Software Support and Evaluation of Hardware Transactional Memory on Blue Gene/Q

Amy Wang, Matthew Gaudet, Peng Wu, Martin Ohmacht, José Nelson Amaral, *Senior Member*, *IEEE*, Christopher Barton, Raul Silvera, and Maged M. Michael

Abstract—This paper describes an end-to-end system implementation of a transactional memory (TM) programming model on top of the hardware transactional memory (HTM) of the Blue Gene/Q machine. The TM programming model supports most C/C++ programming constructs using a best-effort HTM and the help of a complete software stack including the compiler, the kernel, and the TM runtime. An extensive evaluation of the STAMP and the RMS-TM benchmark suites on BG/Q is the first of its kind in understanding characteristics of running TM workloads on real hardware TM. 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.

Index Terms—Hardware transactional memory, runtime system, TM, performance evaluation

1 Introduction

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 [14], [28].

The following code snippet is an example of a typical transactional memory programming interface.

```
transaction
{
  a[b[i]] += c[i];
}
```

A *transaction* is a synchronization construct that allows operations inside a transaction to be executed as one (atomic) operation with respect to operations in other concurrent transactions. The semantics of a transaction can be implemented in several ways. The simplest implementation is to acquire and release a global lock when entering and exiting a transaction. Such an implementation, however, can be overly pessimistic in the amount of concurrency allowed. For instance, if all concurrent transactions in the previous example update different elements of array a, a global lock implementation would allow only one transaction to proceed, while ideally all non-conflicting transactions should be able to execute at the same time. Transactional memory is such a mechanism to allow maximal concurrency among non-conflicting transactions. The basic idea is to allow all transactions to execute speculatively and concurrently. During a speculative execution, the TM system will monitor and detect conflicts among memory accesses of all concurrent transactions. Once a conflict is detected, the TM system has the ability to *abort* one of the transactions as if the transaction was never executed and retry the transaction at a later time. In a nutshell, the ability to speculatively execute and abort a computation and to detect memory conflicts among transactions is the building block of any TM implementation.

There has been a long history of research exploitation of TM implementations. Given the high cost of implementing TM in hardware, the research community early on developed several implementations of software transactional memory (STM) [9], [11], [23], [26] and conducted simulation-based studies of hardware transactional memory (HTM) [2], [3], [21]. More recently real HTM implementations start to emerge. An early implementation of HTM was reported but 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 [8]. The specification of a hardware extension for TM in the AMD64 architecture has yet to be released in hardware [7]. Recently IBM [4], [13], [16] and Intel [15] disclosed that they are releasing implementations of HTM.

This paper studies and evaluates BG/Q HTM, one of the first commercially available HTM implementations today. We make three important contributions. First, it provides a detailed description of the BG/Q HTM implementation (Section 3) and an in-depth analysis of its major sources of overheads (Section 4). One large pain point of STM is the high overhead associated with monitoring memory accesses and maintaining speculative state inside a transaction [5]. While it is widely expected that transactional execution overheads can be significantly reduced in an HTM implementation, a surprising finding of this study is that the BG/Q HTM overhead, while much smaller than that of STM's, is still non-trivial. Some of the overheads are the result of hardware design

- *A. Wang, M. Gaudet, C. Barton, and R. Silvera are with the IBM Canada Software Laboratory, 8200 Warden Ave, Markham, Ontario L6G-1C7, Canada.*
- *P. Wu, M. Ohmacht, and M.M. Michael are with the IBM T.J. Watson Research Centre, Yorktown, NY.*
- *M. Gaudet and J.N. Amaral are with the Department of Computing Science, the University of Alberta, Edmonton, Alberta, Canada. E-mail: jamaral@ualberta.ca.*

Manuscript received 18 Dec. 2012; revised 04 Sep. 2013; accepted 06 Sep. 2013. Date of publication 16 Sep. 2013; date of current version 12 Dec. 2014. Recommended for acceptance by M. Parashar.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.190

choices. For instance, in order to allow transactions with a large memory footprint, the BG/Q HTM is implemented mainly in the L2 cache. To simplify the design of the in-core L1 without breaking transactional memory functionality, the L1 cache is either bypassed during a transactional execution or flushed when entering a transaction. The result is that a transaction executing on BG/Q HTM may suffer a loss of locality in the L1 cache.

