Evaluation of Blue Gene/Q Hardware Support for Transactional Memories

Amy Wang
IBM Toronto Software Lab.
Markham, ON, Canada
aktwang@ca.ibm.com

Matthew Gaudet
Dep. of Computing Science
University of Alberta
Edmonton, AB, Canada
mgaudet@ualberta.ca

Peng Wu
IBM Research
Yorktown, NY, USA
pengwu@@us.ibm.com

José Nelson Amaral
Dep. of Computing Science
University of Alberta
Edmonton, AB, Canada
jamaral@ualberta.ca

Martin Ohmacht
IBM Research
Yorktown, NY, USA
mohmacht@us.ibm.com

Christopher Barton
IBM Toronto Software Lab.
Markham, ON, Canada
kbarton@ca.ibm.com

Raul Silvera
IBM Toronto Software Lab.
Markham, ON, Canada
rauls@ca.ibm.com

Maged Michael
IBM Research
Yorktown, NY, USA
magedm@us.ibm.com

ABSTRACT

This paper describes an end-to-end system implementation of the transactional memory (TM) programming model on top of the hardware transactional memory (HTM) of the Blue Gene/Q (BG/Q) machine. The TM programming model supports most C/C++ programming constructs on top of a best-effort HTM with the help of a complete software stack including the compiler, the kernel, and the TM runtime.

An extensive evaluation of the STAMP benchmarks on BG/Q is the first of its kind in understanding characteristics of running coarse-grained TM workloads on HTMs. The study reveals several interesting insights on the overhead and the scalability of BG/Q HTM with respect to sequential execution, coarse-grain locking, and software TM.

1. DELIVERING THE PROMISED TRANSACTIONAL MEMORY SIMPLICITY

Transactional memory (TM) was proposed more than twenty years ago as a hardware mechanism to enable atomic operations on an arbitrary set of memory locations [13, 20]. The target applications for TM are those with concurrent computations where it is not possible to determine, until runtime, which specific computations will result into conflicting accesses to memory.

Given the high cost of implementing TM in hardware, the research community developed several implementations of software transactional memory (STM) [7, 9, 18, 19] and conducted simulation-based studies of hardware transactional memory (HTM) [1, 2, 17]. An early implementation of HTM was never distributed commercially [6]. For the HTM by Azul, there is little public disclosure on the implementation and no performance study of the TM support [5]. The specification of a hardware extension for TM in the AMD64 architecture has yet to be released in hardware [4]. It is only recently that IBM [12] and Intel [14] disclosed that they are releasing implementations of HTM.

This paper makes three important contributions. First, it provides a detailed description of the BG/Q HTM implementation and quantifies its major sources of overheads. One of the main pain points of STMs is the high overhead incurred to start and commit a transaction and to instrument and monitor memory references inside a transaction [3]. While it is widely expected that such overheads be significantly reduced in an HTM, one of the surprising findings of this performance study is that, the BG/Q HTM overhead, while much smaller than that of STM's, is still non-trivial. The causes of HTM overheads are also very different from those of STM's. For instance, BG/Q TM maintains speculative states in the L2 cache. This allows for transactions with a large memory footprint, the price to pay, however, is the overhead, where the L1 cache is either bypassed during a transaction or flushed upon entry to a transaction. The loss of cache locality is the dominant cause of BG/Q TM overhead.

Second, the paper conducts a thorough evaluation of the STAMP benchmark suite [16] running on BG/Q TM and in comparison with sequential execution, OpenMP critical, and TinySTM [10]. By comparing the performance of alternative concurrency implementations of the same benchmark with respect to sequential execution baseline, we try to answer the question of how effective BG/Q TM is to improve performance. The performance study leads us to divide typical concurrent applications into three categories. There are applications that are suitable for BG/Q-style HTM, where significant performance improvements can be achieved with very low programming effort. Such appli-

PACT '12 Minneapolis, Minnesota USA

cations usually use medium to large transactions that fit in the capacity of BG/Q HTM and exhibit low abort ratios. On the other hand, applications that scale well with conventional locking should use neither STM nor BG/Q TM as both incur larger single-thread overhead. Furthermore, applications using very small transactions but with abundant optimistic concurrency may be better suited for STM because the single-thread overhead of an STM system may be compensated by the additional scalability it enables.

Third, we describe how the best-effort HTM support in BG/Q can be complemented with a software stack that includes the kernel, the compiler, and the runtime system to deliver the simplicity of a TM programming model. The HTM support in BG/Q is best effort because some program-level transactions may never succeed in executing on BG/Q HTM due to the bounded capacity for transactional reads and writes. In terms of programmability, HTM is a clear win over STM. The latter often requires annotations and instrumentation of all codes that can potentially be invoked in a transactional execution. In terms of performance, there is also a noticeable programming difference between codes manually instrumented for STM and those using BG/Q TM. For example, the STM version of the STAMP benchmark is heavily privatized to minimize instrumented memory accesses, whereas the BG/Q TM version of the codes requires only block annotation of transactional codes. As a result, on the two benchmarks that trigger capacity overflow in a single-thread BG/Q TM execution, the STM version performs well and with extremely low overhead because the amount of instrumented states are dramatically reduced by manual instrumentation.

The rest of the paper is organized as follows. Section 2 describes the hardware support for transactional memory in BG/Q. Section 3 describes the TM programming model. And Section 4 describes the extension to the XL compiler, the runtime system, and the kernel to support the TM programming model. The performance study using the STAMP benchmark suite is presented in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2. HARDWARE TRANSACTIONAL MEMORY IMPLEMENTATION IN BG/Q

In BG/Q, each compute chip has 16 A2 processor cores, each core can run four hardware Simultaneous Multi-Threaded (SMT) threads. A core has a dedicated 16K-byte level-1 (L1) cache that is 8-way set-associative with a cache line size of 64-byte and a 2K-byte prefetching buffer. All 16 cores share a 32M-byte level-2 (L2) cache with a cache line size of 128-byte.

