Software Transactional Memory for Large Scale Clusters *

Robert L. Bocchino Jr. Vikram S. Adve
University of Illinois at Urbana-Champaign
201 N. Goodwin Ave.
Urbana, IL 61801
{bocchino, vadve}@cs.uiuc.edu

Bradford L. Chamberlain
Cray Inc.
411 First Ave. S., Suite 600
Seattle, WA 98104
bradc@cray.com

Abstract

While there has been extensive work on the design of software transactional memory (STM) for cache coherent shared memory systems, there has been no work on the design of an STM system for very large scale platforms containing potentially thousands of nodes. In this work, we present Cluster-STM, an STM designed for high performance on large-scale commodity clusters. Our design addresses several novel issues posed by this domain, including aggregating communication, managing locality, and distributing transactional metadata onto the nodes. We also re-evaluate several STM design choices previously studied for cache-coherent machines and conclude that, in some cases, different choices are appropriate on clusters. Finally, we show that our design scales well up to 512 processors. This is because on a cluster, the main barrier to STM scalability is the remote communication overhead imposed by the STM operations, and our design aggregates most of that communication with the communication of the underlying data.

Categories and Subject Descriptors D.1.3 [*Software*]: Concurrent Programming—Distributed Programming; D.1.3 [*Software*]: Concurrent Programming—Parallel Programming; D.3.3 [*Software*]: Language Constructs and Features—Concurrent Programming Structures

General Terms Languages, Performance

Keywords Software Transactional Memory (STM), Distributed Memory Architectures, Clusters, Scalability

1. Introduction

Transactional memory is a promising mechanism for simplifying shared memory parallel programming. Transactions improve upon locks because they are easier to reason about, are more composable, and in some cases provide progress guarantees without complex algorithm design. A transactional memory solution may be implemented purely in software (software transactional memory, or STM), purely in hardware, or some hybrid of the two.

On a distributed memory system, transactional memory is a natural fit for languages that use a Partitioned Global Address Space (PGAS) programming model. Such languages include Unified Parallel C (UPC) [12], Co-Array Fortran (CAF) [28], Titanium [38], and the languages developed in the DARPA HPCS program (Chapel [8], Fortress [37], and X10 [9]). Indeed, two of the three HPCS languages (Chapel and Fortress) incorporate constructs implying transactional memory. Implementing these features effectively requires an efficient and highly scalable transactional runtime system that can be targeted by compilers or by library writers for those languages. The implementation must use an STM scheme (with or without hardware acceleration) because there is generally no hardware support for shared address spaces across nodes in the system.

However, while there has been extensive work on STM designs for cache coherent shared memory systems, to our knowledge *there is no work investigating STM designs for large-scale systems of any kind*. Prior STM work has focused almost exclusively on relatively small scale cache coherent systems (e.g., 1–50 processors). We refer to this work collectively as *cc-STM*. The only STM work we know of that is intended for message-passing systems relies on global cache coherence and has not been shown to scale to large numbers of nodes [19, 22].

The most important difference between smaller-scale hardware shared memory systems and large-scale distributed memory systems is that the former support fast access to arbitrary (local or remote) addresses, while the latter require expensive communication for remote accesses, typically using a software communication layer. On a commodity cluster, remote accesses can be several orders of magnitude slower than local ones (i.e., accessing main memory within a single cluster node), and it is therefore critical for performance to distribute both the data and the computation to minimize communication.

These observations lead to a very different design rationale for STM on a large cluster than on a cache coherent machine. In cc-STM, the key to good performance is to reduce the scalar (i.e., single-processor) overhead imposed by the STM bookkeeping operations. On a cluster, there are three potentially important goals: (1) to reduce or eliminate the *extra* remote operations imposed by the STM; (2) to ensure that the STM interface and implementation enable aggregating communication and exploiting locality; and (3) to minimize scalar overhead. In fact, the scalar overhead does not impact *scalability*: it primarily adds a constant factor overhead to overall performance. While such constant factors can be important, we expect that scalar overhead will be less important on large-scale clusters for two reasons. First, except for very efficient programs with low communication overhead, the scalar overhead

* This work is supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-001 and by the National Science Foundation Computer Systems Research program under grant number CNS 07-20772. The work is also supported by NSF under the following programs: Partnerships for Advanced Computational Infrastructure, Distributed Terascale Facility (DTF) and Terascale Extensions: Enhancements to the Extensible Terascale Facility.

PPoPP'08, February 20–23, 2008, Salt Lake City, Utah, USA.

will often be overlapped with (and dominated by) remote communication costs. Second, we expect that hardware support will be added to multicore processors to reduce this scalar overhead significantly [33, 26], while the other two issues require careful STM interface and algorithm design.

In this work, we make four distinct contributions:

1. We present Cluster-STM, an STM system designed for high performance on large-scale distributed memory systems such as commodity clusters. Our design addresses several important issues for these systems that have not been addressed in previous STM designs, including managing locality and distributing STM metadata onto the nodes. Both the STM library interface and the underlying algorithm implementing that interface are important in achieving these goals.
2. We evaluate a prototype of Cluster-STM, implemented on top of the GASNet communication library [6], on up to 512 processors of a large Intel Xeon cluster. We show that our design scales well across that range of processors because we carefully reduce, and in many cases eliminate, any extra remote-access overhead imposed by the STM bookkeeping operations. We do this by folding the bookkeeping operations into the underlying data communication operations that are occurring anyway.
3. We decompose the cc-STM work into an eight-axis space that allows most existing designs to be expressed as points in the space. This decomposition helped us identify what design issues, previously studied for cc-STM, may require different choices on large-scale systems. The design space also facilitates comparison among the different existing algorithms, and shows how new design choices might be constructed from existing ones.
4. We evaluate several design choices in the context of Cluster-STM. Our results show that two different choices are indeed appropriate on such clusters compared with cc-STM. First, reader locks perform acceptably even though they can degrade performance on cache coherent systems by causing cache evictions. Second, write buffering performs acceptably (and in our experiments performed competitively with undo logging), whereas previous researchers have concluded that undo logging is necessary for good performance. Both of these results flow from the fact that on a cluster, the remote operations dominate the latency.

The rest of this paper proceeds as follows. In Section 2, we discuss prior work on PGAS programming and software transactional memory. In Section 3, we present the interface design for Cluster-STM. In Section 4, we describe the algorithmic choices that we investigated for implementing the interface. In Section 5, we present our experimental results. In Section 6, we discuss related work and conclude.

2. Background and Related Work

In this section, we briefly discuss partitioned global address space (PGAS) languages, which provide the context for our work. We then compare our work with previous work on software transactional memory design and evaluation.

2.1 Background: PGAS Programming

The partitioned global address space (PGAS) model, embodied in languages such as Unified Parallel C (UPC) [12], Co-Array Fortran [28], and Titanium [38], has recently emerged as an attractive way to program large parallel systems, including commodity clusters, the focus of our work here. PGAS represents a middle ground between a "pure shared memory" style emulated in software, such as software distributed shared memory (SDSM) [2], and an explicit message passing style, such as MPI. Like shared memory programming, and unlike MPI, PGAS languages provide a global address space, allowing threads to refer to remote memory directly, instead of via message passing calls. However, by allowing the programmer to control locality and communication, the PGAS model also provides better performance and scalability than SDSM. None of these traditional PGAS languages supports transactional memory, but one could extend them to include transactional concepts like an `atomic{...}` block.