Second, the paper presents a thorough evaluation of two TM benchmark suites—STAMP [20] and RMS-TM [18]—running on BG/Q TM (Sections 5 and 6). The performance study aims at answering the question of how effective BG/Q TM is to improve performance with respect to sequential execution as well as alternative concurrent implementations using locks and TinySTM [12]. The performance study leads to a division of typical concurrent applications into three categories. 1) There are applications that use medium-to-large transactions but that often execute successfully in the BG/Q HTM without many aborts. These applications are suitable for BG/Q HTM and can achieve good performance with little additional programming efforts. 2) There are applications that scale well with conventional locks, therefore should not use either STM or BG/Q TM because both incur a larger single-thread overhead than a lock-based implementation. 3) Some applications use small transactions that usually do not result in memory conflicts. These small transactions appear frequently, for instance, by residing inside a loop, and thus constitute the critical path of an application. Such applications may be better suited for STM because the single-thread overhead of an STM system may be compensated by the concurrency that the STM enables.

Third, the paper describes how the HTM support in BG/Q can be complemented by the software stack—which includes the kernel, the compiler, and the runtime system—to deliver the simplicity of a TM programming model (Sections 2 and 4). The HTM support in BG/Q is *best effort* in nature because not all computation may execute successfully in a hardware transaction. This limitation is mainly due to the boundedness of the hardware implementation, such as having a limited capacity to maintain speculative state during transactional execution. The TM software stack provides a fall-back mechanism to execute the transaction non-speculatively under a special mode called the *irrevocable* mode.

In terms of programmability, HTM is a clear win over STM. A TM programming model based on an STM implementation often requires the programmer to annotate codes and/or instrument memory references that may execute inside a transaction. The BG/Q TM programming model, on the other hand, is much simpler and requires only a block annotation of transactional codes. It is also worth pointing out the performance-productivity aspect of different TM implementations because there is a noticeable difference in the effort required to achieve good performance using STM versus HTM. For example, the STM version of the STAMP benchmark is manually instrumented to minimize the read- and write-set maintained by the STM in order to achieve a good performance, whereas the BG/Q TM version of these benchmarks is not.[1]

The rest of the paper is organized as follows. Sections 2, 3, and 4 describe the TM programming model, BG/Q HTM implementation, and the software stack that supports the TM programming model, respectively. Section 5 describes the evaluation methodology and the benchmarks. The performance study of BG/Q TM is presented in Section 6 (comparison between the short- and long-running modes), Section 7 (single-thread performance), and Section 8 (scalability). A discussion of related work appears in Section 9, and we conclude in Section 10.

2 Transactional Memory Programming Model

BG/Q provides a simple programming model based on the abstraction of *transaction*. The semantics of a transaction is similar to that of a critical section or a relaxed transaction as defined in [29]. In a concurrent execution, transactions appear to execute sequentially in some total order with respect to each other. Specifically, operations inside a transaction appear not to interleave with any operation from other concurrent transactions. Two transactions are nested if one transaction is entirely inside the other transaction. Nested transactions 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.[2] BG/Q TM, as a programming model, is privatization-safe, but not obstruction free because when a transaction fails to execute as a hardware transaction, it will be executed non-speculatively in the irrevocable mode, which may block the progress of other transactions (see Section 4.3).

The BG/Q TM programming model syntactically defines a transaction as a single-entry and single-exit code block using the annotation `#pragma tm_atomic`. The specification of transactional code region is orthogonal to the threading model, such as the use of OpenMP or pthreads. Any standard language construct is allowed in a transaction, and the computation inside a transaction can be arbitrarily large and complex. The only constraint is that the boundary of a transaction must be statically determinable in order for the compiler to insert proper codes to end a transaction. As a result, certain unstructured control-flow constructs that may exit a transactional block may result in a compile- or runtime error. Similarly, exceptions thrown by a transaction are unsupported.

