Future Technologies Group Technical Report Series

Future Technologies Group
Oak Ridge National Laboratory
Oak Ridge, Tennessee
USA

http://ft.ornl.gov

Scalable Software Transactional Memory for Global Address Space Architectures

Authors
Sridharan, S.
Vetter, J.S
Kogge, P.M

Abstract

This paper presents the challenges encountered in and potential solutions to designing scalable Software Transactional Memory (STM) for large-scale distributed memory systems with thousands of nodes. We introduce Global Transactional Memory (GTM), a generalized and scalable STM design supporting a dynamic programming model based on thread-level parallelism, Single Process Multiple Data (SPMD) parallelism, and remote procedure call invocation. In addition to this, GTM is the first STM design to support the non-blocking abstraction, i.e. the control returns immediately from the STM procedure call instead of waiting on its entire operation to finish. This allows remote STM operations to execute asynchronously with respect to the caller, thus tolerating remote communication latency and exposing new opportunities for parallelism within the context of software transactions. Our experimental results for Red-Black Tree, Priority Queue, and Bank Transaction workloads exhibit good strong and weak scaling attributes on a wide range of node counts (up to 1024 quad-code nodes) and problem sizes (up to one billion transaction commits). The results also demonstrate the benefits of leveraging asynchronous protocols for tolerating latency, providing additional speedups of 1.8X, 1.75X, and 1.27X or greater over their synchronous counterparts for these workloads.

Future Technologies Group Technical Report FTGTR-2009-04
Total Pages 10
Publication Date 04/1/2009

Also published as

Scalable Software Transactional Memory for Global Address Space Architectures*

Srinivas Sridharan
Department of Computer Science & Engineering
University of Notre Dame
ssridhar@nd.edu

Jeffrey Vetter
Future Technologies Group
Oak Ridge National Laboratory
vetter@ornl.gov

Peter Kogge
Department of Computer Science & Engineering
University of Notre Dame
kogge@cse.nd.edu

ABSTRACT

This paper presents the challenges encountered in and potential solutions to designing scalable *Software Transactional Memory* (STM) for large-scale distributed memory systems with thousands of nodes. We introduce *Global Transactional Memory* (GTM), a generalized and scalable STM design supporting a dynamic programming model based on thread-level parallelism, *Single Process Multiple Data* (SPMD) parallelism, and remote procedure call invocation. In addition to this, GTM is the first STM design to support the *non-blocking abstraction*, i.e. the control returns immediately from the STM procedure call instead of waiting on its entire operation to finish. This allows remote STM operations to execute *asynchronously* with respect to the caller, thus tolerating remote communication latency and exposing new opportunities for parallelism within the context of software transactions.

Our experimental results for Red-Black Tree, Priority Queue, and Bank Transaction workloads exhibit good strong and weak scaling attributes on a wide range of node counts (up to 1024 quad-code nodes) and problem sizes (up to one billion transaction commits). The results also demonstrate the benefits of leveraging asynchronous protocols for tolerating latency, providing additional speedups of $1.8X$, $1.75X$, and $1.27X$ or greater over their synchronous counterparts for these workloads.

1. INTRODUCTION

Transactional Memory (TM) [20, 24] extends the idea of database transactions to provide a generic concurrency control mechanism for simplifying the task of parallel programming and reasoning about correctness. Using TM, programmers identify sequences of operations that need to execute atomically, namely *transactions*, and delegate the choice of implementation mechanism to the compiler and/or hardware. Current research has successfully demonstrated numerous productivity benefits (performance, programmability, portability, and robustness) for using TM over lock-based approaches. However, much of this early TM work focussed on applications with processing and memory requirements that fit within the capabilities of a single node.

In contrast, a large number of legacy and emerging applications, such as those targeted by the DARPA *High Productivity Computing Systems* (HPCS) program [13], require that applications scale to thousands of processors in order to benefit from large aggregate memory and computational capabilities. Unlike most legacy applications, most emerging applications tend to be tightly coupled, data-intensive, and cannot be easily partitioned using Message Passing Interface (MPI). For example, graph [22] and combinatorial applications tend to have dynamic communication patterns predominantly with small messages and do not exploit locality as efficiently as some of the legacy codes.

These requirements have spurred a considerable amount of related work in programming languages and models: *Unified Parallel C* (UPC) [16], *Co-Array Fortran* (CAF) [28], Titanium [33], and the DARPA HPCS languages (Chapel [11], X10 [10], and Fortress [4]). Further, the research community now believes TM may provide additional productivity benefits for these large-scale applications. Not surprisingly, recent languages have proposed syntatic extensions for specifying transactions, such as Chapel's `atomic` construct, that maps directly to an underlying TM runtime system.

This work tackles some of the challenges encountered in designing scalable *Software-based TM* (STM) mechanisms for distributed memory systems with thousands of nodes. Early work in this area relied on software-based global cache-coherence protocols and did not scale beyond tens of nodes [21, 25]. More recently, researchers have proposed a STM design targeting *Partitioned Global Address Space* (PGAS) systems [5]. In general, these investigations have proposed a number of novel techniques for data/metadata distribution, communication aggregation, and locality management.

Yet, these designs tend to fall short in the following two aspects. First, current designs are not suitable for the newer HPCS languages since they cannot handle the dynamic multi-core multi-node parallelism supported by these languages.

*This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-ASC05-00OR22725.

In other words, STM designs for HPCS languages must be able to exploit multicore processors as well as scale to thousands of nodes. Second, current designs enforce a *blocking STM abstraction*. This implies, a process/thread issuing a STM request (e.g. transactional load/store) waits for the underlying runtime system to *synchronously* complete the data/metadata operations necessary to satisfy the request. Thus, the performance of STM applications targeting SMP/multicore systems is dominated by STM algorithmic overheads [9]. Rigorous compiler optimizations along with novel algorithmic and implementation strategies have helped improve the overall performance [1, 9]. Nevertheless, the baseline overheads of individual operations is still too large, mainly due to the synchronous execution behavior imposed by the blocking abstraction, for applications to benefit from STM-based approaches. On the other hand, in large-scale distributed memory systems, the cost of single-node STM metadata management becomes insignificant and performance is limited by the overheads of remote communication [5]. As a matter of fact, optimizing for single-node STM performance helps reduce the scalar overheads and does not improve the overall scalability in these systems.