As one of the first commercial implementations of HTM, BG/Q provides the following hardware mechanisms to support transactional execution:

- **Buffering of speculative states.** Stores made during a transactional execution constitute *speculative states*. In BG/Q, speculative states are buffered in the L2 cache and are only made visible (atomically) to other threads after a transaction commits.

- **Conflict detection.** Under the transactional execution mode, the hardware detects read-write, write-read, or write-write conflicts among concurrent transactions or when a transactional access is followed by a non-transactional write to the same address. When a conflict is detected, the hardware sends interrupts to transactional threads involved in the conflict. A special conflict register is flagged to record various hardware events that cause a transaction to fail.

2.1 Hardware Support for Transactional Execution in L2

BG/Q's hardware support for transactional execution is implemented primarily in the L2 cache, which serves as the point of coherence. The L2 cache is divided into 16 slices, where each slice is 16-way set-associative. To buffer speculative states, the *multi-versioned* L2 cache can store multiple versions of the same physical memory line, each version occupies a different L2 way. BG/Q uses a pre-existing core design and therefore there is no hardware modification to support transactional execution in L1.

Upon a transactional write, the L2 allocates a new way in the corresponding set for the write. A value stored by a transactional write is private to the thread. It is made visible to other threads when a transaction commits and is discarded upon a transaction abort.

In addition, the L2 directory records, for each access, whether it is read or written, and whether it is speculative. For speculative accesses, it also tracks which thread has read or written the line by recording the speculation ID used by the thread to activate speculation. This tracking provides the basic bookkeeping to detect conflicts among transactions and between transactional and non-transactional memory operations.

The hardware uses unique speculative IDs to associate a memory operation with a transaction. BG/Q provides 128 speculative IDs for transactions. If a program runs out of IDs then the start of a new transaction blocks until an ID becomes available. The L2 examines, at predetermined intervals, lines whose speculation ID has either been invalidated or committed. When all the lines associated with an invalidated ID are marked as invalid, or when lines associated with a committed ID are merged with the non-speculative state, the speculation ID is reclaimed and made available again. This process is called *ID scrubbing*. The interval between two starts of the scrubbing process is the *scrubbing interval*. The default scrubbing interval is 132 cycles but it can be altered by the runtime via a system call. Setting this interval too high may lead to the blocking of new transactions. Setting it too low may cause the scrubbing activity to interfere with the normal operation of the L2 cache.

The buffering of speculative states in the L2 requires cooperation from components of the memory subsystem that are closer to the pipeline than the L2, namely, the L1 cache. BG/Q supports two transactional execution modes for proper interaction between the L1, the L1 prefetcher (L1P) and the L2, each with a different performance consideration. From herein L1 refers to both L1 and L1P unless otherwise stated. The main difference between the two modes is in how the L1 cache keeps a speculative thread's writes invisible to the other three SMT threads sharing the same L1.

- **Short-running mode (via L1-bypass).** In this mode, when a TM thread stores a speculative value, the core evicts the line from the L1. Subsequent loads from the same thread have to retrieve the value from that point on from L2. If the L2 stores multiple values for the same address, the L2 returns the thread-specific

data along with a flag that instructs the core to place the data into the register of the requesting thread, but to not store the line in the L1 cache. In addition, for any transactional load served from the L1, the L2 is notified of the load via an L1 notification. The notification from L1 to L2 goes out through the store queue.

- **Long-running mode (via TLB aliasing).** In this mode, speculative states can be kept in the L1 cache. The L1 cache can store up to 5 versions, 4 transactional ones for each of the 4 SMT threads, and one non-transactional. To achieve this, the software creates an illusion of versioned address space via Translation Lookaside Buffer (TLB) aliasing. For each memory reference by a speculative thread some bits of the physical address in the TLB are used to create an aliased physical address by the Memory Management Unit (MMU). Therefore, the same virtual address may be translated to 4 different physical addresses for each of the 4 SMT TM threads at the L1 level. However, as the load or store exits the core, the bits in the physical address that are used to create the alias illusion are masked out because the L2 maintains the multiversion through the bookkeeping of speculation ID. In this mode the L1 cache is invalidated upon entering the transaction because there is no L1 notification to the L2 on an L1 hit. The invalidation of the L1 cache makes all first TM accesses to a memory location visible to the L2 as an L1 load miss.

These two modes are designed to exploit different locality patterns. By default, an application runs under the long running mode. One can specify an environment variable to enable the short-running mode before starting an application. The main drawback of the short-running mode is that it nullifies the benefit of the L1 cache for read-after-write access patterns within a transaction. Thus it is best suited for short-running transactions with few memory accesses. The long-running mode, on the other hand, preserves temporal and spatial locality within a transaction, but, by invalidating L1 at the start of a transaction, prevents reuse between code that run before entering the transaction and code that run within the transaction or after the transaction ends. Thus, this mode is best suited for long-running transactions with plenty of intra-transaction locality.

2.2 Causes of Transactional Execution Failures

BG/Q hardware supports bounded, best-effort, transactional execution. While the entire instruction set is allowed in transactional execution, a transaction may fail in the following scenarios:

- **Transactional conflicts** are detected by the hardware at the L2 cache level as described earlier. In the short-running mode conflicts are detected at a granularity of 8 bytes if no more than two threads access the same cache line, or 64 bytes otherwise. In the long-running mode the granularity is 64 bytes and can degrade depending on the amount of prefetching done by a speculative thread.

- **Capacity overflow** cause a transaction to fail when the L2 cache cannot allocate a new way for a speculative store. By default, the L2 guarantees 10 ways to be used for speculative storage without an eviction. Therefore, up to 20M-bytes (32M*10/16) of speculative state can be stored in the L2. A set may contain more than 10 speculative ways if the speculative ways have not been evicted by the Least Recently Used (LRU) replacement policy. In practice, however, capacity failures may occur at a much smaller speculative state footprint by exhausting the way limitation within a set.