Fig. 1 shows a critical section from a STAMP benchmark expressed in the BG/Q TM programming interface. Note the simplicity of this programming interface. In contrast, programming models based on STM implementations require more code annotations for the compiler and, to achieve good performance, often require careful manual instrumentation of memory accesses inside transactions.

3 Hardware Transactional Memory Implementation in BG/Q

In BG/Q each compute chip has 16 processor cores and each core can run four hardware Simultaneous Multi-Threaded

1. Note that BG/Q TM provides no mechanism to selectively allow non speculative memory accesses in a transactional execution.

2. The nesting support is implemented purely in software in the TM runtime.

```
for (i = start; i < stop; i++) {
   index = common_findNearestPoint(feature[i],
           nfeatures, clusters, nclusters);
   if (membership[i] != index) delta += 1.0;
   membership[i] = index;

   #pragma tm_atomic
   {
    new_centers_len[index]++;
    for (j = 0; j < nfeatures; j++) {
      new_centers[index][j] += feature[i][j];
    }
   }
}
```

Fig. 1. The main transaction of STAMP/kmeans benchmark using the BG/Q TM annotation.

(SMT) threads. A core has a dedicated 16K-byte L1 cache that is 8-way set-associative with a cache line size of 64 bytes and a 2K-byte prefetching buffer. All 16 cores share a 32M-byte L2 cache with a cache line size of 128 bytes.

BG/Q provides the following hardware mechanisms to support transactional execution:

- **Buffering of speculative state.** Stores made during a transactional execution form the *speculative state*. In BG/Q, transactional speculative state is buffered in the L2 cache and is only made visible (atomically) to other threads after a transaction commits.
- **Conflict detection.** During a transactional execution, the hardware detects read-write, write-read, or write-write conflicts among concurrent transactions and conflicts resulted from a transactional access followed by a non-transactional write to the same line. When a conflict is detected, the hardware sends interrupts to threads involved in the conflict that execute transactions. A special conflict register is flagged to record various hardware events that cause a transaction to fail.

The above hardware support is used to provide both ordered and unordered memory transactions on BG/Q. The former is also known as thread-level speculation (TLS) [27]. Since the BG/Q support for TLS is beyond the scope of this paper, the rest of the paper focuses exclusively on the unordered transactional-memory support of BG/Q.

3.1 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 state, the L2 cache can store multiple versions of the same physical memory line. Each version occupies a different L2 way.

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 until either it is made visible to other threads when the transaction commits or it is discarded when the transaction is aborted.

For each access, the L2 directory records whether it is read or written and whether it is speculative. For speculative accesses, the L2 directory also tracks which thread has read or written the line by recording a unique ID, called the *spec-ID*, associated with the transaction. This tracking provides the basic bookkeeping to detect conflicts among transactions and between transactional and non-transactional accesses.

The commit of a transaction is done by the hardware in two phases. First a central speculation control unit notifies all L2 slices of its intention to commit a spec-ID and waits for responses. Slices that acknowledge the feasibility of a commit enter a fail-prevention state that disallows any action that may disrupt the on-going commit. After collecting all responses, the central unit notifies all slices whether the commit is successful or not. The commit latency is defined by the round-trip latency from the cores to the central speculation control unit, which is about 100 cycles and can be fully pipelined, and the duration of the two-phase commit, which is about 18 cycles.

At the hardware level, an abort has practically no overhead because it requires the issuing of a single store to invalidate the spec-ID. However, the detection of a conflict, the invocation of the software handler, and the recycling of spec-IDs do have a cost. Moreover, conflicts detected by the L2 slices are reported to the cores as they occur, setting a flag in a core accessible register. This register may be polled by the software during lazy conflict detection, which takes about 30 cycles.