We believe that to effectively support the TM paradigm on large-scale systems (e.g., Cray XT [3], BlueGene/P [2]), the underlying STM runtime must support multiple levels of parallelism and allow transactions to tolerate the long latency of remote data/metadata operations. The first goal can be achieved by supporting a flexible programming model based on a combination of multithreading, SPMD style parallelism, and *Remote Procedure Call* (RPC) invocation. The second goal can be achieved, for example, through a *non-blocking STM abstraction*. Using such an interface, communication latency can be efficiently tolerated by having simultaneous data/metadata requests in-flight across the system and allowing each of these to execute *asynchronously* with respect to the process/thread issuing these requests. Communication overheads will thus be overlapped either with useful computation or other independent communication requests.

The specific contributions of this paper are as follows:

1. We introduce GTM, a high-performance, library-based STM for large-scale systems. GTM exploits asynchronous protocols to tolerate latency and supports a dynamic programming model based on thread-level parallelism, SPMD parallelism, and RPC invocation.
2. We describe a prototype implementation of GTM based on the GASNet [7] communication interface. Our implementation is thread-safe and allows multiple threads of control within a given SPMD process to perform both synchronous and asynchronous transactional operations across the entire system using the GTM interface.
3. We evaluate our prototype for a variety of workloads (Priority Queue, Red-Black Tree, and Bank Transaction). In addition to exhibiting good strong and weak scaling attributes, our experimental results illustrate the need for leveraging asynchronous mechanisms for tolerating latency. We observe best-case speedups as high as $2.2X$.
4. Finally, we show a clear mapping to language-level constructs in new languages that will benefit from the enhanced performance and scalability provided by GTM. In this paper, we focus on Cray's Chapel language and its `atomic` block construct as an illustration of how GTM will be used by future languages.

The rest of the paper is organized as follows. Section 2 provides a brief introduction to the topic. Section 3 uses a simple example to amplify the need for leveraging asynchronous mechanisms in STMs. Section 4 presents the execution model, memory model, and interface design of GTM. This is followed by a description of our prototype implementation in Section 4.4. Section 5 presents the experimental results, followed by a brief conclusion in Section 6.

2. BACKGROUND WORK

This section provides a brief description of the relevant background work on PGAS programming models and Transactional Memory.

2.1 PGAS Models and GASnet

The *Partitioned Global Address Space* (PGAS) model provides the benefits of both shared memory and message-passing style of parallel programming. Various implementations of the PGAS model is already being supported in languages such as UPC, CAF, and Titanium. Also, the HPCS languages have adopted the PGAS model as the underlying memory model to tackle various productivity, portability, and scalability challenges in future large-scale systems.

The PGAS model treats the distributed memory address space as a single globally-addressable memory space. A global memory location is identified using a pair of arguments: *rank* or process identifier that uniquely identifies a specific target process and a virtual memory address within that process's address space. This ordered pair logically represents a global pointer for a given address within the memory partition of the specified process and is valid throughout the system.

The programmer is required to ensure memory consistency when accessing or updating local/remote memory. For example, UPC provides barriers, fences, and `upc_locks` for this purpose. As mentioned earlier, the benefits of using TM over locks has inspired the HPCS languages to provide language support, such as the `atomic` construct, for specifying transactions.[1]

GASNet is a highly-portable low-level communications library providing an scalable infrastructure based on the SPMD model for implementing PGAS languages; it has been used to implement almost all of the above languages.[2] The GASNet programming interface (or API) consists of two parts: the core API and the extended API. The core API is based

[1]The `atomic` construct in X10, unlike Chapel or Fortress, is not optimistic and does not require STM support.

[2]GASNet's SPMD execution model does not enforce the execution model high-level language to also be SPMD. For example, Chapel is implemented using GASNet but provides a *global-view* paradigm.

on the *Active Messages* (AM) [32] paradigm and formulates support for asynchronous RPC operations in terms of request-reply handler mechanisms. The extended API provides the remote data access functionality as one-sided get/put operations.

2.2 Transactional Memory

A lot of attention has been recently focused on leveraging *Transactional Memory* (TM) [20, 24] to simplify parallel programming. TM has been shown to provide performance advantages over locks by eliminating the unnecessary serialization and allowing concurrent access to disjoint data objects. Prior work in TM can be classified into *Software Transactional Memory* (STM) [31, 29, 12, 15], *Hardware Transactional Memory* (HTM) [20, 26], or *Hybrid Transactional Memory* (HyTM) [27, 30] based on the underlying implementation being a software-only solution, a hardware-only solution, or a combination of both software and hardware functionality, respectively.

We are not the first explore the benefits of extending TM across nodes of a distributed memory system. Herlihy et al. [21], Manassiev et al. [25], and Kotselidis et al. [23] proposed STM designs based on a shared memory view of the globally distributed memory. Adopting such a view allows portability of applications developed for cache-coherent, shared-memory architectures. However, the reliance on global software-based cache coherence fundamentally limits the overall scalability of these implementations to tens of nodes.

Recently, Bocchino et al. proposed Cluster-STM [5], the only other STM design (to our knowledge) to target PGAS systems. Cluster-STM provides a strict programming model based on SPMD-style parallelism with purely synchronous STM interface and no support for thread-level parallelism. At startup, Cluster-STM processes dedicate a portion of their heap address space for use as a globally accessible *Transactional Store* (TS). The interface provides transactional semantics only for these TS memory segments of the global address space. This in turn restricts where a compiler/programmer using Cluster-STM procedures can allocate the data accessed from within a transaction. Similar to distributed database designs, Cluster-STM stores all the metadata pertaining to a given transactional memory address on the same SPMD address space as the TS segment to which the memory location belongs. This reduces the number of messages required to perform STM book-keeping operations by combining them with data requests. Finally, Cluster-STM was the first to extend the idea of leveraging RPC mechanisms to exploit locality in distributed STM applications.