- **Jail mode violation (JMV)** occur when the transaction performs *irrevocable actions*, that is, operations whose side-effects cannot be reversed, such as writes to device I/O address space. Irrevocable actions are detected by the kernel under a special mode called the *jail mode*, which will send a JMV interrupt to the owner thread of the event.

```
#pragma tm_atomic
{
    a[b[i]] += c[i];
}
```

Figure 1: Transaction in `kmeans` expressed in BG/Q TM annotation.

3. TRANSACTIONAL MEMORY (TM) PROGRAMMING MODEL

BG/Q provides a simple TM programming model based on an abstraction called *transactions*. Transactions are single-entry/single-exit code blocks denoted by `#pragma tm_atomic` annotations. Any computation is allowed in a transaction. The only constraint is that the boundary of a transaction must be statically determinable in order for the compiler to insert proper code to end a transaction. As a result, control-flow constructs that may exit a transactional block may result in a compile- or run-time error. Similarly, exceptions thrown by a transaction are unsupported.

The semantics of a transaction is similar to that of a critical section or that of relaxed transactions as defined in [11]. In a multi-threaded execution, transactions appear to execute sequentially in some total order with respect to other transactions. Specifically, operations inside a transaction appear not to interleave with any operation from other (concurrent) transactions. The specification of transactional code region is orthogonal to the threading model such as the use of OpenMP or Pthreads.

Two transactions are nested if one transaction is entirely inside the other transaction. Nested transactions in BG/Q are *flattened*: the entire nest commits at the end of the outermost level of nesting. A failed nested transaction rolls back to the beginning of the outermost nesting level. The nesting support is implemented purely in software, specifically in the TM runtime.

Figure 1 shows a transaction code expressed by BG/Q TM annotations. This simple programming interface may enclose complex and large transaction regions.

4. SOFTWARE SUPPORT FOR TM PROGRAMMING MODEL

While the BG/Q HTM is bounded and can fail a transactional execution in various ways, a transaction, as defined by

the programming model, can be arbitrarily large and is guaranteed to eventually succeed. The TM software stack, developed to bridge the gap between the TM programming model and the TM hardware implementation, includes a transactional memory run-time system and extensions to the kernel and the compiler.

The main component of the TM software stack is the TM runtime. The description of the state transition flow managed by the TM runtime to execute a transaction will refer to the letters that appear in Figure 2.

4.1 Saving and Restoring of Register Context

In BG/Q TM, register checkpointing is done by the software in ⓐ. A straightforward way to save and restore the register context is to use the system `setjmp` and `longjmp` routines [21]. The system `setjmp` routine typically saves all registers into the context buffer and the corresponding `longjmp` routine restores the context by restoring all of these registers. Since the `setjmp` overhead is incurred every time a transaction is entered, the context-save overhead can be significant for small transactions. Therefore, the BG/Q runtime is specially optimized to reduce this overhead. Since rolling back a transaction is equivalent to performing a `longjmp` to the start of the transaction, three registers need to be restored in order to perform the `longjmp` and to establish the minimal context. The registers are the current value of the stack pointer, the global offset table (GOT) pointer – to a table that holds addresses for position-independent code, and a pointer to the first instruction of the register-restore code, which is placed immediately before the start of the transaction. These three registers, which comprise the minimal context information, are passed to the runtime and the runtime passes them down to the kernel because both the runtime and the kernel can rollback a transaction. The analysis to determine which other registers require saving and restoring is left to the compiler. The compiler uses live-range analysis to determine the set of live registers that are modified inside the transaction, to identify the registers that need to be saved and restored.

4.2 Managing Transaction Abort and Retry

The TM runtime activates a hardware transactional execution by writing to a memory-mapped I/O location. When the execution reaches the end of the transaction, it enters the TM runtime routine that handles transaction commit in ⓓ. The TM runtime attempts to commit the transaction. If the commit fails, the transaction is retried. If commit fails for a transaction T_A because of conflict with a transaction T_B, the runtime invalidates the speculation ID associated with T_A causing the hardware to clear the conflict register of T_B. Therefore, T_B now has a chance to commit successfully.

Capacity overflow and JMV prevent the thread from further execution, thus the transaction failure must be handled immediately. For JMV, the kernel invalidates the current transaction and invokes the restart handler. The restart handler *sends* the thread back to the start of the failing transaction.

For capacity overflow, the hardware invalidates the transaction and an interrupt is triggered in ⓖ. For conflict-induced transactional failures, the runtime can configure the hardware to trigger a conflict interrupt for *eager* conflict detection in ⓖ as well. This is the default setting of the runtime. Alternatively, conflict interrupts can be suppressed and conflicts are then handled *lazily* when the execution reaches the end of the transaction. A transactional failure due to capacity overflow is retried in the same way as conflict-induced failures because capacity overflow may be a transient failure.

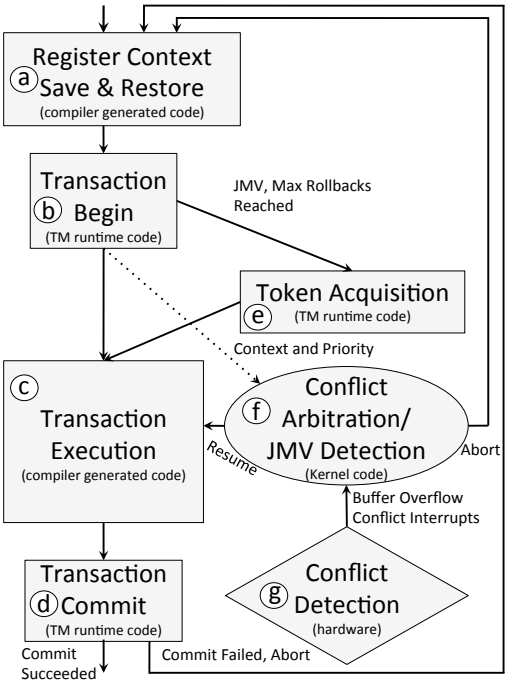

Figure 2: Transactional Memory Execution Overview

In the eager detection scheme, once the hardware triggers an interrupt, the interrupt handler in the kernel performs *conflict arbitration* by comparing the starting time of the conflicting threads and favours the survival of an older thread. When a transaction is started, the current time is recorded through a read of the timebase register and passed by the runtime to the kernel, which is used as the priority value during conflict resolution.