BG/Q provides 128 spec-IDs to distinguish memory accesses made by concurrent transactions. Each new transaction, including retrying transactions, needs to apply for a spec-ID when it starts. If the system runs out of available spec-IDs, the start of the transaction is blocked until a spec-ID becomes available. When a spec-ID is invalidated, it is still stored in the L2 slices' directories and it needs to be removed before it can be re-used. Invalid spec-IDs are removed whenever a load or store accesses the set that contains the spec-ID. An automatic background scrub accesses sets at a programmable rate—with a minimum of 12 cycles between set visits—to reclaim invalid spec-IDs. At predetermined intervals, the L2 cache examines all cache lines and checks whether they are associated with spec-IDs from transactions that are either aborted or committed. After all lines associated with a spec-ID are either marked as invalid or merged with the non-speculative state (i.e., committed), the spec-ID is reclaimed and made available again. This reclamation process is called *spec-ID scrubbing*. The interval between two starts of the scrubbing process is the *scrubbing interval*. The default scrubbing interval is 132 cycles but can be altered by the runtime via a system call. Note that setting the scrub interval too high may lead to the blocking of new transactions, while setting it too low may cause more interference to normal operations of the L2 cache.

The buffering of speculative state in the L2 requires cooperation from components of the memory subsystem that are closer to the processor pipeline than the L2, namely, the L1 cache and the L1 prefetcher (L1P).[3] In BG/Q there is little hardware modification to support transactional execution in the L1 because it uses a pre-existing core design. As such, BG/Q supports two transactional execution modes for proper interaction between the L1, the 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 transactional write invisible to other threads that share the same L1.

3. Prefetched data may evict speculative state from L2 leading to unnecessary aborts.

- **Short-running mode (via L1-bypass).** In this mode, when a transaction 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. As the L2 stores multiple values for the same address, it is able to return the thread-specific data along with a flag that instructs the core to place the data directly into the register of the requesting thread, without storing 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 state can be kept in the L1 cache. The L1 cache can store up to 5 versions, 4 transactional ones for the 4 SMT threads and a non-transactional one. To achieve this, the software creates an illusion of versioned address space via Translation Lookaside Buffer (TLB) aliasing. The TLB translates virtual into physical memory addresses. The illusion created allows a single virtual address to be translated to multiple physical addresses at the L1 level. For each memory reference issued by a transaction, some bits of the physical address in the TLB are used to create an aliased physical address by the memory management unit. Therefore, the same virtual address may be translated to four different physical addresses for each of the four threads that share the same L1 cache. However, when 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 multi-version through the bookkeeping of spec-IDs. The L1 cache is invalidated upon entering a transaction. Such invalidation makes all first transactional accesses to a memory location visible to the L2 as an L1 load miss.

The short- and long-running modes are designed to exploit different locality patterns. The long-running mode is the default running mode, but 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 the locality within a transaction. However, by invalidating L1 at the start of a transaction, it prevents reuse between codes that run before entering the transaction, and codes that run within the transaction, or after the transaction ends. Thus, this mode is best suited for long-running transactions with plenty of intra-transactional locality.

3.2 Causes of Transactional Execution Failures

BG/Q supports bounded and best-effort transactional execution. A hardware transaction may fail in the following scenarios:

- **Transactional conflicts** are detected by the hardware at the L2 cache level as described earlier. The *conflict-detection granularity* is the minimum distance between two memory accesses distinguishable by the conflict detection system. That is, accesses closer than the granularity may be flagged as a conflict even when there is no actual overlap. 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** causes a transaction to fail when the L2 cache cannot allocate a new way for a speculative store. By default, the L2 guarantees 10 of its 16 ways to be used for speculative storage without an eviction.[4] Therefore, up to 20M-bytes $(32\mathrm{M}^{*}10/16)$ of speculative state can be stored in the L2. A set may contain more than 10 speculative ways if speculative ways have not been evicted by the least-recently-used replacement policy. In practice, capacity failures may occur at a much smaller speculative-state footprint, for instance, when the number of speculative stores mapped to the same cache set exceeds the number of ways available in the set.
- **Jail mode violation (JMV)** occurs when a transaction performs *irrevocable actions*, that is, operations whose side-effects cannot be reversed, such as writes to I/O-device address space. Irrevocable actions are detected by the kernel under a special mode called the *jail mode* and lead to a JMV interrupt to the owner thread of the event.