In contrast to Cluster-STM, GTM provides better scalability and performance by providing a specific technique – an asynchronous non-blocking protocol – to allow the runtime to hide the high latency of remote communication operations. In addition to this, GTM also provides a more generalized programming model for large-scale systems by combining SPMD, multithreading, and RPC mechanisms. Finally, GTM guarantees transactional semantics for the entire global address space as long as such accesses use the appropriate GTM procedures. At the time of writing this paper, Cluster-STM software was not readily available for a direct performance comparison.

```
        (a) BANK TRANSCTION IN CHAPEL CODE
1 : atomic {
2 :   acc1.balance -= amount;
3 :   acc2.balance += amount;
4 : }

       (b) SYNCHRONOUS BLOCKING ABSTRACTION
5 : TX_BEGIN;   /* Start Transaction */
6 : temp1 = TX_LOAD_B(tgtNode1, &acc1->balance);
7 : temp1 = temp1 - amount;
8 : TX_STORE_B(tgtNode1, &acc1->balance, temp1);
9 : temp2 = TX_LOAD_B(tgtNode2, &acc2->balance);
10: temp2 = temp2 + amount;
11: TX_STORE_B(tgtNode2, &acc2->balance, temp2);
12: TX_COMMIT;   /* Commit Transaction */

    (c) ASYNCHRONOUS NON-BLOCKING ABSTRACTION
13: TX_BEGIN;   /* Start Transaction */
14: TX_LOAD_NB(temp1, tgtNode1, &acc1->balance);
15: TX_LOAD_NB(temp2, tgtNode2, &acc2->balance);
16: TX_WAIT(temp1);
17: temp1 = temp1 - amount;
18: TX_STORE_NB(tgtNode1, &acc1->balance, temp1);
19: TX_WAIT(temp2);
20: temp2 = temp2 + amount;
21: TX_STORE_NB(tgtNode2, &acc2->balance, temp2);
22: /* ...wait for both TX_STORE_NBs to finish */
23: TX_COMMIT;   /* Commit Transaction */
```

Figure 1: Bank Transaction.

Given the lack of hardware support, STM designs, including GTM (Section 4.4), use locks to manage shared STM metadata [12, 15]. On the other hand, few STM designs have used *non-blocking algorithms* [17, 19], enabling concurrently executing transactions to manipulate shared metadata without acquiring locks. These algorithms help provide obstruction-free, lock-free, or wait-free progress guarantees at the cost of increased implementation complexity and lower performance. These algorithms are different from the non-blocking STM abstractions presented in this paper. A comparative study of these approaches is beyond the scope of this paper and will be investigated as part of future work.

3. EXAMPLE: BANK TRANSACTION

To motivate our ideas for non-blocking mechanisms for distributed STM transactions, we use the widely-used, concise example of a bank transfer operation. A simple bank transaction involves the transfer of an `amount` from one bank account to another, say from `acc1` to `acc2`. The top-most part of Figure 1 (lines 1–4) implements this operation using Chapel-like language concepts.[3] Consider the data-structures representing the two bank accounts is allocated on two different nodes, say `tgtNode1` and `tgtNode2` respectively. Figure 1 also illustrates the use of synchronous blocking (lines 5–12) and asynchronous non-blocking (lines 13–23) abstractions for implementing the bank transaction.

The STM procedures `TX_START` (line 5/13) and `TX_COMMIT` (line 12/23) are used to start and commit a transaction respectively. `TX_START` allocates a *read set/write set* and saves the execution state at the point right before the `TX_START`

[3]We are actively collaborating with the Chapel team to implement the `atomic` functionality using GTM transparently.

GTM Procedures and Arguments	Description
gtm_init	Initialize data-structures.
gtm_exit	Clean-up and exit.
TDesc *tx = gtm_tx_create ()	Returns a new transaction descriptor (tx).
gtm_tx_destroy (tx)	Destroys a transaction descriptor identified by tx on srcProc.
gtm_tx_begin (tx)	Begin executing transaction identified by descriptor tx.
gtm_tx_commit (tx)	Attempt to commit transaction identified by tx.
gtm_tx_abort (tx)	Abort transaction identified by tx.
gtm_tx_load (tx, op, tgtNode, destAddr, srcAddr, size)	Transactional load of size bytes starting at srcAddr on tgtNode to address destAddr on calling node.
gtm_tx_store (tx, op, tgtNode, srcAddr, size, destAddr)	Transactional store of size bytes starting at srcAddr on calling node to address destAddr on tgtNode.
gtm_tx_malloc (tx, op, tgtNode, size, addr)	Transactional allocation of size bytes on tgtNode.
gtm_tx_free (tx, op, tgtNode, addr, size)	Transactional free of size bytes starting at addr on workProc.
gtm_fn (op, tgtNode, fnName, inBuf, inSize, outBuf, outSize)	Execute fnName on tgtNode, with input arguments passed via inBuf (size inSize) and results returned via outBuf (size outSize).
gtm_tx_fn (tx, op, tgtNode, fnName, inBuf, inSize, outBuf, outSize)	Similar in functionality to gtm_fn but must always be called inside transaction (identified by tx).
HDesc *op = gtm_op_create()	Returns a new handle (op) to be used for tracking asynchronous operations.
gtm_op_destroy (op)	Destroy handle identified by op.
gtm_op_test (op)	Return execution status of request associated with op.
gtm_tx_wait (tx, op)	Wait for an earlier non-blocking request associated with op to return finish successfully or enclosing abort transaction.
gtm_fn_wait (op)	Wait for an earlier non-blocking gtm_fn request associated with op to finish successfully.

Table 1: List of GTM Procedures

call. The read set and write set keeps track of locations read and written by a transaction respectively. The TX_COMMIT attempts to commit the transaction by ensuring that the transaction does not *conflict* with other concurrently executing transactions and then releases the read/write set. A *conflict* occurs when the read/write set of a transaction overlaps with the write set of another transaction. In case of a conflict, only one transaction can commit, while the rest have to restart their execution beginning at *TX_START*.

The STM procedure TX_LOAD_B (line 6) performs a transactional load operation from the global memory (acc1->balance) on a remote node (tgtNode1) to the transaction's private memory (temp1) *synchronously* with respect to the caller. The procedure TX_STORE_B (line 8) attempts a transactional store operation from the transaction's private memory (temp1) to the global memory (acc1->balance) on a remote node (tgtNode1), again in a synchronous manner. In both these cases, the process/thread executing the transaction blocks for the remote side to satisfy the corresponding data/metadata request and to send an appropriate acknowledgment to the source. Since no remote latency is overlapped, the transaction spends thousands of cycles waiting for these operations to complete.

On the other hand, the procedures TX_LOAD_NB (line 14/15) and TX_STORE_NB (line 18/21) are non-blocking in nature, and transfer data between the global memory space and the temporary private storage *asynchronously* with respect to the caller. This implies, the caller issues remote operations and continues executing the rest of the code instead of waiting for those requests to complete immediately. For example, TX_LOAD_NB to tgtNode1 (line 14) is overlapped with TX_LOAD_NB to tgtNode2 (line 15). The TX_WAIT procedure (line 16/19) ensures the remote request completes before the relevant data is accessed. Thus, asynchronous execution of remote requests helps tolerate long-latency communication events by overlapping them with computation or other remote communication. We intend to leverage this insight to build scalable STM schemes for large-scale systems. We plan to extend these concepts to reduce scalar single-node STM overheads as part of future work.