4.3 Sandboxing of Speculative Execution

System resources must be protected from being corrupted by a transaction that may later get aborted. BG/Q uses a *sandbox*, called *jail mode* to confine transactional executions and prevent speculative accesses to memory-mapped I/O space. These accesses include the ones for starting and stopping hardware transactional execution and the Direct-Memory Access (DMA) engine start operation. In jail mode, transactional accesses to protected TLB entries generate an access-violation exception called *Jail-Mode Violation* detected in ⓕ. Jail mode is entered and exited via system calls inside the `tm_begin` and `tm_end` runtime routines. By default, all transactions are executed in jail mode.

A second aspect of BG/Q sandboxing, which is orthogonal to the jail-mode, is that the interrupt handler in the kernel is, by default, capable of checking whether the triggering thread is speculative or not. System-level side effects of speculative execution, such as TLB misses, divide-by-zero and signals triggered by program fault cause the interrupt

handler to invalidate the current speculation and invoke the restart handler.

4.4 Ensuring Forward Progress via Irrevocable Mode

The TM programming model supports transactions of arbitrary size and with any computation including irrevocable ones. However, not all transactions expressed by the programming model can execute successfully by the underlying best-effort HTM. Therefore, one main functionality of the TM software stack is to ensure that a transaction eventually succeeds.

Such guarantee is provided by judiciously retrying failed transactions in a special mode called the *irrevocable mode*. Under the irrevocable mode, a transaction executes non-speculatively and can no longer be rolled back. To execute in the irrevocable mode, a thread must acquire the *irrevocable token*, that is associated with all `tm_atomic` blocks in the program. Interference of the irrevocable token with user-level locking may cause deadlock. Token acquisition in ⓔ is implemented using special BGQ L2 atomics. In essence, transactions executing under the irrevocable mode are serialized and the `tm_atomic` blocks behave like unnamed critical sections. It is however possible that there are concurrent speculative transactions and a transaction running in the irrevocable mode at the same time.

When a hardware transactional execution failure or a JMV exception occurs in ⓓ, the TM runtime determines how to retry the transaction. When the cause of a hardware transaction failure is transient, for example, due to a conflict, the TM runtime may retry transactional execution a few times before switching to the irrevocable mode. However, when a transactional execution failure is likely persistent, for example, due to a JMV, the TM runtime retries the transaction in the irrevocable mode.

4.5 Runtime Adaptation

The runtime employs a simple adaption scheme: it retries a failing transaction a fixed number of times before switching into irrevocable mode. After completion of the transaction in irrevocable mode, the runtime computes the serialization ratio of the executing thread. If the serialization ratio is above a certain threshold, the runtime records this transaction into a hash table. This hash table tracks *problematic* transactions. Once a transaction is entered into the hash table, its next execution will be speculative. Upon failing, it immediately switches into the irrevocable mode. This scheme allows a *problematic* transaction to have a single rollback. The amount of time that a transaction remains in the hash table is controlled via a runtime parameter.

5. EXPERIMENTAL RESULTS

This evaluation runs the STAMP benchmark suite [16] on a single 16-core, 1.6 GHz, compute node of a production BG/Q machine. We choose STAMP as our evaluation target because it is most widely used TM benchmark with largely coarse-grain transactions. The binaries are compiled by a prototype version of the IBM XL C/C++ compiler with the option of `-O3 -qhot -qsmp=omp -qtm`. The study reports the mean of five runs with an error bar. In the absence of more information, the measurements are assumed to be normally distributed. Thus, the length of the error bar is four standard deviations, two above and two below the mean, to approximate 95% confidence.

The baseline for speedups is always a sequential, non-threaded, version of the STAMP benchmark run with the 1-thread input. Table 1 shows that the large input, which is intended for runs on real hardware, is used. The comparison against Software Transactional Memories (STMs) uses TinySTM 1.0.3 [9].

[1]Input for `kmeans` is `inputs/random-n65536-d32-c16.txt`.

Benchmark	Running Options
bayes	-v32 -r4096 -n10 -p40 -i2 -e8 -s1
genome	-g16384 -s64 -n16777216
intruder	-a10 -l128 -n262144 -s1
kmeans_low	-m40 -n40 -t0.00001 -i ⟨input⟩[1]
kmeans_high	-m15 -n15 -t0.00001 -i ⟨input⟩[1]
labyrinth	-i inputs/random-x512-y512-z7-n512.txt
ssca2	-s20 -i1.0 -u1.0 -l3 -p3
vacation_low	-n2 -q90 -u98 -r1048576 -t4194304
vacation_high	-n4 -q60 -u90 -r1048576 -t4194304
yada	-a15 -i inputs/ttimeu1000000.2

Table 1: STAMP Benchmark Options

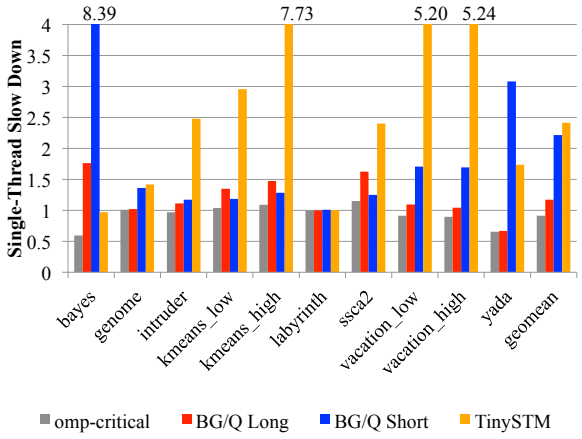

Figure 3: Relative execution time of OMP-Critical, BG/Q TM, and TinySTM over sequential version.