4 Software Support for TM Programming Model

While a computation may fail in a hardware transactional execution in various ways, a transaction, as defined by the programming model, is guaranteed to eventually succeed. The TM software stack is developed to bridge the gap between the TM programming model and the hardware TM implementation. The software stack includes the TM run-time system, extensions to the kernel, and the compiler.

Fig. 2 illustrates the main state transition flow of the TM software stack. Register check-pointing is a necessary step to restore register state during a transaction rollback. Since BG/Q does not support hardware register check-pointing, this functionality is implemented in software as Step ⓐ of Fig. 2.

The task of determining which other registers require saving and restoring is left to the compiler. The compiler uses live-range analysis to determine the set of registers that are modified inside a transaction and remain live after the transaction commits, and generates codes to check-point these registers.

4.1 Managing Transaction Abort and Retry

The TM runtime activates a hardware transactional execution by writing to a memory-mapped I/O location. When a transaction is started, the current time is recorded through a read of the timebase register. This recorded time is then used by the kernel as a priority value during conflict resolution. When the execution reaches the end of the transaction, it enters the TM runtime routine that handles transaction commit in Step ⓓ. The TM runtime attempts to commit a transaction. If the

4. This default can be changed but it is advisable to leave a reasonable number of ways for other threads using this shared cache in a non-speculative way.

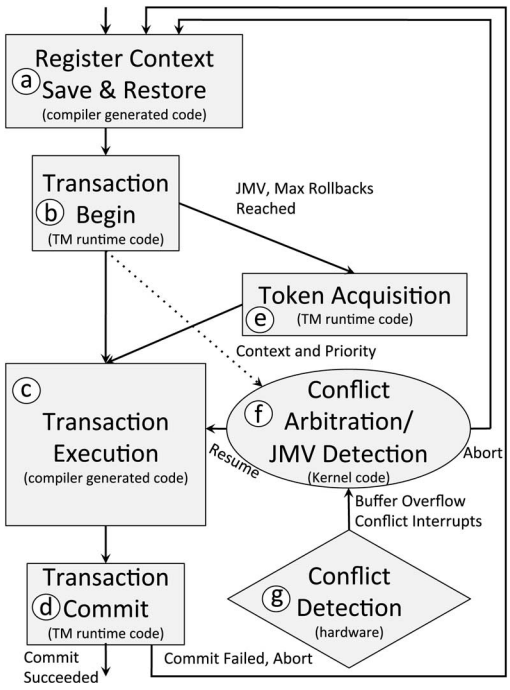

Fig. 2. Transactional memory execution overview.

commit fails, the transaction is invalidated (by invalidating its spec-ID) and retried at a later time. Specifically, if a transaction T_A fails to commit due to a conflict with another transaction T_B, the runtime invalidates the spec-ID associated with T_A, by executing a store to the status control register and a store to the conflict register—both in the central speculation control unit[5], causing the hardware to clear the conflict register of T_B so that T_B now has a chance to commit.

For transactional failures caused by memory conflicts, the runtime can configure the hardware to trigger a conflict interrupt for *eager* conflict detection as shown in Step ⓖ. Under the eager detection scheme, once the hardware triggers an interrupt, the interrupt handler performs *conflict arbitration* by comparing the starting time of the conflicting transactions and favours the survival of an older transaction. Alternatively, transactional conflicts can be detected *lazily* when the execution reaches the end of a transaction. Lazy detection is achieved by suppressing interrupts caused by conflicts with other transactions and by relying on the runtime to check the status of the conflict register before committing a transaction. The lazy conflict detection scheme cannot suppress interrupts caused by conflicts with non-transactional accesses. Such interrupts are necessary to ensure the strong-isolation guarantee of BG/Q HTM so that a transaction will not observe any inconsistent state. By default, the TM runtime uses the eager conflict detection scheme.