4. GLOBAL TRANSACTIONAL MEMORY

Global Transactional Memory (GTM) is a high-performance, library-based STM design. We describe the following aspects of GTM design in this section: memory/execution model, interface functionality, and interface specification. Note that GTM is meant as a scalable target for a compiler supporting language-level TM constructs, like Chapel's atomic construct, and not as an end-user programming model.

4.1 Memory and Execution Model

The primary execution model is based on the *Single Process Multiple Data* (SPMD) model of parallelism, where a fixed number of SPMD *nodes* or *processes* are created at startup and no additional process creation/migration allowed later in the execution. Unless explicitly stated, the term "node" refers to a SPMD process and not a hardware node. Further, *source node* identifies the SPMD process that initiated the original request and *target node* identifies the SPMD process on whose context the operation must be executed. The

Call semantics	HDesc (op)	NDesc (tgtNode)
Local Blocking	NULL	Source NDesc
Local Non-Blocking	Valid HDesc	Source NDesc
Remote Blocking	NULL	Remote NDesc
Remote Non-Blocking	Valid HDesc	Remote NDesc

Table 2: GTM Call Semantics as applicable to `gtm_tx_load`, `gtm_tx_store`, `gtm_tx_malloc`, `gtm_tx_free`, `gtm_tx_fn`, and `gtm_fn` procedures. All other GTM procedures are strictly blocking but may implicitly generate remote messages.

request is considered *local* if the source node is the same as the target node, and *remote* otherwise.

The memory model is based on the PGAS view of global memory, where the global address space is partitioned among SPMD processes as described in Section 2. Transactional semantics are guaranteed across this global address space as long as these operations occur inside the dynamic context of a transaction and use the appropriate GTM procedure calls.

Each SPMD process may use multithreading to exploit multicore processors within a hardware node. Our prototype implementation of the GTM library is thread-safe, and allows multiple threads of control within a given SPMD process to invoke GTM procedures across the entire system. In general, this allows for a more dynamic execution model than the traditional SPMD model where all processes are expected to follow similar code execution paths.

4.2 GTM Functionality

The overall functionality of the GTM procedures (Table 1) is listed below:

- Start, commit, and abort transactions. Transactions must be started and committed on the same node, but any number of the following local/remote transactional operations can be performed within the "context" of a transaction.
- Inside a transaction, transfer data between local/remote transactional storage and local private memory with transactional semantics. Further, manage transactional storage dynamically inside a transaction by allocating or deallocating memory locally/remotely with appropriate transactional semantics.
- Invoke user-level procedure calls on a target node using transactional RPC mechanisms. This helps exploit locality by enforcing where a specific function call must execute, and provides an alternative to fetching the required data and executing the same function locally.
- Guarantee completion of asynchronous requests invoked through the non-blocking equivalent of the mechanisms listed above.

4.3 GTM Interface

The three primary descriptor data-structures exposed by GTM procedures in Table 1 are:

```
                (a) SYNCHRONOUS BLOCKING GTM INTERFACE
1 : TDesc *tx = gtm_tx_create();
2 : gtm_tx_begin(tx);  /* Start Transaction */
3 : gtm_tx_load(tx, NULL, tgtNode1, &temp1, &acc1->balance,...);
4 : temp1 = temp1 - amount;
5 : gtm_tx_store(tx, NULL, tgtNode1, &temp1,..., &acc1->balance);
6 : gtm_tx_load(tx, NULL, tgtNode2, &temp2, &acc2->balance,...);
7 : temp2 = temp2 + amount;
8 : gtm_tx_store(tx, NULL, tgtNode2, &temp2,..., &acc2->balance);
9 : gtm_tx_commit(tx);  /* Commit Transaction */
10: gtm_tx_destroy(tx);

            (b) ASYNCHRONOUS NON-BLOCKING GTM INTERFACE
11: TDesc *tx = gtm_tx_create();
12: HDesc *op1 = gtm_op_create();
13: HDesc *op2 = gtm_op_create();
14: gtm_tx_begin(tx);  /* Start Transaction */
15: gtm_tx_load(tx, op1, tgtNode1, &temp1, &acc1->balance,...);
16: gtm_tx_load(tx, op2, tgtNode2, &temp2, &acc2->balance,...);
17: gtm_tx_wait(tx, op1);
18: temp1 = temp1 - amount;
19: gtm_tx_store(tx, op1, tgtNode1, &temp1,..., &acc1->balance);
20: gtm_tx_wait(tx, op2);
21: temp2 = temp2 + amount;
22: gtm_tx_store(tx, op2, tgtNode2, &temp2,..., &acc2->balance);
23: gtm_tx_wait(tx, op1);
24: gtm_tx_wait(tx, op2);
25: gtm_tx_commit(tx);  /* Commit Transaction */
26: gtm_tx_destroy(tx);

          (c) ASYNCHRONOUS NON-BLOCKING GTM RPC INTERFACE
27: TDesc *tx = gtm_tx_create();
28: HDesc *op1 = gtm_op_create();
29: HDesc *op2 = gtm_op_create();
30: inBuf1.acc = acc1; inBuf1.amount = -amount;
31: inBuf2.acc = acc2; inBuf2.amount = amount;
32: gtm_tx_begin(tx);
33: gtm_tx_fn(tx, op1, tgtNode1, transfer_fn, inBuf1,...);
34: gtm_tx_fn(tx, op2, tgtNode2, transfer_fn, inBuf2,...);
35: gtm_tx_wait(tx, op1);
36: gtm_tx_wait(tx, op2);
37: gtm_tx_commit(tx);
38: gtm_tx_destroy(tx);

39: void transfer_fn(account acc, int amount) {
40:   acc.balance += amount;
41:  }
```

Figure 2: Bank Transfer Transaction (Figure 1) using GTM Procedures.

Transaction descriptor (TDesc): for holding all the private metadata (read/write set, execution state for rollback etc.) that defines a transaction.

Handle descriptor (HDesc): for waiting on pending asynchronous operations initiated through the non-blocking interface. A given HDesc can track only one in-flight request at any given time.

Node descriptor (NDesc): for representing the target node on whose context the request must execute.

These descriptor arguments are represented by the literals `tx`, `op`, and `tgtNode` respectively in Table 1. Table 2 summarizes the legal usage combinations and semantics of the HDesc and NDesc descriptor arguments. These descriptors expose the local/remote and blocking/non-blocking call semantics for most GTM procedures. We describe the various GTM procedures in greater detail below. Figure 2 illustrates the use of GTM procedures to implement the pseudo-code in Figure 1. The use of GTM RPC procedures is also shown.