5.1 Single-thread TM performance

This section evaluates the overhead of TM execution running on a single-thread. Figure 3 shows the slow down of single-thread BG/Q TM under the short- and long-running modes relative to that of the sequential code. There is no noticeable run-to-run variations in performance for this experiment. For BG/Q TM, `TM_BEGIN` and `TM_END` macros of the STAMP benchmarks are replaced by BG/Q pragma annotation. Likewise, for `omp-critical`, the macros are replaced by `omp critical` pragma. TinySTM, uses the STM version of the benchmarks with extensive privatization to minimize instrumented states. Table 2 shows the number of L1 misses per 100 instructions as well as instruction path

lengths, which is the number of instructions executed, collected by hardware performance monitor.

The long-running mode incurs less overhead than the short-running mode for all STAMP benchmarks except for `ssca2` and `kmeans`, which have very small transactions (see Section 5.1.1).

Even using the best performing mode for each application, BG/Q TM still incurs noticeable overhead over the sequential version. The overhead comes primarily from increases in L1 cache misses, but, for some benchmarks, also from path length increases.

The relative speedup of the long-running mode over the sequential version in some benchmarks is the artifact of outlining `OMP` regions, which leads to reduced register pressure and allows the compiler to generate better code. Similar single-thread speedups are also observed on some `omp-critical` versions of the benchmarks.

The rest of the section explains in details the root causes of BG/Q TM overhead.

5.1.1 Cache performance penalty

One of the most dominant BG/Q TM runtime overhead is caused by the loss of L1 cache support due to the L1 flush and bypass needed for the bookkeeping of transactional state in L2. This observation is born by the number of L1 cache misses in short- and long-running modes relative to that of the sequential version. For instance, the three-fold slowdown for `yada` (Figure 3) in short-running mode is caused by 5 times as many L1 misses as the sequential version (Table 2). Similarly, the eight-fold slowdown for `bayes` is caused mainly by the 12× increase in L1 misses. Note that the L2 cache and L1P buffer load latencies are 13x and 5x higher than L1 load latency.

Under the long-running mode, L1 is flushed upon entering the transaction thus destroying any reuse between code before the transaction and code after the transaction. The locality penalty can be severe for small transactions with few memory accesses or little reuse. As shown in Table 2, the long-running mode suffers less L1 misses than the short-running mode for all but `ssca2` and `kmeans`. For `ssca2`, the short-running mode works well because the transactions mostly implement single-address atomic updates. For `kmeans`, both short- and long-running mode cause significant increase in the number of L1 misses because locality exists both within a transaction and across transactional boundaries.

While the long-running mode preserves more locality than the short-running mode for applications with not-so-small transactions, we still observe 15% to 38% increase in L1 misses compared to the sequential version.

5.1.2 Capacity overflow

Compared to STMs, one main disadvantage of a best-effort HTM is the bounded size of speculative states. Figure 4 shows the percentage of rollbacks caused by capacity overflow in relation to the total number of rollbacks experienced by each benchmark for various number of threads. Two of the STAMP benchmarks experience capacity overflow under single-thread TM execution. In `labyrinth`, capacity overflow is persistently triggered in one of its two main transactions, which unconditionally copies a global grid of 14M bytes inside the transaction. As a result, the most important transactions of `labyrinth` (accounting for 50% transactions) are executed in the irrevocable mode, resulting in a TM performance similar to `omp-critical`. For `bayes`, only 3% of the transactions trigger capacity overflow. However, due to the low serialization ratio, each failed transaction is retried 10 times before switching to the irrevocable mode, resulting in a significant (55%) increase in path length. This problem is not observed on `labyrinth` because the runtime adaptation is able to quickly limit the number of retries for consecutively failed transactions.

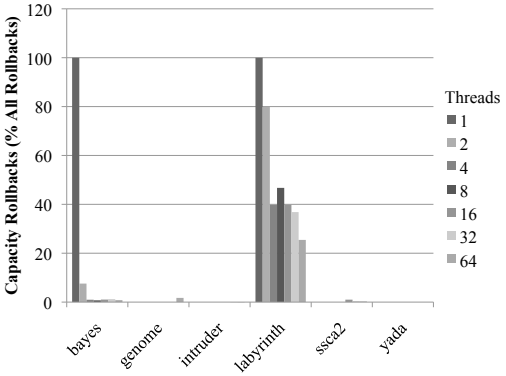

Figure 4: Percentage of rollbacks caused by capacity overflow

As shown in Figure 4, except for `labyrinth` and `bayes` which are known to exhibit large transactional footprint, capacity overflow on BG/Q is insignificant for most of the STAMP benchmark. At thread-counts above 16, `genome`, `intruder`, `ssca2` and `yada` experience some capacity overflow because of the shared speculative state, however never more than 2% of transaction rollbacks are caused by capacity overflow. The percent of executed transactions which experience capacity overflow in `bayes` and `labyrinth` decreases as the thread count increases because transactional conflicts become the leading cause of aborts.

5.1.3 Transaction enter and exit overhead

The basic overhead of entering and exiting a transaction comes from several sources: 1) saving of register contexts, 2) obtaining a spec ID, 3) the latency to write to memory-mapped I/O to start or commit a transaction, 4) the system calls to turn on and off kernel sandboxing, 5) additional runtime bookkeeping.

We measured the single-thread overhead of entering and exiting a transaction that implements a single atomic update. While this overhead for both BG/Q TM and TinySTM is in the order of hundreds of cycles, the overhead for BG/Q TM is smaller, about 44% of the TinySTM overhead for short-running mode, and 76% for long-running mode. Long-running mode has higher overhead because accesses to internal TM run-time data structures before and after transaction execution also suffer from L1 misses due to the L1 cache invalidation.

