. We use the largest recommended input set in each case to run non trivial instances of the benchmark. As Figure 11 shows, our SLE implementation scales well in many of these benchmarks. One notable exception is Labyrinth, where we do not support the low-level "read set discard" optimisation used in the benchmark – this operation has no counterpart in a lock-based program.

We have not tuned either TL2 or our own runtime; performance can be significantly improved in either case by tuning the parameters in each runtime. For example, we found that increasing the `occ_hashsize` parameter increased performance in all the benchmarks, ranging from 5% in vacation to 30% in genome. In the case of labyrinth, our assumption of 4 byte aligned objects when constructing the hash function for mapping to the locktable leads to excessive conflicts. Switching to a slightly modified hash function that correctly assumes object alignments of 8 (labyrinth uses arrays of 64-bit elements) led to SLE performing exactly as well as TL2. Based on this observation we are considering exposing object alignments as another tunable parameter on a per-lock basis (assuming the same set of locks is always needed for the same object).

Overall, our evaluation of SLE shows that when carefully used it can provide significant scaling to code that synchronises using coarse grained locks. Using finer object granularities is clearly beneficial for performance by avoiding unnecessary shadowing of large objects, and using embedded version numbers can also improve performance. Finally, tuning elidable lock attributes such as the hash table size based on the nature of the critical section can further contribute to better performance. We intend to explore such tuning in more detail in later work, ideally building an automatic solution.

6. Automating SLE

In Sections 3–5 we have used an interface to our SLE runtime system that is designed for manual use in a C/C++ program. This has let us evaluate the possibilities for software-based implementations of SLE on a small number of benchmarks, and let us tune some aspects of the runtime system's implementation. We are currently extending this system to form an implementation that works entirely automatically, and we are integrating it with our profiling work [Roy 2009].

The core SLE runtime system stays as described in this paper, but there are a number of differences between the API for manual use and the API for automatic use:

- First, when SLE is applied automatically, we cannot rely on the programmer identifying convenient locations to store SLE metadata. In the worst case we must always use an external metadata table, and use a hash function to map an address to its associated metadata. Static analyses may help in some typical usage patterns, as with hybrid word/object STMs [Riegel 2008]. Data read/write operations expand to calls into the runtime system in the same manner as with an STM implementation.
- Second, in the automated setting, we cannot rely on the careful placement of `sle_finish_sharing` operations. Again, we can employ known techniques from STM implementations; either introducing additional synchronisation modelled on Menon *et al.*'s implementation of privatization safety [Menon 2008], or using memory protection hardware to identify transitions [Abadi 2009]. Note that whereas Menon *et al.*'s scheme uses process-wide synchronisation with single global lock atomicity, we would only require per-application-lock synchronisation.

Our anticipated automatic implementation of SLE is based on a lightweight binary rewriting system, modifying the implementation of speculative critical sections, and the implementation of lock/unlock operations. We wish to avoid modifying the implementation of code outside critical sections (so that the cost of binary rewriting is not incurred there). As with Judo-STM [Olszewski 2007], changes made by `malloc` and `free` can be tracked by the runtime system, thus ensuring that changes to the memory management data structures are made atomically on commit (a scalable `malloc-free` implementation will typically ensure that there are no conflicts between allocations by different threads). Furthermore, any calls to extend the available heap area using `mmap` or `sbrk` system calls can either be implemented by falling back to acquiring a lock, or they can be handled as special cases.

We can choose whether or not to require modification to non-speculative code inside critical sections. One option, as with our manual prototype, is for this non-speculative code to include calls such as `check_access_safe` to ensure that the data it accesses is not being used by a speculative thread. This is desirable in the manual case because it avoids introducing synchronisation between non-conflicting operations

on the elidable locks. However, since the automatic case already requires this synchronisation for privatization safety, it may be preferable to control speculative/non-speculative interactions on a per-application-lock basis rather than a fine-grained object basis. This would avoid the need for any changes to the implementation of non-speculative critical sections. Finally, lock and unlock demarcations of critical sections may not be straightforward [Wang 2008]. However we can always fall back to non-speculative execution in such cases.

7. Related Work

There has been previous work on combining transactions with locks. TxLinux [Rossbach 2007] combines HTM-implemented critical sections with software-implemented spin-locks in the Linux kernel. In our work we have explored software-only approaches for current off-the-shelf processors, and looked at combining speculative and non-speculative execution for multiple kinds of locks. Another difference is that the HTM used by TxLinux results in any thread acquiring a lock non-speculatively to automatically abort any threads that hold it speculatively. We implement this functionality entirely in software.

Ziarek *et al.* have combined traditional Java monitors with a form of TM-based optimistic concurrency control [Ziarek 2008]. As with Ziarek *et al.*'s work, we only support correctly-synchronized programs. However, we allow the use of non-speculative read-only locking (rather than Java's exclusive-mode monitors), we work without the use of an automatic garbage collector to reclaim storage, and we avoid the need for non-speculative threads to block for speculative ones.

Many STM runtime systems support a notion of *irrevocable* transactions [Spear 2008, Welc 2008]. These mechanisms allow at most one transaction to become irrevocable – e.g. so that it may make system calls. This provides a general-purpose, safe, but potentially non-scalable, implementation technique for `atomic` blocks that have external side-effects. SLE retains the original coarse-grained locks in the application, and so it can fall back to using these for non-speculative execution, rather than using a single global lock.