Note that the number of remote messages inside a transaction is reduced from four to two in the RPC version. The arrows indicate the separation between the issue and completion of the request.

4.3.1 Setup

The `gtm_init` and `gtm_exit` calls provide the initialization and clean-up functionality. Each SPMD processes must call `gtm_init` before all other GTM calls and `gtm_exit` after all other GTM calls.

4.3.2 Transaction Management

The procedures `gtm_tx_create` and `gtm_tx_destroy` create and destroy a transaction descriptor respectively. The procedures `gtm_tx_begin`, `gtm_tx_commit`, and `gtm_tx_abort` start, commit, and abort transactions on the specified TDesc argument (`tx`). All these procedures are strictly blocking and local to the node on which they are initiated. However, the `gtm_tx_commit` and `gtm_tx_abort` calls may implicitly initiate remote communication based on whether the transaction performed remote operations or not. The `gtm_tx_abort` additionally ensures all asynchronous requests previously issued inside the transaction complete before the transaction is retired (see Section 4.3.5).

Nested transactions created using dynamic pairs of `gtm_tx_begin` and `gtm_tx_commit` calls must use the outermost transaction's TDesc descriptor. Nested transactions are handled by subsuming all inner transactions into the context of the outer-most transaction, namely *flat nesting*. In other words, an abort at any nested level will abort the entire transaction. A commit at any nested level will be made globally visible only when the outer-most transaction commits successfully.

4.3.3 Transactional Data Management

GTM supports data transfer operations from/to transfer data to/from local/remote transactional storage from/to local private memory with appropriate semantics. The `gtm_tx_load` call copies `size` bytes from the source address starting at `srcAddr` on `tgtNode` to the destination address starting at `destAddr` within the transaction's local private address space. The `gtm_tx_store` call copies `size` bytes from the source address starting at `srcAddr` on the transaction's private address space to the destination address starting *destAddr* on `tgtNode`'s memory partition.

GTM also supports dynamic management of the global heap storage using the `gtm_tx_malloc` and `gtm_tx_free`. Memory allocated using `gtm_tx_malloc` is automatically freed on an abort and memory freed using `gtm_tx_free` is made visible globally only on a successful commit. In general, these calls are valid only inside the scope of a transaction (i.e. between a dynamic pair of `gtm_tx_begin` and `gtm_tx_commit`). A transaction can commit only if all its data movement and memory allocation procedures execute without conflicting with other concurrently executing transactions. The non-blocking equivalent of these procedures must be followed by a `gtm_tx_wait` with the corresponding HDesc descriptor (Section 4.3.5).

4.3.4 Transactional Procedure Call Invocation

The procedures `gtm_tx_fn` and `gtm_fn` are used to invoke any user-level routines. The invoked routines themselves may start their own transactions or in turn use the `gtm_fn` call to invoke other user-level routines or a combination of both. Table 2 lists the call semantics for these procedures.

All `gtm_tx_fn` procedures must be called only inside a transaction and like nested transactions use the TDesc of the outer-most transaction. A transaction can commit only if all its `gtm_tx_fn` procedures execute without conflicting with other concurrently executing transactions. The non-blocking equivalent of the `gtm_tx_fn` procedure must be followed by a `gtm_tx_wait` with the corresponding TDesc and HDesc descriptor (Section 4.3.5).

The `gtm_fn` procedure helps invoke user-level routines that start and commit transactions on the `tgtNode` node without maintaining any direct dependence with other transactional/non-transactional operations issued by the same process/thread. It is intended to be used from the non-transactional portions of the code. However, if called inside the context of another transaction, either directly or indirectly from a routine invoked via the `gtm_tx_fn` procedure, it executes independently and is not affected by the eventual fate of the transaction. The non-blocking equivalent of the `gtm_fn` procedure must be followed by a `gtm_fn_wait` with the corresponding HDesc handle descriptors.

4.3.5 Handle Management

The procedures `gtm_op_create` and `gtm_op_destroy` create and destroy HDesc descriptor handles used for tracking asynchronous requests. During its lifetime, a given HDesc can be reused for tracking different asynchronous requests initiated through the non-blocking equivalent of the GTM procedures for data transfer, memory allocation, and function call invocation. However, at any given instant each HDesc can track only one asynchronous request in-flight and by design each asynchronous request has a single HDesc associated with it when it is invoked.

The `gtm_op_test` procedure queries the execution status of the asynchronous request associated with the `op` HDesc handle. The call returns immediately with the one of the following: the request executed without any conflicts, the request conflicted with another transaction, or the request hasn't completed. This status information can be used to decide if the transaction must abort, attempt to commit, or keep executing the rest of the transaction.

The `gtm_tx_wait` procedure is always used inside a transaction to block for the asynchronous request associated with `op` to complete. If the request executed without conflicts, then `gtm_tx_wait` frees the HDesc (`op`) for reuse in future non-blocking GTM procedures and makes the data payload (e.g. at `destAddr` in the case of `gtm_tx_load`) available for usage. If the request failed due to conflicts, the `gtm_tx_wait` procedure implicitly aborts the enclosing transaction. The `gtm_tx_wait` also ensures that all other asynchronous requests issued inside the transaction complete before the transaction is aborted and retried.

The `gtm_fn_wait` procedure is used for waiting on non-blocking equivalent of the `gtm_fn` procedure and must al-

ways be called outside a transaction. If the invoked routine (`fnName`) started its own transactions and committed without conflicting with any other transactions, then `gtm_fn_wait` frees the HDesc `op` for reuse in future non-blocking GTM procedures and makes the data payload (i.e. `outBuf`) available for consumption.

4.4 GTM Implementation

This section describes the prototype implementation of GTM based on the GASNet communication interface [7]. The implementation is thread-safe and allows multiple threads of control within a given SPMD process to perform transactional operations across the entire system using GTM procedures. Furthermore, the prototype has been implemented to coexist with GASNet on the same software stack layer, allowing current PGAS languages already using GASNet to seamlessly use GTM for implementing language-level TM constructs. The prototype does not allow `gtm_tx_fn` and `gtm_fn` procedures to invoke routines that initiate additional remote communication. This is not a limitation for the workloads presented in this paper. A number of implementation strategies are currently being evaluated for this purpose.

The GTM prototype extends a number of existing STM algorithms to guarantee the necessary transactional semantics across the entire system. The baseline algorithmic design parameters can be broadly described as: *word-based* [15, 5], *weak isolation* [12, 5], *read versioning* [29, 15], *deferred update* [19, 17], and *eager acquire* [14, 18]. Due to space reasons, the implementation of various GTM procedures based on these design choices and correctness arguments on why asynchronous operations guarantee the right semantics is not presented here.