The transaction enter and exit overhead is also reflected in increases of instruction path length. As shown in Table 2, `ssca2` and `kmeans` experience the most path length increase under BG/Q TM (11% for ĕscans) due to the small size of its transactions.

Benchmark	# L1 misses per 100 instr. (thread=1)			Instr. path length relative to serial (thread=1)		
	sequential	BG/Q Short	BG/Q Long	omp critical	BG/Q Short	BGQ Long
bayes	0.6	8.1	0.7	87 %	155 %	156 %
genome	0.6	1.6	0.7	99 %	101 %	101 %
intruder	0.8	1.5	1.3	97 %	102 %	104 %
kmeans_low	0.1	0.3	2.7	101 %	105 %	106 %
kmeans_high	0.1	0.7	3.2	103 %	110 %	113 %
labyrinth	0.9	1.0	1.2	100 %	100 %	101 %
ssca2	1.0	0.7	1.3	96 %	109 %	111 %
vacation_low	1.5	6.2	2.4	88 %	92 %	93 %
vacation_high	1.7	7.0	2.4	88 %	91 %	92 %
yada	1.5	7.2	0.8	101 %	90 %	90 %

Table 2: Hardware performance monitor stats for the STAMP benchmarks.

5.2 HTM Scalability

Figure 5 shows the speedup of BG/Q TM relative to the sequential code for up to 64 threads. The rest of the experiments use long-running mode for all STAMP benchmarks except for `kmeans` and `ssca2` because short-running mode is only beneficial for very small transactions. In `kmeans` short-running mode can be up to 50% faster then long-running mode. In the experiments, each thread is bound to a different core on a round-robin breadth-first fashion.

Table 3 shows two metrics computed from transaction-execution statistics collected by the TM runtime:

- *Serialization ratio* is the percentage of committed transactions that were executed in the irrevocable mode. It is an indicator of the degree of concurrency in the TM execution.
- *Abort ratio* is the percentage of executed transactions that are aborted. It indicates the amount of wasted computation in the TM execution.

Both metrics reflect the combined effects of optimistic concurrency inherent to the program, hardware conflict detection granularity, and the retry adaptation of the TM runtime. A generally preferred scenario for TM is for an application to have both low serialization ratio and low abort ratio. A TM execution with both high serialization and high abort ratio is likely to show low scalability and even absolute performance slowdown. The lower abort ratio at higher number of threads, in Table 3, for some benchmarks (`labyrinth`, `kmeans`, and `vacation`) is counter intuitive. However, aborts caused by conflicts are highly dependent on the start and commit timing for the various transactions and therefore changing the number of threads may change this ratio in unexpected ways.

We first look at the relative speedup of TM execution and classify the benchmarks into several categories:

- **Effective HTM.** `Genome` and `vacation` exhibit good speedup and low serialization ratio. Note that performance boost beyond 16 threads comes from SMT threads multiplexing on the processor pipeline and hiding in-order processor stalls.
- **Spec-ID bottleneck.** Despite zero abort and serialization ratios, `ssca2` scales only up to 4 threads. This is because short and frequent transactions lead the system to quickly run out of spec-IDs, which blocks on spec-ID request until after some spec-IDs are recycled. BG/Q TM has 128 spec IDs that need to be recycled, based on the scrub interval, before being reassigned to new transactions. Figure 6 shows a sensitivity study on the impact of scrub interval on the performance of `ssca2`. With a scrub setting above 34, `ssca2` runs out of IDs and transactions are blocked waiting for an ID. And the scalability improves with lower intervals.
- **Contention bottleneck.** For `yada`, `bayes`, `intruder`, and `kmeans`, high serialization ratio at higher thread counts is the main bottleneck for scalability. The high serialization ratio is directly caused by the significant increase in aborts. Note that `bayes` exhibits high variability in the execution time because its termination condition depends on the order of its transactions commit.
- **Capacity bottleneck.** Due to the capacity problem (see Section 5.1.2), the main transactions of `labyrinth` are always executed in irrevocable mode and serialized with the irrevocable token. In the end, HTM execution exhibits no scaling and performs very close to naive locking.

5.3 Comparing against OMP-Critical and TinySTM

Using OMP critical (i.e. a big-lock) and using an STM represent the two end points of the spectrum of concurrency implementation. Big-locks have the lowest single-thread overhead, but are more likely to become a locking bottleneck. While STM has the best relative scaling (with respect to single-thread STM), but often exhibits the worst single-thread overhead.

The BG/Q TM fits in between these two ends both in terms of single-thread overhead and relative scalability. This is exactly what we observe. As shown in Figure 5 the scalability curves of `omp-critical` are mostly flat except for `kmeans` which suffers from little contention on a single big lock. When `omp-critical` scales (`kmeans`), it performs the best among the three because it has the lowest overhead. TinySTM, on the other hand, exhibits the best relative scalability with respect to single-thread TinySTM. Comparing BG/Q TM against TinySTM, we make the following observations:

- **Effective HTM.** For `genome` and `vacation`, BG/Q TM has both a steeper and a longer ascending curve