The DARPA HPCS languages — Chapel [8], Fortress [37], and X10 [9] — build upon the PGAS model and aim to provide the next generation in high performance programming languages. Like the traditional PGAS languages, they are based on a partitioned global address space, but they also provide many higher-level features, including flexible task creation, a rich set of parallel operations on global arrays, and object-oriented programming. Both Chapel and Fortress provide an `atomic{...}` block, which guarantees that code enclosed within the block will behave like a transaction. (X10 also provides an `atomic` construct, but it is intended for short critical sections and requires that the accessed memory locations be statically known and local to the executing thread.) How to implement this construct, and its performance implications for large-scale computing, are open questions that we address in this work.

2.2 Cache Coherent STM

The concepts of transactional processing [13], the property of serializability [29], and practical techniques to guarantee serializability [13, 21, 4] were originally developed in the database community. In the mid-1990s, researchers started applying transactions to general purpose programming, first as *hardware transactional memory* [18] and then as *software transactional memory*, or STM [34]. With the advent of multicore computing, in recent years there has been a great deal of interest in transactional memory, for the reasons explained in the introduction. Many designs have been proposed, either purely in software [1, 32, 15, 16, 10, 23, 17, 24, 14] or with hardware support [7, 20, 30, 5, 36, 27]. The focus of our work is on software designs for implementing transactional memory, and in particular doing so on large scale systems.

To date, most work on STM has focused on cache-coherent shared memory machines, either SMP (including the emerging class of multicore desktop machines) [32] or cache-coherent NUMA machines [15]. Manassiev et al. describe a transactional system designed for clusters [22], but their approach is based on SDSM, uses a kind of broadcast-based software cache coherence, and has not been shown to scale beyond about ten processors. Herlihy and Sun describe a design for a distributed memory STM, also based on global coherence [19]. We are aware of *no* work on STM that is directly applicable to the context of a distributed memory system containing thousands of nodes.

In Section 4.1, we describe how to decompose the set of existing cc-STM designs into a set of (mostly orthogonal) design choices, which can be viewed as axes in the design space, so that different algorithms can be viewed as combinations of such design choices. While there has been some prior work in this spirit [23, 24, 32, 11], no prior work has recognized all these dimensions explicitly (though they are implicit in the designs), nor has there been any evaluation of their impact on large-scale clusters.

3. STM Interface Design

Cluster-STM consists of an interface and an algorithm to implement that interface. In this section we describe our interface design. As in prior STM work, we provide an STM API as a set of low-level C functions that could be targeted by a compiler when translating

```
increment(proc_t proc, int *addr) {
  tmp = get(proc, addr)
  ++tmp
  put(proc, addr, tmp)
}
```

(a)

```
increment(proc_t proc, int *addr) {
  on (proc) {
    ++*addr
  }
}
```

(b)

Figure 1. (a) Remote increment without `on` construct; (b) remote increment with `on` construct. Potential remote operations are marked in bold. Assuming that `proc` is remote to this code, then (a) incurs two remote communication events (get and put), while (b) incurs one (spawning the remote operation).

higher-level constructs like the `atomic` block provided by Chapel and Fortress.

3.1 Execution Model

We assume a Single Program Multiple Data (SPMD) execution model similar to the traditional PGAS languages, in which one execution context is started for each processor at the outset of the program, and each runs until the end of its computation. No other execution contexts are created or destroyed during the life of the program. We assume that every data element (a scalar or an array element) is assigned a "home" processor, and not replicated automatically on any other processors. The application is responsible for caching local copies of remote values explicitly. Further, we treat the processors as a flat set, and do not distinguish between, e.g., processors within a node and processors across nodes. Unlike most SPMD models, we do allow an execution context on one processor to invoke operations on another processor to exploit data locality. Note, however, that we retain the limitation of p dynamic tasks for p processors. Chapel and Fortress support the ability to create more than p tasks, so our future work will include relaxing our execution model to support more general multithreaded programming.

3.2 Transactional Semantics

Cluster-STM provides *weak atomicity* [25]. This means that we guarantee atomicity only as to conflicting pairs of transactional accesses; conflicts between transactional and non-transactional accesses are not protected. Weak atomicity is less composable than strong atomicity (protecting all pairs where at least one is a transactional access). It also raises subtle problems, e.g., granular lost updates. However, the runtime overhead of strong atomicity can be prohibitively high in the absence of hardware support [25, 35].

In order to avoid the subtle problems of weak atomicity in our STM, we impose the *programming restriction* that each memory location must be accessed always within transactions or always outside transactions; we refer to these as *transactional* and *non-transactional* data. This restriction is only required within each interval between global synchronization points (e.g., global barriers); a memory location can be transactional in one such interval and non-transactional in another. The global barrier ensures that all threads see consistent behavior of the location throughout the execution.

With this programming restriction, Cluster-STM guarantees *serializability* [29] of reads and writes to transactional data. A transactional execution is serializable if the reads and writes can be interchanged into some serial order (so that there is no interleaving among transactions) without interchanging any conflicting operations (i.e., pairs of overlapping accesses such that at least one is a write) [4]. The Cluster-STM interface specifies that any implementation of the interface must guarantee serializability; it does not specify the algorithm for achieving this guarantee. We explore several algorithmic possibilities in the next section.

One potential complication of the above programming restriction is that certain STM algorithms may detect sharing conflicts at a granularity greater than one word. We call this unit of detecting sharing conflicts the *conflict detection unit*, or CDU (defined more precisely in Section 4). The complication is that an entire CDU must meet the programming restriction, i.e., must be always transactional or always non-transactional. To avoid exposing this internal algorithmic parameter to programmers, we impose the restriction that *every allocated object* be entirely transactional or entirely non-transactional. The compiler and run-time can ensure that distinct allocated objects never share a single CDU (unless the compiler can prove that two such objects are both transactional or both non-transactional). In practice, the CDU will be a few words, which should keep the memory penalty small.

3.3 Design Principles

In addition to providing the usual STM constructs (start, commit, read, write, and allocate memory), we based our design on several core principles motivated by the desire for high performance on clusters:

Multiword data movement is important. cc-STM designs are based on a shared heap-private stack model of memory in which (1) the heap is protected by STM metadata; (2) each transaction allocates private variables on the stack for local computations; and (3) transfers between (1) and (2) are done using single-word or single-object `stm_read` and `stm_write` operations [1, 32, 15, 16, 17]. On clusters, however, fine-grained remote reads and writes are usually inefficient, and transfers from remote shared to local private memory must be done in bulk where possible. Therefore, our design should support bulk transfers of data to and from the transactional store.

A transactional "on" construct is important. Multiword data transfers allow the programmer to "bring the data to the computation." However, it is often also important to be able to "send the computation to the data." For example, consider the code in Figure 1(a). If the destination processor is remote to the processor running the code, this code is inefficient, because it does a fine-grained get followed by a fine-grained put. The code in Figure 1(b) is written with an "on" construct specifying that the code should be incremented "in place" on the processor where the data is, rather than moving it back and forth to increment it. The Chapel language specification includes an "on" construct for exactly this purpose. We believe that the "on" construct is important in high-performance programming on clusters, and that our STM should support a transactional version of "on," so that `on` sections may be freely combined with `atomic` sections. In Section 5, we give examples of such combinations, and we show how the `on` construct is essential to getting good performance in a real application.