GTM maintains a variety of metadata information for guaranteeing the necessary transactional semantics, including but not limited to: *globally shared metadata* and *private per-transaction metadata*.

Each node maintains a globally-accessible hash-table (2^{20} entries per node) for supporting transactional operations initiated by both the local node and any remote node in the system, and is similar to *ownership records* (ORecs) used in most word-based STM designs. Hash-table entries store and manage metadata for some unique set of memory locations in the node's address space (stack, heap, and global section). The least significant bit (LSB) of each entry is set if any of the memory addresses that maps onto it are currently being written. Otherwise, the entry maintains a $63 - bit\ version\ number$ for the memory addresses it is responsible for. Transactional load/store/abort/commit operations modify these entries appropriately. The actual size of this data-structure affects runtime performance only if the rate of conflicts between concurrently executing transactions in a given node is very high and does not limit the overall scalability of the implementation.

Each node also maintains a dynamic pool of TDesc descriptors (Section 4.3). TDesc descriptors are used in GTM procedures to identify a specific transaction and store metadata private to that transaction, including but not limited to: transactionÕs status, nesting level, read/write sets, and information on completed/pending operations initiated as part of the transaction. The read set/write set are implemented as two separate list-based data-structures, and keep track of the list of locations read/written by the transaction along with a copy of the global metadata entry for each of those locations. The list of pending/completed operations help track operations initiated across the entire system and are used to perform commit/abort sequences on those target nodes as required.

5. RESULTS

This section presents experimental results based on three workloads: Red-Black Tree, Priority Queue, and Bank Transaction. These workloads are based on benchmarks used in prior STM work targeting multicore/SMP systems [12, 15]. Good STM benchmarks targeting large-scale systems are required for robust performance evaluation. However, such development efforts will not be undertaken until high-performance STM designs are readily available in the first place. This chicken-and-egg problem was also faced by early TM work targeting multicore/SMP systems [24]. For example, lock-based applications were used until recent benchmarking efforts were undertaken to fill the gap [8]. The results presented in this section attempts to demonstrate the overall scalability of GTM. We are currently developing benchmarks that exploit distributed STM designs and plan to use GTM as an implementation platform against which future comparisons can be made.

All experiments were performed on the Jaguar Cray XT4 system at the Oak Ridge National Laboratory. Each node consists of a 2.1 GHz Quad-core AMD Opteron processor with $8GB$ of memory and a Cray Seastar NIC. Nodes are interconnected in a 3D torus topology, providing very low latency messaging between nodes. All workloads were compiled using PGI-7.2.4 C language compiler (O3 optimization) and GASNet Portals Conduit (version 1.12.0). Each Jaguar XT4 quad-core node is assigned a single SPMD process and each SPMD process is single-threaded. The problem size (S, M, L) identifies the total successful transaction commits across the entire system. The problem is equally distributed amongst all the nodes and each node commits *(problem size / node count)* transactions. As we increase the node count for a given problem size, each node commits fewer transactions keeping the sum total constant. The random number generator follows uniform distribution and is controlled to provide comparable results across two different runs of a given data point.

For each workload, we present two different sets of graphs. In the *Execution Time* graphs, the *X-Axis* represents the node count and the *Y-Axis* represents the total execution time for each problem size (S, L, and M). The use of blocking versus non-blocking GTM procedures is represented by *GTM_B* and *GTM_NB*, respectively. On a single node, GASNet does not differentiate asynchronous requests from synchronous requests resulting in similar execution times for *GTM_B* and *GTM_NB*. In the *Speedup* graphs, the *X-Axis* represents the node count and *Y-Axis* represents the speedup (larger the better) and is calculated by the ratio of execution time for the *GTM_B* case over *GTM_NB* case for a given problem size and node count. For example, Speedup-S = <*GTM_B,S*> / <*GTM_NB,S*>.

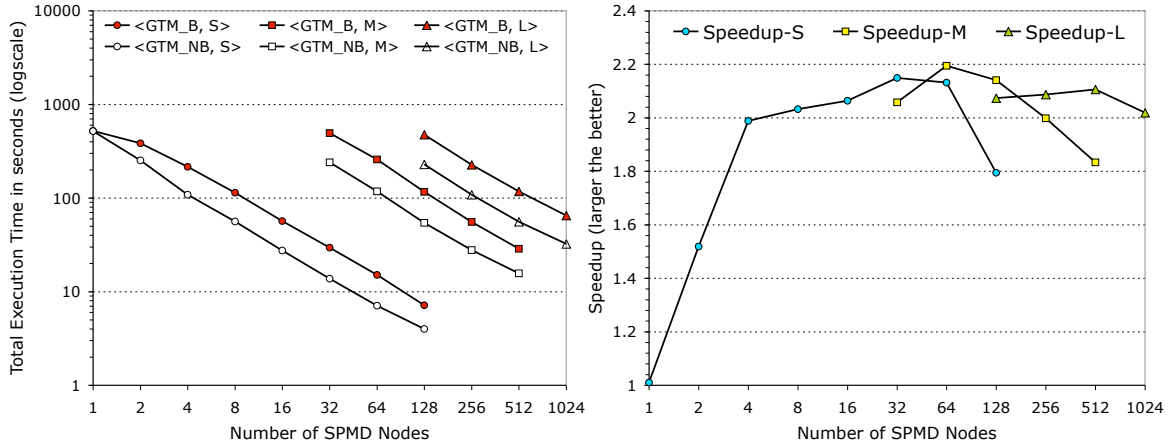

Figure 3: Red-Black Tree Results: Execution time and Speedup results for problem sizes *S, M, and L*.