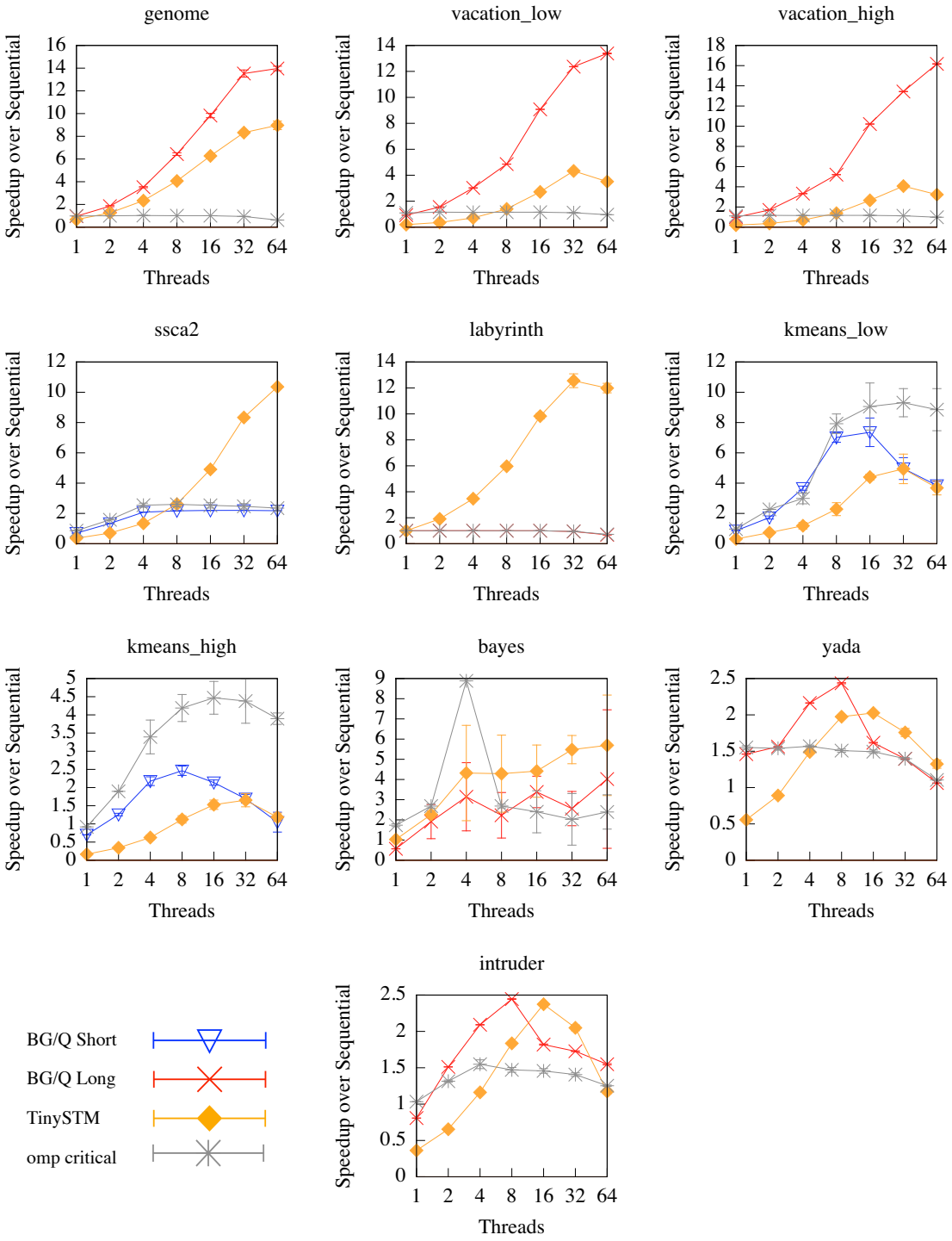

Figure 5: Speedup over sequential for upto 64 threads.

than TinySTM does. This is because, for these benchmarks, BG/Q TM does not suffer from any HTM-specific scaling bottlenecks. In addition, the lower overhead of BG/Q TM likely reduces the window of overlap among concurrent transactions.

- **Spec-ID and capacity bottleneck.** For `ssca2` and `labyrinth`, BG/Q TM peaks much earlier than TinySTM due to HTM-specific scaling bottlenecks.

- **Effective STM via privatization.** The good scaling of `labyrinth` and `bayes` on TinySTM is the result of a STM programming style that relies heavily on privatization and manual instrumentation. On the only two benchmarks with capacity overflow during a single-thread BG/Q TM execution, the STM codes in-

Benchmark	Serialization ratio (# threads)							Abort ratio (# threads)						
	1	2	4	8	16	32	64	1	2	4	8	16	32	64
bayes	3 %	4 %	53 %	64 %	85 %	84 %	84 %	23 %	39 %	50 %	55 %	62 %	73 %	73 %
genome	0 %	0 %	0 %	0 %	1 %	0 %	1 %	0 %	0 %	4 %	5 %	13 %	13 %	18 %
intruder	0 %	0 %	1 %	19 %	58 %	63 %	66 %	0 %	3 %	24 %	53 %	56 %	59 %	59 %
kmeans_low	0 %	0 %	0 %	1 %	7 %	19 %	82 %	0 %	0 %	3 %	21 %	59 %	65 %	59 %
kmeans_high	0 %	0 %	2 %	11 %	62 %	81 %	88 %	0 %	1 %	26 %	53 %	57 %	58 %	57 %
labyrinth	50 %	50 %	50 %	72 %	89 %	94 %	89 %	34 %	55 %	68 %	59 %	59 %	62 %	67 %
ssca2	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %
vacation_low	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %	15 %	8 %	8 %	17 %	26 %	20 %
vacation_high	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %	12 %	6 %	8 %	14 %	28 %	22 %
yada	0 %	5 %	7 %	11 %	41 %	53 %	53 %	0 %	39 %	43 %	49 %	49 %	52 %	52 %

Table 3: Percentage of irrevocable and aborted transactions in BG/Q TM execution.

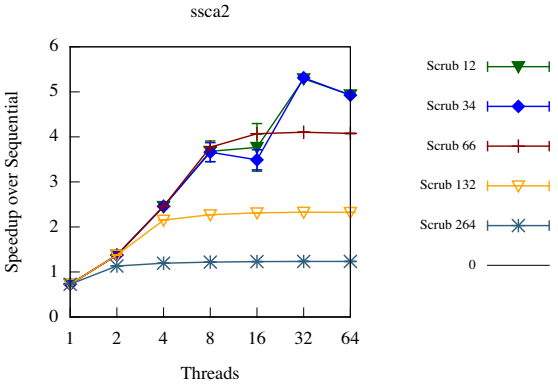

Figure 6: Effect of varying scrub intervals for ssca2.

cur no single-thread overhead because instrumented states are aggressively reduced to a tiny fraction of the actual footprint of the transactions.