Any software cache should be orthogonal to the STM. On a cluster, there is often data reuse such that remote data should be cached locally for good performance. This caching must be done in software. Our initial STM design integrated a transparent software cache into the STM such that, when a remote value was read, it would be cached for later use for the life of the transaction. However, we realized that this design complicated the implementation and did not necessarily provide what is wanted: while conve-

Table 1. STM Interface

Function	Explanation
Memory Allocation	
`stm_all_alloc(work_proc, size)`	Allocate a pointer to `size` bytes on `work_proc` and return the pointer to all processors.
`stm_alloc(src_proc, work_proc, size)`	On behalf of `src_proc`, allocate and return a pointer to `size` bytes on `work_proc`.
`stm_free(work_proc, addr)`	Free the memory allocated at address `addr` on processor `work_proc`.
Transaction Start and Commit	
`stm_start(src_proc)`	Start a transaction on behalf of `src_proc`.
`stm_commit(src_proc)`	Commit a transaction on behalf of `src_proc`.
Data Movement	
`stm_open_read(src_proc, addr, size)`	On behalf of `src_proc`, open `size` bytes starting at `addr` for reading.
`stm_read(src_proc, dest, src, size)`	On behalf of `src_proc`, copy `size` bytes from transactional memory `src` to private memory `dest`.
`stm_open_write(src_proc, addr, size)`	On behalf of `src_proc`, open `size` bytes starting at `addr` for writing.
`stm_write(src_proc, dest, src, size)`	On behalf of `src_proc`, copy `size` bytes from private memory `src` to transactional memory `dest`.
`stm_get(src_proc, dest, work_proc, src, size, open)`	On behalf of `src_proc`, copy `size` bytes from transactional memory `src` on processor `work_proc` to private local memory `dest`. If `open==true`, open the source data for reading first.
`stm_put(src_proc, work_proc, dest, src, size, open)`	On behalf of `src_proc`, copy `size` bytes from `src` private local to transactional memory `dest` on processor `work_proc`. If `open==true`, open the destination data for writing first.
Remote Work	
`stm_on(src_proc, work_proc, function, arg_buf, arg_buf_size, result_buf, result_buf_size)`	On behalf of `src_proc`, perform function `function` on processor `work_proc`, using the argument in `arg_buf` and placing the result if any in `result_buf`. Return the number of bytes written to `result_buf`.

nient in many cases, transparent caching can also be less efficient than programmer-controlled locality management. Therefore, we decided that the software caching policy should be up to the application programmer, compiler, or a runtime layer above the STM. The STM is responsible only for guaranteeing serializability, and it should work with a variety of software caching protocols.

3.4 Interface Description

Table 1 illustrates the design of our STM interface using the assumptions and design principles stated above. Two of the parameters require explanation:

- *src_proc*: The processor on behalf of which the operation is occurring. This parameter is needed because, e.g., processor P may spawn some remote transactional work on processor Q using the `on` construct. The STM algorithm must log updates and check conflicts on behalf of P. To enable this, we pass P as an argument to the transactional API functions called by Q. This is similar to the technique of passing a thread-local transaction descriptor into each function called within a transaction [32]. We extend this technique to remote procedure invocation.
- *work_proc*: The processor on which the work is to be done. In the case of `get` and `put`, `work_proc` is the other processor involved in the transfer. In the case of `on`, `work_proc` is the processor where the work is to be done. If `work_proc` is equal to the processor invoking the function, then the operation is local and may be done without any remote communication. For operations that are always local (such as `stm_start` and `stm_commit`; see below), this parameter is omitted.

All API functions are *blocking*, i.e., a transaction must wait until the call returns to continue. In particular, no processor may create parallelism inside a transaction using these operations. This means that the API does not allow overlapping communication and computation within a transaction. We have found that this limitation does not impose much of a performance penalty for the benchmarks we have studied. Allowing nonblocking puts, gets, and `on` work within a transaction complicates the API semantics and is a subject of future work.

Memory allocation. We provide a function `stm_alloc` that is called by one processor and returns a pointer to allocated data to that processor only, initializing the associated transactional metadata. This function may be called inside a transaction, in which case the allocation is logged on behalf of `src_proc`; if the transaction aborts, the allocation is undone.

We also provide a function `stm_all_alloc` for allocating from the transactional store outside a transaction. This function must be called by all processors, employs a barrier synchronization, and returns a pointer to the allocated data (allocated on processor `work_proc`) to all calling processors. It is an error to call `stm_all_alloc` within a transaction. This design captures a common and useful programming idiom (allocate a common block of memory at the outset of execution, then use it repeatedly across transactions). If more general shared object creation is needed, one can allocate on one processor using `stm_alloc` and then manually communicate the address to other processors using communication and synchronization primitives.

Finally, we provide an `stm_free` function for freeing transactional memory allocations. This function may be called by any processor, but it must be called exactly once for each allocated object; in practice, it is usually called by the `work_proc` as a local operation.

Transaction start and commit. Transactions are started with `stm_start` and ended with `stm_commit`. A transaction must be started and committed on the same processor. All transactional operations must occur within a dynamically matched pair of these operations. Transactions may be nested, but in the present design we flatten nested transactions into a single transaction, treating only the dynamically first `stm_start` and dynamically last `stm_commit` as significant. Our interface has no explicit `abort` operation; any abort is implicit and is triggered by the conflict detection mechanism in the implementation of this interface.

Data movement. We provide both local and remote data movement operations. The local interface is similar to previously proposed STM interfaces [1, 16]. We provide `stm_open_read` and `stm_open_write` operations that open a contiguous set of memory locations for reading or writing. We also provide `stm_read` and `stm_write` functions for moving data between transactional and local private memory. Each transactional location touched by `stm_read` must be opened for reading at least once prior to the read, and each location touched by `stm_write` must be opened for writing at least once. Decomposing the interface in this way allows a single `stm_open` operation to protect multiple subsequent read or write operations, eliminating redundant operations.

We also provide one-sided put and get operations similar to those provided by GASNet [6], except that they are supported by logging and conflict detection necessary to guarantee serializabil-

(a)

```
increment(proc_t proc, int *addr) {
  atomic {
    on(proc) {
      ++*addr
    }
  }
}
```

(b)

```
increment(proc_t proc, int* addr) {
  stm_start(MY_ID)
  stm_on(MY_ID, proc, increment_local,
         addr, sizeof(int*), 0, 0)
  stm_commit(MY_ID)
}

increment_local(proc_t src_proc,
                void* arg,
                size_t arg_size,
                void *result,
                size_t result_size) {
  int *addr = *((int*) arg);
  int tmp;
  stm_open_read(src_proc, addr, sizeof(int))
  stm_read(src_proc, &tmp, addr, sizeof(int))
  ++tmp;
  stm_open_write(src_proc, addr, sizeof(int))
  stm_write(src_proc, addr, &tmp, sizeof(int))
}
```

Figure 2. The remote increment example of Figure 1 in Chapel-like pseudocode, implemented using the `atomic` and `on` constructs (a) and translated to our STM interface (b). STM API functions are marked in bold.

ity. The put and get operations may be on any number of contiguous bytes (up to some implementation-defined maximum) and may be local or remote. Operations known to be local should be done with `stm_read` or `stm_write` for performance. The function `stm_put` moves data from local memory on the processor invoking the function to transactional memory on the destination processor, and the function `stm_get` does the reverse. In each case, the remote data may be optionally opened first, to avoid the overhead of a separate remote open.

Remote work. We provide an `stm_on` construct for performing a user-defined function on the specified processor. The `stm_on` function may be called within a transaction, and it may start its own transactions. In both cases, the `src_proc` parameter of `stm_on` specifies on behalf of what processor the operations are occurring. Any operations inside the `stm_on` function that access transactional memory must be done with the appropriate transactional functions given in Table 1. Nested `stm_on` constructs are possible; in this case the `src_proc` parameter is threaded through the sequence of `stm_on` calls, identifying the processor that started the sequence. `stm_on` reads its arguments from and writes its output to local private memory.