Marathe and Moir designed an STM system that combines a streamlined blocking fast-path with a more complex non-blocking algorithm [Marathe 2008]. Our desire to support existing non-speculative code led us to investigate OS mechanisms to remove obstructing threads, rather than designing a non-blocking user-mode algorithm.

Concurrent with our own work, Smaragdakis *et al.* identified cases where lock-based and TM-based implementations of critical sections do not coincide, and examined the case for dynamically selecting between different implementations of critical sections [Smaragdakis 2008].

8. Conclusion and Future Work

We have described the design and implementation of a software lock elision runtime. The runtime system allows threads to execute critical sections speculatively, works on off the shelf microprocessors, does not impose any memory management constraints and does not require threads holding a lock to wait on threads that have speculated past it. We achieve this non-blocking property without using indirection on the heap, instead using a novel software-only implementation of revocable locks.

As the performance evaluation suggests, a key direction for future work is examining the best way to automatically tune parameters of the runtime for the best performance. Another important area of work we are pursuing is integration with a lightweight binary rewriting engine in order to remove all burden on the programmer.

9. Acknowledgements

We would like to thank the reviewers for their constructive comments, and our shepherd George Candea for his focused efforts in helping us towards a better paper. We would also like to thank Adam Welc, Jean-Philippe Martin and Terence Kelly for their feedback. Finally, we thank DAMTP Cambridge for the use of their Altix 4700 supercomputer.

References

[Abadi 2008] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL '08, 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 63–74, January 2008.

[Abadi 2009] Martín Abadi, Tim Harris, and Mojtaba Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *PPoPP '09, 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2009.

[Blumofe 1995] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. Technical Report MIT-LCS-TM-548, 1995.

[Boehm 2008] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *PLDI '08: Proc. 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–78, June 2008.

[Cao Minh 2008] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proc. IEEE International Symposium on Workload Characterization*, pages 35–46, September 2008.

[Dice 2006] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC '06: Proc. 20th International Symposium on Distributed Computing*, pages 194–208, September 2006.

[Fraser 2003] Keir Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[Harris 2003] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proc. 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, November 2003.

[Harris 2005] Tim Harris and Keir Fraser. Revocable locks for non-blocking programming. In *PPoPP '05: Proc. 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 72–82, June 2005.

[Herlihy 2003] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proc. 22nd ACM Symposium on Principles of Distributed Computing*, pages 92–101, July 2003.

[Israeli 1994] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *PODC '94: Proc. 13th ACM Symposium on Principles of Distributed Computing*, pages 151–160, August 1994.

[Kung 1980] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5 (3):354–382, 1980.

[Larus 2007] Jim Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[Marathe 2008] Virendra J. Marathe and Mark Moir. Toward high performance nonblocking software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 227–236, New York, NY, USA, February 2008.

[Marathe 2006] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer, III, and Michael L. Scott. Lowering the overhead of software transactional memory. Technical Report TR 893, Computer Science Department, University of Rochester, March 2006.

[Menon 2008] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard Hudson, Bratin Saha, and Adam Welc. Single global lock semantics in a weakly atomic STM. In *TRANSACT '08, 3rd ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, February 2008.

[Menon 2007] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Towards a lock-based semantics for Java STM. Technical Report UW-CSE-07-11-01, University of Washington, Nov 2007.

[Michael 2004] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[Moore 2006] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proc. 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. February 2006.

[Olszewski 2007] Marek Olszewski, Jeremy Cutler, and J. Gregory Steffan. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In *PACT '07: Proc. 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 365–375, September 2007.

[Rajwar 2001] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO '01: Proc. 34th ACM/IEEE International Symposium on Microarchitecture*, pages 294–305, December 2001.

[Riegel 2008] Torvald Riegel and Diogo Becker de Brum. Making object-based STM practical in unmanaged environments. In *TRANSACT '08, 3rd ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, February 2008.

[Rossbach 2007] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. TxLinux: using and managing hardware transactional memory in an operating system. In *SOSP '07: Proc. 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 87–102, October 2007.

[Roy 2009] Amitabha Roy, Steven Hand, and Tim Harris. Exploring the Limits of Disjoint Access Parallelism. In *Proceedings of HotPar '09: 1st USENIX Workshop on Hot Topics in Parallelism (to appear)*, March 2009.

[Smaragdakis 2008] Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. General and efficient locking without blocking. In *MSPC '08: Proc. 2008 ACM SIGPLAN workshop on Memory Systems Performance and Correctness*, pages 1–5, March 2008.

[Spear 2007] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. Technical Report TR 915, Computer Science Department, University of Rochester, February 2007.

[Spear 2008] Michael F. Spear, Maged M. Michael, and Michael L. Scott. Inevitability mechanisms for software transactional memory. In *TRANSACT '08: Proc 3rd ACM SIGPLAN Workshop on Transactional Computing*. February 2008.

[Wang 2007] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO '07: Proc. 2007 International Symposium on Code Generation and Optimization*, pages 34–48, March 2007.

[Wang 2008] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott A. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI '08: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 281–294, 2008.

[Welc 2008] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA '08: Proc. 20th Symposium on Parallelism in Algorithms and Architectures*, pages 285–296, June 2008.

[Ziarek 2008] Lukasz Ziarek, Adam Welc, Ali-Reza Adl-Tabatabai, Vijay Menon, Tatiana Shpeisman, and Suresh Jagannathan. A uniform transactional execution environment for Java. In *ECOOP '08: Proc. 22nd European Conference on Object-Oriented Programming*, pages 129–154, July 2008.