- **Contention bottleneck.** For the rest of the benchmarks, BG/Q TM has a steeper ascending curve initially, but often peaks at a lower thread count and descends more rapidly afterwards in comparison to TinySTM. This may be explained by the bounded hardware implementation and the coarser granularity of conflict detection in BG/Q TM.

6. RELATED WORK

Despite of many HTM proposals in the literature, only two other real HTM implementations exist today, including the Rock processor [8] and the Vega Azul system [5]. While all three are best-effort HTMs, their design points differ drastically. Table 4 compares the key characteristics of the three systems in detail. The recently announced Intel's Haswell implementation of TSX is not included in Table 4 because that level of information is not yet available.

Both Rock HTM and Vega from Azul have small speculative buffers, compared to BG/Q's 20Mbytes of speculative states. Rock imposes many restrictions on what operations can happen in a transaction excluding function calls, divide, and exceptions. Rock also restricts the set of registers that functions may save/restore to enable the use of save/restore

HTM	BG/Q	Rock	Azul
Buffer capacity	20MB	32 lines	16 KB
Speculative buffer	L2	L2 cache	L1
Register save/restore	no	yes	no
Unsupported ISA	none	div, call, sync	none
Conflict detection	8-64B	n/a	32B
User-level abort	no	n/a	yes

Table 4: Basic features of real HTM implementations.

instructions that use register windows [8]. In contrast, in BG/Q, the entire instruction set architecture is supported within a transaction and register save/restore are a responsibility of the compiler and never cause transaction failure.

The method presented in this paper to build a software system to offer guarantee of forward progress on top of a best-effort HTM is an elegant solution that does not require that TM programmers provide an alternative code sequence for transaction rollbacks. Similar methods have been used before [5, 8] and should be useful for upcoming HTM systems [14, 15].

The TM system in Azul deals with more expensive transaction entry/exit operations by restricting speculation to contended locks that successfully speculate most of the time. Azul also avoids speculation on rollbacks. A transaction that has failed once is always executed non-speculatively. This is a contrast to the adaptive retry system described in this paper.

7. CONCLUSION

This detailed performance study of one of the first commercially available HTM systems has some surprising findings. Even though the single-thread overhead is reduced in comparison with STM implementations, it is still significant. The use of L2 to support TMs is essential to enable a sufficiently large speculative state. However, for many TM applications recovering the lower latency of L1 for reuse inside a transaction, through the use of long running mode in BG/Q, is critical to achieve performance. An end-to-end solution that delivers a programming model that supports the entire ISA delivers the simplicity promised by TMs.

Acknowledgments

The BG/Q project has been supported and partially funded by Argonne National Laboratory and the Lawrence Livermore National Laboratory on behalf of the U.S. Department

of Energy, under Lawrence Livermore National Laboratory subcontract no. B554331. The evaluation study reported in this paper was partially funded by the the IBM Toronto Centre for Advanced Studies and by grants from the Natural Sciences and Engineering Research Council of Canada. Thanks to Wang Chen for diligently managing the release of this paper through IBM legal process.

Trademarks

8. REFERENCES

[1] J. Bobba, K. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. In *International Conference on Computer Architecture (ISCA)*, pages 81–91, San Diego, CA, USA, 2007.

[2] C. C. S. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie. Unbounded transactional memory. In *High-Performance Computer Architecture (HPCA)*, pages 316–327, San Francisco, CA, USA, February 2005.

[3] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Communications of the Association for Computing Machinery*, 51(11):40–46, November 2008.

[4] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, and D. Grossman. ASF: AMD64 extension for lock-free data structures and transactional memory. In *Intern. Symposium on Microarchitecture (MICRO)*, pages 39–50, Atlanta, GA, USA, December 2010.

[5] C. Click. Azul's experiences with hardware transactional memory. In *HP Labs' Bay Area Workshop on Transactional Memory*, 2009.

[6] S. C. R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Simultaneous speculative threading: a novel pipeline architecture implemented in sun's rock processor. In *International Conference on Computer Architecture (ISCA)*, pages 484–495, Austin, TX, USA, 2009.

[7] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Principles and practice of parallel programming*, pages 67–78, Bangalore, India, January 2010.

[8] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 157–168, Washington, DC, USA, March 2009.

[9] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, December 2010.

[10] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Principles and practice of parallel programming*, pages 237–246, Salt Lake City, UT, USA, February 2008.

[11] T. M. S. D. Group. Draft specification of transactional language constructs for C++ (version 1.1), 2012.

[12] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim. The ibm blue gene/q compute chip. *IEEE Micro*, 32(2):48–60, March-April 2012.

[13] M. Herlihy and J. E. Moss. Transactional memory: Architectural support for lock-free data structures. In *International Conference on Computer Architecture (ISCA)*, pages 289–300, San Diego, CA, USA, May 1993.

[14] Intel Corporation. *Intel Architecture Instruction Set Extensions Programming Reference*, 319433-012 edition, February 2012.

[15] D. Kanter. Analysis of Haswell's transactional memory. http://www.realworldtech.com/page.cfm?ArticleID=RWT021512050738&p=1, February 2012. Real World Technologies.

[16] C. C. Minh, J. Chung, C.Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *International Symposium on Workload Characterization (IISWC)*, pages 35–46, Seattle, WA, USA, September 2008.

[17] 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 *International Conference on Computer Architecture (ISCA)*, pages 69–80, San Diego, CA, USA, 2007.

[18] 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 *Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, January 2006.

[19] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. In *ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 275–284, Munich, Germany, June 2008.

[20] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. Multiple reservations and the Oklahoma update. *IEEE Parallel & Distributed Technology (PDT)*, 1(4):58–71, November 193.

[21] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabataba. Code generation and optimization for transactional memory constructs in an unmanaged language. In *Code Generation and Optimization (CGO)*, pages 34–48, San Jose, CA, USA, March 2007.