Figure 2 illustrates the implementation of the remote increment function using our STM interface. Note that `increment` starts a transaction and uses `stm_on` to invoke `increment_local` on `proc` on behalf of the execution processor. Note also that `increment_local` does some work on behalf of the caller (represented by `src_proc`). All the work done here is local to `proc`.

4. Algorithm Design

In this section we describe the algorithmic choices that we investigated for implementing the interface described in the previous section. We first discuss the design choices motivated by previous work on cc-STMs. Then we discuss choices specific to PGAS computing and clusters.

4.1 Choices from the cc-STM Design Space

A key part of designing the Cluster-STM algorithm was to make suitable design choices that researchers have already considered for cc-STM, but which may require a different choice on large-scale clusters. To do this systematically, we studied the cc-STM literature and distilled the space of design choices represented there into eight dimensions. We used this study to ascertain which conclusions from the cc-STM research were likely to carry over to Cluster-STM, and which deserved further investigation in the new domain. A secondary benefit is that many current cc-STM designs, including at least those described in [17, 15, 14, 24, 23, 1, 32, 16], can be represented as a combination of choices from the eight axes, i.e., are points within this design space. Below we describe each of the eight dimensions, the choices made by the previous cc-STM designs in each dimension, the choices we made for our implementation within that dimension, and the motivations for our choices.

1. Transactional view of the heap. "Word-based" STMs [1, 32, 16, 15] read and write data words directly on the heap, while "object-based" STMs [23, 17, 24, 14] use an extra level of pointer indirection. The object-based design enables elegant synchronization mechanisms but requires that all data be accessed as objects. For our Cluster-STM we chose a word-based design because of its stronger support for numerical and array-based computations.

2. Read synchronization. Most STMs use "read validation": when transaction A reads location x, then transaction B writes x before A commits, A detects the violation at commit time and aborts [15, 32, 16]. An alternative is to use concurrent-read exclusive-write (CREW) locks to prevent B from writing [32]. Previous researchers have concluded that, for cc-STM, locks are worse than read validation because the lock operations create extra cache misses and hence potentially significant scalar overhead [15, 32]. However, in a distributed memory context, the remote lock operations can be piggybacked on the explicit messages required to access remote data, lessening or eliminating the penalty of reader locks. Further, read validation requires an extra remote validation at commit, which may increase overhead. For Cluster-STM, we implemented both read locking and read validation to quantify these tradeoffs.

3. Write synchronization. In most STMs, a transaction T writing to location x attempts to acquire x for exclusive access, and either the writer or the lockholder is aborted in the presence of contention [1, 32, 23, 17, 24, 16]. Alternatively, the writing transaction can maintain both "before" and "after" state for the written location, so other transactions may read the value while T is holding it [15, 14]. For Cluster-STM, we implemented exclusive-access acquire, because the single-writer, multiple-reader implementation is complicated, and its benefit has not been proved. As future work, it may be interesting to study the effects of these two design choices, both for cc-STM and Cluster-STM.

4. Recovery mechanism. Some STMs use a *write buffer*: Each transaction buffers its writes until commit, so no global state is

changed until the transaction can definitely commit [15, 17, 24, 14]. Others use an *undo log*: Each transaction makes its writes in place, saving the old value to be restored if necessary on abort [1, 32, 16]. Because it provides "invisible writes," the write buffering mechanism allows more flexibility in the timing of acquiring locations for writing (see immediately below). However, it is also potentially less efficient, because (1) it requires the write buffer to be searched on every read; and (2) it requires that written locations be copied on commit, instead of abort, and we want commit to be fast.

In Cluster-STM, we implemented both write buffering and undo logging. We expected that the additional overhead of write buffering would be less dramatic in cases where the runtime is dominated by remote access latencies, so long as the write buffering and undo logging implementations use the same number of remote operations, and the only extra overhead of write buffering is local scalar overhead (searching and copying).

5. Time of acquire for write. In implementations that use a write buffer, a transaction may acquire a location for writing at the time of the write [1, 32, 16, 17, 24], or it may acquire the location at commit time [14, 15, 24]. Acquiring locations later in the transaction may allow more concurrency, or it may allow doomed transactions to spend more time doing wasted work. The tradeoffs between write-time and commit-time acquire have been studied for cache coherent machines [23, 24, 11]. For Cluster-STM, we implemented both early and late acquire in our write buffering implementation.

6. Size of conflict detection unit (CDU). In previous work, access conflicts are detected for objects [32, 16, 17, 24, 14], cache lines [32], or groups of words associated by a hash function on their addresses [15]. The tradeoff here is between reducing false sharing (with finer granularity detection) and reducing the memory overhead imposed by the STM metadata (with coarser granularity detection). To test this effect, we use the same STM metadata to guard contiguous blocks of 2^n words, where $n \geq 0$ is a parameter initialized at startup. We call each such contiguous block a *conflict detection unit*, or CDU.

7. Progress guarantee. All STM designs avoid deadlock by aborting transactions that get stuck. Some provide no additional progress guarantee, e.g., they do not guarantee against livelock or guarantee progress when a lockholding thread is preempted or fails [1, 32, 16]. Others are *obstruction free*: they guarantee that some thread will make progress *unless* livelock is occurring [15, 17, 24]. One design is *lock free*, which means that some thread always makes progress (livelock never occurs) [14]. Increasing the level of progress guarantee increases the implementation complexity and, while beneficial in theory, can actually degrade performance in some cases [15, 32]. For Cluster-STM, we provide no progress guarantee except absence of deadlock for simplicity of implementation, and because we expect preemption or failure of lockholding threads to be rare.

8. Where metadata is stored. All STMs store and maintain metadata for detecting conflicting accesses to program data by different transactions. This metadata may be stored in program data objects [16, 32, 1], in transaction descriptors [14], or in side data structures [32, 1, 15, 17, 24]. Cluster-STM employs a novel strategy that uses a transaction descriptor distributed across multiple processors, plus one word of globally shared metadata per CDU.

To summarize, our algorithm is word-based (1), uses exclusive access for writes (3), gives no progress guarantee other than absence of deadlock (7), and uses a novel distributed metadata organization (8). The CDU size (6) is a numerical parameter to the algorithm. For the remaining three axes, we implement both choices in each: read locking (RL) and read validation (RV) (2); write buffering (WB) and undo logging (UL) (4); and early acquire (EA) and late acquire (LA) (5). This creates eight possible combinations. However, late acquire cannot be used with undo logging, leaving six feasible variations of our algorithm.

4.2 Choices Motivated by Clusters

4.2.1 Distribution of metadata onto the processors

One important decision we faced was how to distribute the STM metadata onto the processors. This design decision is not addressed in prior work on cc-STM, because the cache-coherent model assumes efficient access to all memory.

Globally shared metadata. As discussed in the previous section, we reserve one word of globally shared metadata per CDU, where a CDU represents 2^n contiguous words on the transactional store, and n is a user-defined parameter. In all implementations, the least significant bit of the word is set if the corresponding CDU is locked for writing, clear otherwise. The other bits represent (1) the number of readers, in the read locking implementation when the write bit is clear; or (2) the version of the associated CDU, in the read validation implementation.

It is natural to keep this metadata on the processor where the corresponding program data is. Therefore, we reserve the top $1/(2^n+1)$ of each processor's transactional store for metadata, and we map the ith CDU in the store to the ith word in the metadata section.