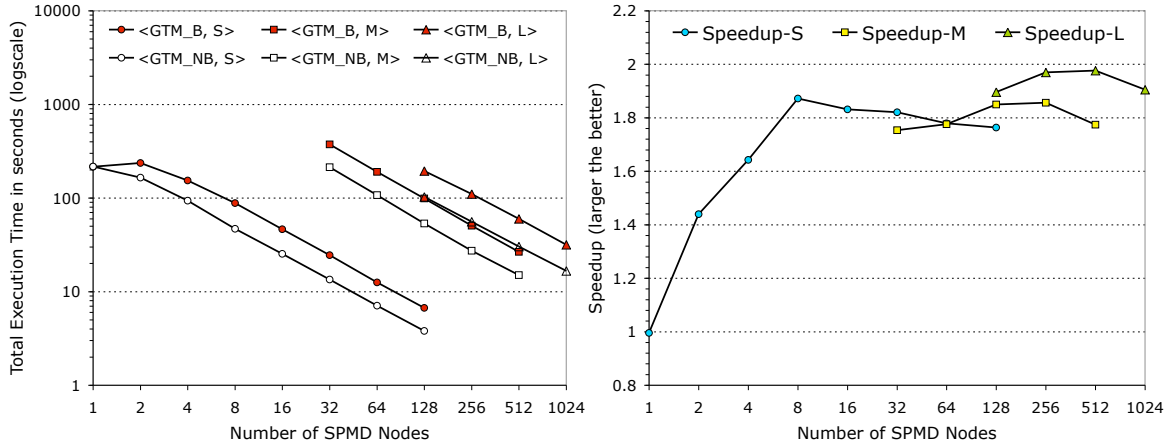

Figure 4: Priority Queue Results: Execution time and Speedup results for problem sizes *S, M, and L*.

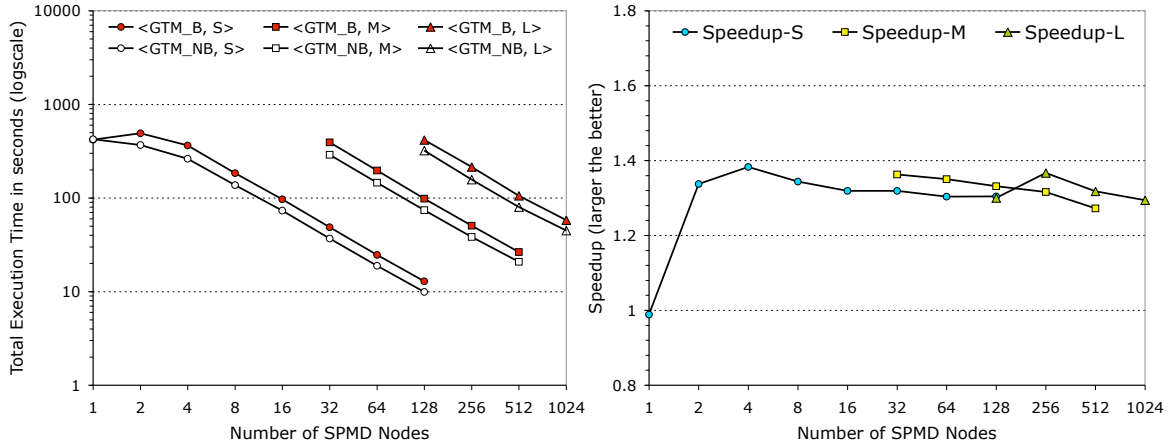

Figure 5: Bank Transfer Transaction: Execution time and Speedup results for problem sizes *S, M, and L*.

5.1 Red-Black Tree

This workload operates on a distributed RB-tree, based on STAMP's version of the TL2 RB-tree code [8]. Each process randomly picks a target node and uses the `gtm_fn` procedure to invoke RB-Tree routines (`insert()`, `delete()`, or `search()`) on the chosen target. These routines start their own transactions and atomically access the local RB-Tree on the target node using local/blocking transactional loads/stores. Each `gtm_fn` results in the successful execution of a single transaction. Transactions are short running with small read/write sets. Here, S is 2^{24}, M is 2^{28}, and L is 2^{30} committed transactions respectively.

Figure 3 presents the execution time and speedup results for the Red-Black Tree workload. *GTM_B* and *GTM_NB* represent the blocking and non-blocking use of the `gtm_fn` procedure. The *GTM_NB* case allocates an array of 32 HDesc descriptors at startup, allowing at most 32 simultaneous in-flight asynchronous `gtm_fn` requests for each SPMD process.

A number of general observations can be made on the scalability of the Red-Black Tree kernel based on the execution time results. First, the execution time almost halves as the node count is doubled, implying good strong-scaling behavior. For example, the total execution time for <*GTM_NB,M*> on 256 nodes (27.73 seconds) is almost half as of 128 nodes (54.33 seconds). Second, the total execution time remains almost the same if the node count and problems sizes are multiplied by the same factor, implying good weak-scaling behavior. For example, the execution time for <*GTM_B,S*> on 32 nodes (29.53 seconds) is close to <*GTM_B,M*> on 512 nodes (28.71 seconds), given problem sizes S and M vary by a factor of 16. It should be noted that the strong and weak scaling trends do not hold perfectly across all the problem and system sizes tested.

Figure 3 also illustrates the real benefits of latency toleration techniques in STMs for distributed memory systems. These techniques provide additional speedups of $2X$ or more for problem size L (*Speedup-L* curve), speedups of $1.83X$ or more for problem size M (*Speedup-M* curve), and finally speedups greater than $1.79X$ for problem size S (*Speedup-S* curve beyond four nodes).

5.2 Priority Queue

This workload operates on a distributed priority queue implemented using a sorted linked-list. The test harness is similar to the Red-Black Tree workload. Each process randomly picks a target node and uses the `gtm_fn` procedure to invoke priority queue routines (`insert()`, `delete()`, or `search()`) on the chosen target. These routines start their own transactions and atomically access the local priority queue on the target node using local/blocking transactional loads/stores. Thus, each `gtm_fn` results in the successful execution of a single transaction. Transactions are long running with large read/write sets. Here, S is 2^{24}, M is 2^{28}, and L is 2^{29} committed transactions respectively. Figure 4 presents the execution time and speedup results for the Priority Queue workload. Here, *GTM_B* and *GTM_NB* represent the blocking and non-blocking use of the `gtm_fn` procedure.

A number of general observations can be made on the scalability of the Priority Queue workload based on the execution time results in Figure 4. First, good strong-scaling and weak-scaling attributes can be observed for most problem sizes and node counts. This implies GTM scales well on large node counts. Second, the execution time for <*GTM_B,S*> increases going from one to two nodes and is a result of over-the-network accesses that begins to occur at two nodes. Unlike in <*GTM_NB,S*>, the overhead of remote communication is fully exposed due to the lack of latency tolerance techniques. But as we increase the node count, the work per node decreases, resulting in better performance compared to the one/two node case. Finally, <*GTM_B,M*> and <*GTM_NB,L*> have similar execution times (on 128 nodes it is $98.8s$ and $102.4s$ respectively). This implies leveraging asynchronous protocols provide sufficient speedup to run larger problems on smaller systems if the problem fits in the memory of fewer nodes.