Transaction-local metadata. Conceptually, each active transaction maintains a transaction descriptor with the following information:

- A *dataset* containing one entry for each CDU read or written by the transaction. The entry contains information such as whether the access was a read or a write; the data being buffered or logged for the CDU; and the old version read (in the read validation implementation).
- The transaction nesting depth and information on where to return on abort (in our implementation, we use `setjmp`/`longjmp` and store this information in a `jmp_buf`).
- A list of transactional allocations, for undo on abort.

Our initial idea was to locate this metadata on the processor that initiated the transaction. However, we realized this would cause unnecessary network traffic, make the implementation unduly complex, or both. For example, to implement `stm_on`, we would either need to send metadata to the initiating processor after each remote operation, thus defeating the purpose of the `on` clause, or we would need some way to marshal the metadata at the end of the remote execution and send it back. Following the lead of distributed database implementations [4, 31], we realized it would be easier and more efficient to keep *all* the metadata associated with a CDU of program data on the home node for that data (defined in Section 3.1).

For a transaction T initiated on processor P, we slice T's descriptor across all the processors on which P initiates memory, allocation, or other local operations. Each processor maintains a transaction descriptor for every processor that invoked operations on it. To make this scheme work, we added the following data to each transaction descriptor slice:

- `src_proc`: the processor on behalf of which the descriptor is recording information
- `tx_proc`: the processor that initiated the currently active (outermost) transaction
- `remote_procs`: the set of remote processors whose memory `src_proc` has accessed in executing the transaction on this processor.

Note that the *only* case where `src_proc` is different from `tx_proc` is if a processor (`src_proc`) initiates an `on` operation (outside any

Table 2. Cluster-STM Algorithm

Vertically stacked operations show a sequence of steps by a processor.
RL = *read locking,* EA = *early acquire,* UL = *undo logging,* RV = *read versioning,* WB = *write buffering, and* LA = *late acquire. See Section 4.1.*

Operation	RL-EA-UL	RV-EA-UL	RL-EA-WB	RL-LA-WB
		Memory Allocation		
`stm_alloc`	Call `memalign` to get an allocation aligned on a CDU boundary and clear associated metadata. If `tx_depth > 0`, record the allocation.			
`stm_all_alloc`	Same as `stm_alloc`, except never inside a transaction and the result is broadcast to all processors.			
`stm_free`	Call `free`			
		Transaction Start and Commit		
`stm_start`	Increment `tx_depth`. If `tx_depth == 1`, set `tx_proc` to the current processor and invoke `setjmp`.			
`stm_commit`	Decrement `tx_depth`. If `tx_depth == 0`, then attempt to commit the transaction:			
		Validate READ CDUs, aborting on failure. Increment version of WRITE CDUs.		Acquire WRITE CDUs, aborting on failure.
			Copy data from WRITE entries	
	Release all held locks			
		Data Movement		
`stm_open_read`	For each CDU touched that is not already in the dataset, create a new READ entry and do the following atomically with respect to the associated global metadata word, aborting if the word is locked for writing:			
	Increment reader count	Record version	Increment reader count	
`stm_read`	Copy data from transactional store to private store		Copy data from buffer if there, otherwise from store	
`stm_open_write`	For each CDU touched that does not already have a WRITE entry in the dataset, do the following atomically with respect to the associated global metadata word, aborting if the word is locked for writing:			
	Abort if reader count exceeds one, or equals one and CDU is not in read set	If a READ entry exists, validate it and abort on failure (since the transaction would later abort on commit anyway).	Abort if reader count exceeds one, or equals one and CDU is not in read set.	
			Get data from store into write buffer.	
	Set write bit of metadata word			
`stm_write`	Copy from private store to transactional store		Copy from private store to buffer	
`stm_get`	If `work_proc` is remote, add `work_proc` to `remote_procs`. Optionally invoke `stm_open_read` on `work_proc`, aborting on failure. Invoke `stm_read` on `work_proc`.			
`stm_put`	If `work_proc` is remote, add `work_proc` to `remote_procs`. Optionally invoke stm_open_write on `work_proc`, aborting on failure. Invoke `stm_write` on `work_proc`.			
		Remote Work		
`stm_on`	If `work_proc` is remote, and `tx_depth > 0`, add `work_proc` to `remote_procs`. Invoke `setjmp`. Invoke specified function on `work_proc`, aborting on failure.			

transaction) and during the execution of that operation a different processor (`tx_proc`) initiates an outermost transaction.

A simple way to implement the slicing is to have each processor keep an array of descriptors, containing one descriptor for every other processor. This is sufficient (and no synchronization is required for these arrays) even in the presence of nested operations because all remote operations, including `stm_on`, are blocking, and therefore two transactions will never compete for the same slice on any processor. Also, since the size of an initialized descriptor is small, and only grows when the descriptor is actually in use by a transaction, the memory overhead of this approach is acceptable. We could also create descriptors on an as-needed basis and store them in a hash map.

4.2.2 Algorithm design

Table 2 summarizes our algorithm design for four out of the six feasible configurations described in Section 4.1 (for lack of time and space, we have not implemented the other two configurations, RV-EA-WB and RV-LA-WB). The table shows, for each interface function in each configuration, the main steps that are carried out. Using these algorithm descriptions, it is straightforward to show that each variant guarantees serializability of transactions by placing a *serialization point* [29] between the lock operations and the unlock operations (for the reader lock implementations) or between the lock operations and the validate operations (for the read validation implementations).

For memory allocations, we reserve a transactional store on each processor at the outset of the program, and manage it using `memalign` and `free` protected by a lock. We also retain the "normal" program heap (managed with `malloc` and `free`) for processor-local computations.

Every transaction maintains the nesting depth `tx_depth`, incrementing and decrementing it on start and commit. `stm_start` sets `tx_proc` if the depth becomes 1, denoting an outermost transaction. The matching `stm_commit` sees the depth become 0 and attempts to commit, which requires sending one or two commit messages to each processor in `remote_procs`. In the RL-EA-UL case, each processor only has to release all read locks that are held; only a single commit message is necessary. In the RV-EA-UL case, one message notifies all participating processors to validate their READ CDUs and increment version numbers for WRITE CDUs; if all respond as successful, a second message notifies all to release all held write locks. The `stm_put/get/on` operations all add the target processor to `remote_procs` to be notified during a subsequent commit of the outermost transaction, before performing the appropriate operation. Both `stm_start` and `stm_on` execute a `setjmp` to prepare to receive control on a possible abort. When an abort occurs inside an `on` clause inside a transaction, control is transferred to the point of the `on` invocation via `longjmp` on the remote processor, then to the invoking processor via handler return, then to the start of the transaction via `longjmp` on the invoking processor.

Our design has several important properties:

We aggregate communication well. For the memory allocation, data movement, and remote work operations, the STM metadata operations introduce *no* extra communication events. `stm_start` is a purely local operation. `stm_commit` does introduce communication events, but our distributed metadata limits their number to one event per processor touched by the transaction, even if many locations are touched on each processor. The same is true on abort. The late acquire and read validation options each add one more event per processor touched on commit or abort.

We maintain one dataset. We maintain a single dataset that serves as our read and write set, undo log and write buffer. This

simplifies the implementation. It also allows us to promote readers to writers, by checking that the writing transaction is the only reader, in contrast to the more complicated approach taken in [32]. As pointed out in recent work [32, 16], the use of sets instead of lists may add local scalar overhead to the STM, but in this work we used a simple implementation because we were concerned primarily with the issues posed by remote data access. Tuning the local scalar part of the STM is a subject of future work.