Figure 4 demonstrates the significant improvements in performance that can be achieved by issuing transactional operations asynchronously. These techniques provide additional speedups of $1.89X$ or more for problem size L (*Speedup-L* curve), speedups of $1.75X$ or more for problem size M (*Speedup-M* curve), and finally speedups greater than $1.76X$ for problem size S (*Speedup-S* curve beyond four nodes).

5.3 Bank Transaction

This workload implements the bank transfer example discussed in Section 3. Each process starts a transaction, randomly picks two target nodes (one of which may be local), and uses `gtm_tx_fn` to invoke the `transfer_fn()` routine (Figure 2) on the chosen target nodes. Unlike the earlier workloads, there can be at most two outstanding operations inside a transaction and one transaction per process. In effect, these transactions are short running with small read/write sets. Here, S is 2^{24}, M is 2^{28}, and L is 2^{29} committed transactions respectively.

Figure 5 presents the execution time and speedup results for the bank transaction workload. Again good strong-scaling and weak-scaling attributes can be observed from the execution time results. The speedup results exhibit completely different trends compared the other workloads, with speedups ranging between $1.27X$ and $1.38X$. Here, a process executes only one transaction at a time with at most two outstanding operations inside a transaction.

5.4 Additional Discussion

Due to space reasons, this section summarizes few additional experimental results.

Serialization Issues: Changing the number of HDesc descriptors beyond 32 for the RB-Tree and Priority Queue had no significant effect on the total execution time or speedup, and is the result of serialization effects that occur within the GASNet layer. For example, one source of serialization happens at the source node where GASNet allocates a finite pool of communication buffers limiting the number of simultaneous in-flight requests.

Multithreading: Leveraging thread-level parallelism helps in better utilization of multicore processors by allowing multiple threads of control within a SPMD process to execute multiple transactions concurrently. On smaller node counts,

having two threads per SPMD process resulted in better execution times than the single thread case (presented above). For larger node counts, the overhead of communication and the serialization effects described earlier limited the benefits of an additional computation thread.

Transaction Aborts: On an average, 99% of the transactions committed successfully without aborting even a single time. This high rate of success is due to both workload characteristics and the serialization of AM handlers in GASNet.

6. CONCLUSIONS AND FUTURE WORK

This paper presented GTM, a high-performance, library-based STM design for large-scale systems. GTM supports a dynamic programming model based on thread-level parallelism, SPMD parallelism, and RPC mechanisms. In addition to this, GTM leverages asynchronous protocols to tolerate remote communication latency. This paper also presented various details regarding semantics, interface design, prototype implementation, and performance evaluation.

We are actively collaborating with the Chapel language group in using GTM as the compiler target for transparently implementing the `atomic` construct. We are also developing large-scale STM benchmarks (e.g. unstructured adaptive mesh, boolean satisfiability, and graph-based problems) in Chapel to help evaluate various semantic and implementation design choices. This initial work leads us to believe GTM is capable of providing both a scalable and generalized solution for satisfying the needs of the Chapel language.

7. REFERENCES

[1] A.-R. Adl-Tabatabai, et al. Compiler and Runtime Support for efficient Software Transactional Memory. In *PLDI '06*, New York, 2006. ACM Press.

[2] S. Alam, et al. Early evaluation of IBM Bluegene/P. In *SC08*, Austin, 2008. ACM/IEEE.

[3] S. R. Alam, et al. An Evaluation of the Oak Ridge National Laboratory Cray XT3. *International Journal of High Performance Computing Applications*, 2008.

[4] E. Allan, et al. The Fortress Language Specification. Technical Report Version 0.618, Sun Microsystems, Apr 2005.

[5] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software Transactional Memory for Large Scale Clusters. In *PPoPP '08*, ACM Press.

[6] C. Kotselidis, et al. Distm: A software transactional memory framework for clusters. In *ICPP '08*. IEEE Computer Society Press.

[7] D. Bonachea. Gasnet Specification, v1.1. Technical report, Berkeley, CA, USA, 2002.

[8] C. Cao Minh, et al. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *ISCA' 07*, ACM Press.

[9] C. Cascaval, et al. Software Transactional Memory: Why is it only a research toy ? *Communications to the ACM*, 51(11):40–46, 2008.

[10] P. Charles, et al. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *OOPSLA '05*, ACM Press.

[11] Cray. Chapel: The Cascade High-Productivity Language. *HIPS*, 00:52–60, 2004.

[12] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *DISC 2006*, 2006.

[13] J. Dongarra. DARPA's HPCS program: History, Models, Tools, Languages. In *Advances in Computers*, Volume 72. Elsevier, 2008.

[14] R. Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[15] P. Felber, C. Fetzer, and T. Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory In *PPoPP '08*, ACM Press.

[16] T. E. Ghazawi, et al. *UPC: Distributed Shared Memory Programming*. Wiley, 2005.

[17] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03*, ACM Press.

[18] T. Harris, et al. Optimizing memory transactions. *SIGPLAN Not.*, 41(6):14–25, 2006.

[19] M. Herlihy, et al. Software transactional memory for dynamic-sized data structures. In *Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2003.

[20] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA'93*, ACM Press.

[21] M. Herlihy and Y. Sun. Distributed Transactional Memory for Metric-Space Networks. In, Volume 3724 of *Lecture Notes in Computer Science*, pages 324–338. Springer, 2005.

[22] S. Kang and D. A. Bader. An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. In *PPoPP '09*, ACM Press.

[23] C. Kotselidis, et al. Distm: A software transactional memory framework for clusters. In *ICPP '08*, IEEE Computer Society Press.

[24] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan Claypool, 2006.

[25] K. Manassiev, M. Mihailescu, and C. Amza. Exploiting Distributed Version Concurrency in a Transactional Memory Cluster. In *PPoPP '06*, ACM Press.

[26] A. McDonald, et al. Characterization of TCC on chip-multiprocessors. In *PACT '05*, IEEE Computer Society.

[27] M. Moir. Hybrid Transactional Memory, Jul 2005. Unpublished manuscript.

[28] R. W. Numrich and J. Reid. Co-Array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[29] B. Saha, et al. Mcrt-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. In *PPoPP '06*, ACM Press.

[30] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for Software Transactional Memory. In *MICRO '06*, IEEE Computer Society.

[31] N. Shavit and D. Touitou. Software Transactional Memory. In *PODC'95*, Aug 1995.

[32] T. von Eicken, et al. Active messages: A Mechanism for Integrated Communication and Computation. In *ISCA'92*, ACM Press.

[33] K. Yelick, et al. Titanium: A High-Performance Java Dialect. *Concurrency: Practice and Experience*, 1998.