We optimize block data moves. Our dataset stores one entry per CDU, in contrast to recent cc-STM designs that store one entry per word in the undo log, even when the conflict detection granularity is larger than one word [1, 16]. This allows us to use one dataset entry per CDU size in words, rather than one entry per word. In the write buffering implementation, it also reduces the number of buffer searches on read and write to one per CDU size.

5. Evaluation

To evaluate our design of Cluster-STM, we built a prototype in C on top of the GASNet communication library [6]. We assign one GASNet process at startup to each processor, and we use GASNet's Active Message capability to spawn remote `on` work. We ran our experiments on the NCSA Tungsten cluster comprising 1280 nodes (each of which is a Dell PowerEdge 1750 server with two Intel Xeon 3.2 GHz processors), running Red Hat Linux and connected by Myricom's Myrinet cluster interconnect network. We ran the experiments using one processor per node, because we found this configuration was necessary to obtain stable runtimes. This configuration also generated faster runtimes than using two processors per node, probably because it eliminated resource contention between two processors on one node.

5.1 Micro Benchmarks

We ran experiments using two concurrent data structure benchmarks that help to clarify the overheads and scalability of the STM operations because they are dominated by these operations. They are similar to benchmarks used in previous STM papers [15, 23].

The first micro benchmark, intset, performs a sequence of random insert and find operations on a set of integers that is evenly distributed over the nodes. Each remote operation occurs entirely inside an `on` clause. In the locking implementation, a lock is taken and released on the remote processor inside that clause. In the STM implementation, a transaction is started and committed inside the clause.

Figure 3(a) shows a plot of the runtimes of intset implemented with distributed queued locks for 1–512 processors ($1 \leq p \leq 512$), with 16% inserts, taking the minimum of several runs. We used two problem sizes, a smaller one (6 million operations) for $p \leq 8$, and a larger one (100 million operations) for $p \geq 8$, because the larger problem size did not fit on 1–2 processors, and the smaller problem size generated runs that were too short for large p. The graph shows a "bump" at 2–4 processors, as the parallel speedup achieved by adding processors is offset by the increased cost of remote communication. After 4 processors, however, the runtime scales linearly with the number of processors.

Figure 3(b) shows the results for each of the four STM implementations with a CDU size of one word ($n = 0$). We have normalized the results so the runtime for the lock implementation is one, and the other STM runtimes are expressed as ratios to it. Note that while scalar overhead is significant (about 2.8x–3.7x) on one processor, the STM performance penalty all but disappears after $p = 2$, as the runtime becomes dominated by the remote access latency. Further, all four STM implementations show about the same performance. These results occur because all four STM implementations show about the same performance. This occurs because all four STM implementations use the same number of remote accesses (i.e., one `on` invocation per operation) and impose no extra remote operations over the lock version for this benchmark. Note also that . We also ran these experiments for a CDU size of 16 ($n = 4$). The results were similar, but the STM overhead was slightly higher because of the increase in false conflicts.

The second benchmark, hashmap swap, tests an atomic swap of the values associated with two randomly selected keys in a hash map. Again the data is distributed evenly over the nodes. The swap consists of two finds followed by two inserts, all in a single atomic section. In the locking implementation, we lock the two keys (in numerical order to avoid deadlock), perform the swap, and unlock the keys. In the STM implementation, we start a transaction and nest `on` clauses inside it for the remote finds and inserts. In the common case of swapping two remotely-stored values, the lock implementation uses eight separate remote accesses per swap operation (two each of lock, find, insert, and unlock). The RV-EA-UL and RL-LA-WB STM implementations also use eight (two each of find, insert, and commit, plus two validations for RV and two acquires for LA). However, the two RL-EA implementations use only *six* remote operations (two each of find, insert, and commit, avoiding the extra validation and acquire as discussed in the previous section). Thus, we expect the RV and LA STM implementations to perform about as well as the lock implementation, and the RL-EA STM implementations to outperform the other STM implementations (and locks) by about 6/8.

Figure 4(a) shows a plot of the runtimes for the lock implementation, and Figure 4(b) shows the STM results, again expressed as ratios to the lock runtimes. Again we use two problem sizes, for the reasons discussed above. The results are close to what we expect: the two RL-EA STM implementations significantly outperform locks, and the other two STM implementations do slightly better than locks. In both cases, after $p = 2$, the STM runtimes are slightly lower than we would predict by counting remote operations. This is likely because the locking implementation takes two remote locks even when both are on the same processor, whereas the STM implementation "automatically" consolidates accesses to the same processor. The locking implementation could be made more efficient, but at a cost to programmability. Again, we see that the significant scalar overhead for $p = 1$ (about 3x–4x) disappears after $p = 2$ as the runtimes become dominated by the remote access latency.

Finally, note that reader locks (RL) perform well in both benchmarks, and they outperform read validation (RV) in the hashmap swap benchmark, in contrast to the cc-STM literature reporting that read validation outperforms reader locks [15, 32]. Note also that write buffering (WB), when used with RL and EA, performs about as well as undo logging, again in contrast to the results reported for cc-STM [32, 16]. Both of these results illustrate the differences between the design spaces of cc-STM and STM on large-scale clusters.

5.2 SSCA2 Kernel 4

To evaluate our design on a larger, more realistic application, we used kernel 4 from Version 1.0 of the Scalable Synthetic Compact Application (SSCA) benchmark number 2 (Graph Analysis) [3]. SSCA2 kernel 4 is a graph clustering problem. The graph consists of n vertices grouped into random-sized cliques (i.e., sets of vertices such that every pair in the set is connected by a graph edge) of a specified maximum clique size. There are also random inter-clique edges (i.e., edges between pairs of vertices that are in different cliques). The point of the kernel is to find clusters in the graph, so there is no *a priori* knowledge of graph locality that could be used to distribute the graph onto the nodes for memory locality. This fact makes the application especially challenging for high performance on a cluster with high remote memory access latency.

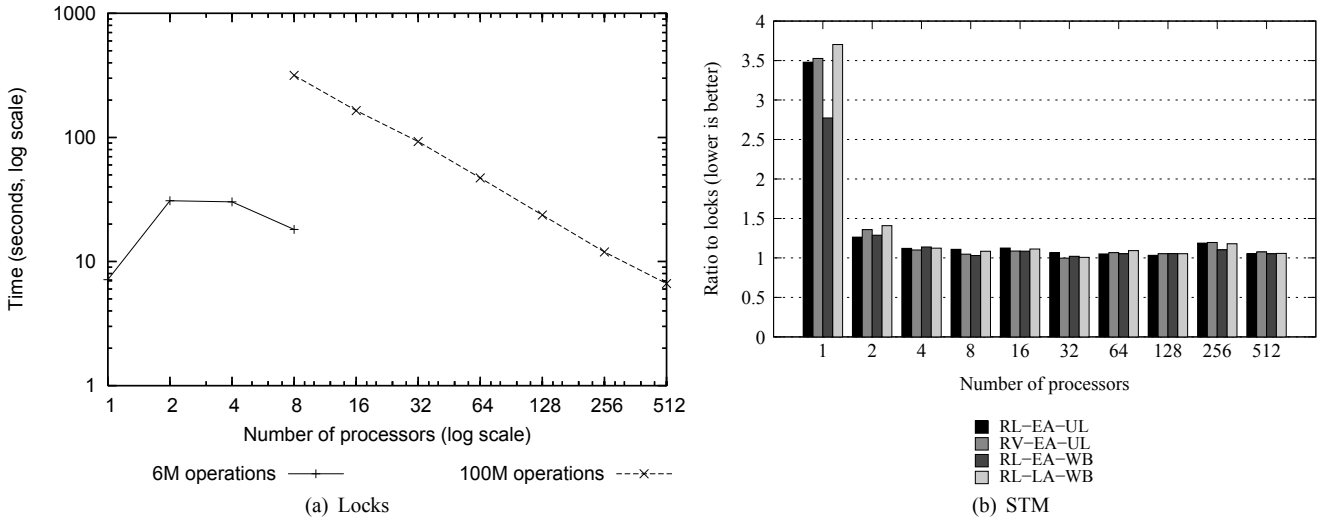

Figure 3. (a) Runtimes for intset implemented with locks. (b) Ratios of STM runtime to lock runtime for each STM implementation. The bar graph shows the results for the larger problem size at $p = 8$.

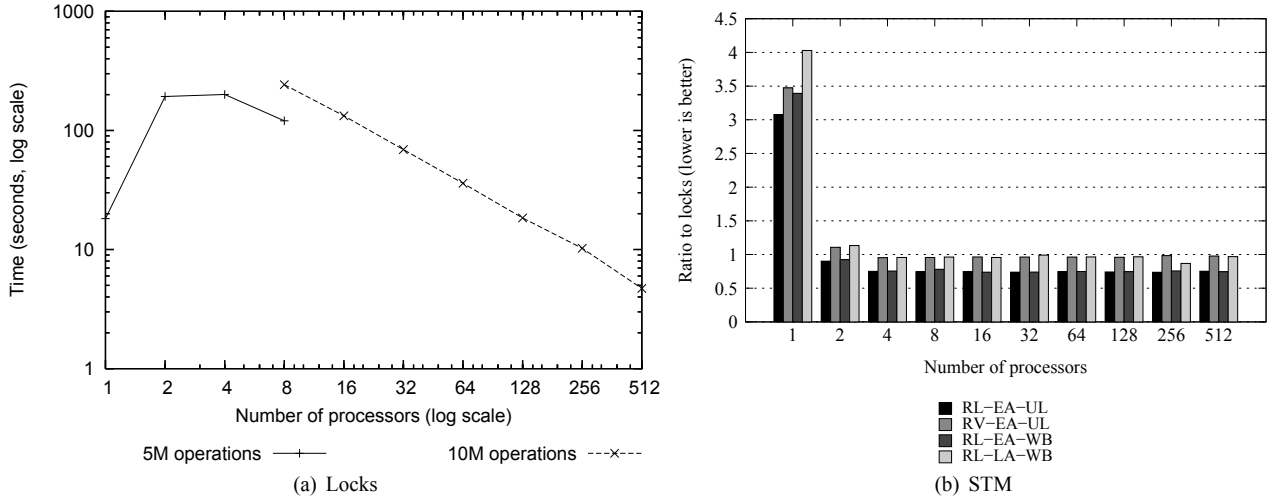

Figure 4. (a) Runtimes for hashmap swap implemented with locks. (b) Ratios of STM runtime to lock runtime for each STM implementation. The bar graph shows the results for the larger problem size at $p = 8$.

5.2.1 Algorithm Description

Our implementation of SSCA2 kernel 4 is based on the algorithm suggested in the benchmark specification [3]. The graph is stored in an adjacency list representation. Before the algorithm starts, the vertices of the graph and their adjacency lists are distributed evenly across the processors. In the explicit locking version, each processor iterates through its local vertices in parallel. For each vertex v, if v has already been claimed by another processor, the loop continues to the next iteration. Otherwise, the processor attempts to lock v and its adjacent vertices. If the locking is successful, the processor uses a heuristic to pick one of the adjacent vertices as the next candidate for inclusion in the cluster. It then repeats this process, locking all the adjacent vertices and picking one, until it has formed a cluster of sufficient size. If at any point the locking fails, the processor unlocks all its held vertices, discards the cluster it was building, and retries the original start vertex v.

The Cluster-STM version is similar, except that there is no explicit locking; instead, the synchronization is handled by the transactional implementation. Each loop iteration is a transaction. If the start vertex is claimed, the loop moves on to the next vertex, as before. Otherwise, the processor builds its cluster, reading and writing the shared graph data. If these reads and writes cause a conflict between two processors, the STM framework detects it and rolls back the iteration. We have aggressively privatized the computation so that only shared data is accessed by the STM operations. Purely local operations (such as computing the cluster once all the adjacency sets have been pulled into cache) are done in private memory with no STM overhead. This technique keeps the overall STM overhead reasonable.

In both the STM and locking versions of SSCA2 kernel 4, we carefully tuned the program to avoid fine-grained remote data access. For example, when a processor is reading the adjacency sets

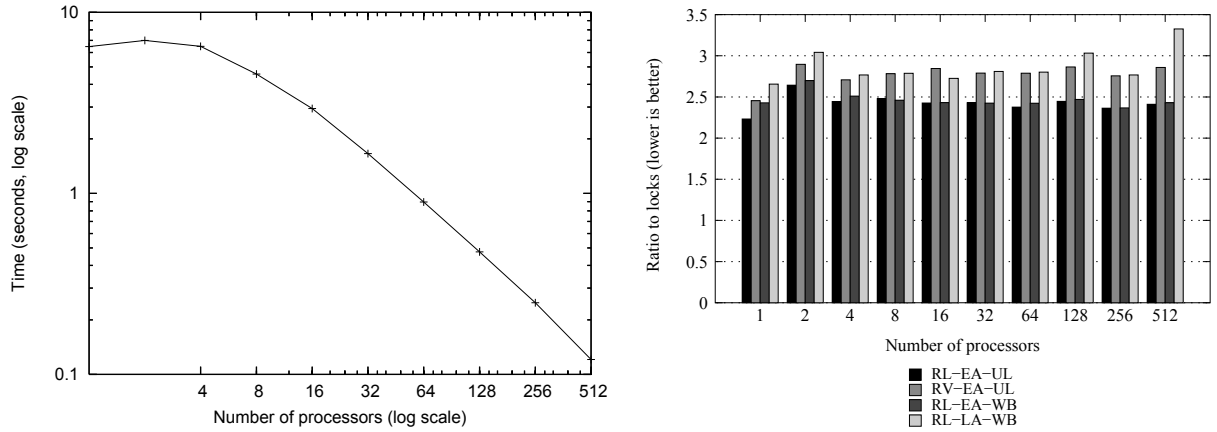

Figure 5. (a) Runtimes for SSCA2 kernel 4 implemented with locks. (b) Ratios of STM runtime to lock runtime for each STM implementation.

of candidate vertices, the straightforward implementation is to send out one message per vertex. However, this incurs unacceptable communication cost. In our version, we sort the vertices by processor, marshal the data for all vertices on one processor into a buffer, and then send a single request to get the adjacency sets of all the vertices via the `on` clause discussed in Section 3.4. Therefore, the on clause was essential to getting good performance on this benchmark.

5.2.2 Performance Results

We ran our SSCA2 kernel 4 implementations on a graph with 2^{20} vertices and a maximum clique size of 8. Figure 5(a) shows a plot of the runtimes for the lock implementation, for 1–512 processors. The overall shape of the graph is similar to the graphs for the micro benchmarks, except that the "bump" in execution time for small p is less dramatic, because this application is well tuned, with a better communication to computation ratio.

Figure 5(b) compares the STM versions, again expressed as ratios to the lock runtimes, using a CDU size of 4 ($n = 2$). Again, the RL-EA implementations perform better, for the reasons discussed above in connection with the hashmap swap benchmark. The STM overhead is lower than in the micro benchmarks for $p = 1$, because much of the computation is private and incurs no STM overhead. However, unlike for the micro benchmarks, the STM overhead remains significant for large p. We believe that this result occurs for two reasons. First, our SSCA2 is better tuned than our micro benchmarks, and wastes fewer cycles waiting for remote computation. Thus, less of the scalar STM overhead is hidden. Second, the lock implementation is able to fold the remote write and unlock operations into one message, whereas the STM version requires separate write and commit operations.

We may be able to improve the scalar overhead with more aggressive use of optimization techniques such as those discussed in [16, 1]. Hardware support could also be used to reduce the performance penalty of the local scalar operations. Because Cluster-STM uses at most two more remote operations than the locking implementation per transaction, the performance of a hardware supported Cluster-STM should approach that of locks. In any event, these graphs show excellent STM scalability to 512 processors for a realistic application with poor locality and demanding remote access patterns.

We also ran this experiment using the RL-EA-UL STM implementation for CDU sizes of 1, 2, 8, 16, and 32, for $p = 16$, 64, and 128. We observed that the runtime was extremely insensitive to CDU size in these experiments. We believe this result occurs for two reasons. First, in this application, *any* read-write or write-write sharing of a single allocation between transactions causes a "genuine" (semantically required) conflict. Second, because we allocate transactional data on a CDU boundary, no two allocations can share a CDU. Therefore, increasing the CDU size does not increase the number of conflicts for this benchmark.

6. Conclusion and Future Work

We have presented Cluster-STM, the first STM we know of explicitly designed for high performance on large-scale distributed architectures. Cluster-STM incorporates several novel features, including distribution of metadata and aggregation of computation, that allow it to execute STM operations with minimal overhead, measured in terms of remote accesses. We validate our design experimentally and show excellent scalability up to 512 processors. We also characterize the existing space of cc-STM designs and show that, on clusters, several design tradeoffs come out differently.

We have several plans for future work. In the algorithm itself, we would like to improve the scalar overhead and exploit shared memory within multiprocessor nodes to make intra-node communication faster. We would also like to test Cluster-STM with additional HPC workloads containing long and short transactions. Finally, we would like to add support for non-blocking remote operations inside a transaction, and for dynamic spawning of threads. These issues complicate both the semantic definition of the STM operations and the handling of distributed metadata in the STM implementation. However, they are important for supporting more general and dynamic parallel programming models.

Acknowledgments

The authors wish to thank Wayne Wong, Steve Deitz, and Mary Beth Hribar for their assistance with this work. We also wish to thank the National Center for Supercomputing Applications (NCSA) at the University of Illinois for granting time and support on the Tungsten cluster.

References

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *ACM Conf. on Prog. Lang. Design and*

Implementation (PLDI), pages 26–37, New York, NY, 2006. ACM Press.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

[3] D. A. Bader and K. Madduri. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. Technical report, Georgia Institute of Technology, Atlanta, GA, September 2005.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proc. Int'l Conf. on Comp. Arch. (ISCA)*, pages 24–34, New York, NY, USA, 2007. ACM Press.

[6] D. Bonachea. Gasnet specification, v1.1. Technical Report No. UCB/CSD-02-1207, University of California at Berkeley, October 2002.

[7] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In *ACM Conf. on Prog. Lang. Design and Implementation (PLDI)*, pages 1–13, New York, NY, 2006. ACM Press.

[8] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. In *Int'l J. High Performance Comp. Apps.*, volume 21, pages 291–312, Thousand Oaks, CA, USA, 2007. Sage Publications, Inc.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proc. ACM SIGPLAN Conf. on Object-Oriented Prog., Systs., Langs., and Apps. (OOPSLA)*, pages 519–538, New York, NY, 2005. ACM Press.

[10] C. Cole and M. Herlihy. Snapshots and software transactional memory. *J. Sci. Comp. Prog.*, 58(3):310–324, 2005.

[11] D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *Proc. Int'l Symp. on Code Gen. and Optimization (CGO)*, pages 21–33, Washington, DC, USA, 2007. IEEE Computer Society.

[12] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, June 2005.

[13] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[14] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003.

[15] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proc. ACM SIGPLAN Conf. on Object-Oriented Prog., Systs., Langs., and Apps. (OOPSLA)*, pages 388–402, New York, NY, 2003. ACM Press.

[16] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *ACM Conf. on Prog. Lang. Design and Implementation (PLDI)*, pages 14–25, New York, NY, 2006. ACM Press.

[17] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Symp. on Principles of Distrib. Comp.*, pages 92–101, New York, NY, 2003. ACM Press.

[18] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. Int'l Conf. on Comp. Arch. (ISCA)*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[19] M. Herlihy and Y. Sun. Distributed transactional memory for metric-space networks. In *Symp. on Distrib. Comp.*, volume 3724. Springer, Sept. 2005.

[20] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog. (PPOPP)*, pages 209–220, New York, NY, 2006. ACM Press.

[21] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. on Database Systs.*, 6(2):213–226, 1981.

[22] K. Manassiev, M. Mihailescu, and C. Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog. (PPOPP)*, pages 198–208, New York, NY, 2006. ACM Press.

[23] V. J. Marathe, W. N. Scherer, and M. L. Scott. Design tradeoffs in modern software transactional memory systems. In *Proc. Workshop on Langs., Compilers, and Run-Time Support for Scalable Systs.*, pages 1–7, New York, NY, 2004. ACM Press.

[24] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *Proc. Int'l Symp. on Distrib. Comp.*, Cracow, Poland, Sep 2005. Earlier but expanded version available as TR 868, University of Rochester Computer Science Dept., May 2005.

[25] M. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comp. Arch. Letters*, 5(2), 2006.

[26] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. *SIGARCH Comput. Archit. News*, 35(2):69–80, 2007.

[27] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proc. Int'l Conf. on Comp. Arch. (ISCA)*, pages 69–80, New York, NY, USA, 2007. ACM Press.

[28] R. W. Numerich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[29] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[30] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional nemory. In *Proc. Int'l Conf. on Comp. Arch. (ISCA)*, pages 494–505, Washington, DC, 2005. IEEE Computer Society.

[31] D. P. Reed. Implementing atomic actions on decentralized data. *ACM Trans. on Comp. Systs.*, 1(1):3–23, 1983.

[32] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog. (PPOPP)*, pages 187–197, New York, NY, 2006. ACM Press.

[33] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *Proc. ACM/IEEE Int'l Symp. on Microarch. (MICRO)*, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[34] N. Shavit and D. Touitou. Software transactional memory. In *Symp. on Principles of Distrib. Comp.*, pages 204–213, New York, NY, 1995. ACM Press.

[35] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. *ACM Conf. on Prog. Lang. Design and Implementation (PLDI)*, 42(6):78–88, 2007.

[36] A. Shriraman, M. F. Spear, H. Hossain, V. J. Marathe, S. Dwarkadas, and M. L. Scott. An integrated hardware-software approach to flexible transactional memory. In *Proc. Int'l Conf. on Comp. Arch. (ISCA)*, pages 104–115, New York, NY, USA, 2007. ACM Press.

[37] Sun Microsystems, Inc. The Fortress language specification, version 1.0 β. Technical report, Sun Microsystems, Inc., March 2007.

[38] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. *Concurrency – Practice and Experience*, 10(11-13):825–836, Sept-Nov 1998.