For transactional failures caused by capacity overflow, the hardware immediately aborts the transaction and triggers an interrupt. A failed transaction due to capacity overflow is retried in the same way as a failed transaction due to conflicts because capacity overflow may also be a transient failure.

For transaction failures caused by JMV, the kernel immediately aborts the current transaction and invokes the restart handler. The handler restores the appropriate context, transfers the execution back to the start of the failing transaction, and executes the transaction in the irrevocable mode.

5. These two stores are fully pipelined and the core does not need to wait for their completion.

4.2 Sandboxing of Speculative Execution

Since transactional execution is speculative by nature, critical system resources must be protected from being corrupted by a transaction that is later aborted. BG/Q uses a sandbox called the *jail mode* to prevent speculative transactions from performing irrevocable actions. The jail mode is entered and exited via system calls during the start and commit/abort of a transaction. There are two forms of irrevocable actions: writes to protected address space and system calls. Under the jail mode, protected address space such as the memory-mapped I/O space is indicated in the TLB. Any access to protected TLB entries as well as system calls under the jail mode generate an access-violation exception called *Jail-Mode Violation* (JMV). This is shown as Step ⓕ in Fig. 2.

The kernel also provides sandboxing during interrupt handling. The interrupt handler always checks whether the thread triggering an interrupt is speculative or not. System-level side effects during a transactional execution—such as TLB misses, divide-by-zero and signals triggered by program fault—cause the interrupt handler to abort the transaction and invoke the restart handler.

4.3 Ensuring Forward Progress via Irrevocable Mode

The TM software stack ensures 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 a single lock called the *irrevocable token* that is associated with all `tm_atomic` blocks in the program. Token acquisition, as shown in Step ⓔ, is implemented using BG/Q's fast L2 atomic operations. Transactions executing under the irrevocable mode are essentially serialized by a single lock and behave like unnamed critical sections. Interference between the irrevocable token and user-level locking may cause deadlock. In some cases, however, certain degree of concurrency can be allowed between a speculative transaction and an irrevocable transaction. Such concurrency is possible because speculative transactions acquire the irrevocable token at the end of the transaction, whereas irrevocable transactions acquire the irrevocable token at the beginning of the transaction. For instance, if a speculative transaction T_A reads the token after a concurrent irrevocable transaction T_B releases the token, both T_A and T_B can commit successfully while overlapping much of their execution.

4.4 Runtime Adaptation

How to retry a transaction can have a significant impact on BG/Q TM performance. Unlike STM systems that have almost unlimited resources, too many immediate retries can lead to serious resource contention for BG/Q TM such as the depletion of the number of available spec-IDs.

To address this issue, the runtime employs a simple adaption scheme: it retries a failed transaction a fixed number of times before switching to the irrevocable mode. After the completion of a transaction in the irrevocable mode, the

runtime computes a metric called the *serialization ratio*, which is the percentage of total transactions executed in the irrevocable mode, for the executing thread. If the serialization ratio is above a threshold, the runtime records this transaction into a hash table that tracks *problematic* transactions. Once a transaction is entered into the hash table, the next time the transaction fails, it will be retried immediately in 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 Setup and Benchmarks

This evaluation of the BG/Q TM performance uses two benchmark suites. The STAMP benchmark suite [20] is the most widely used TM benchmark and has largely coarse-grain transactions. The RMS-TM benchmark suite consists of 7 real-world applications from the Recognition, Mining, and Synthesis (RMS) domain [18]. Each benchmark provides the original sequential code and the parallel codes using different critical-section implementations including pthread lock, OpenMP critical section, and HTM. For the HTM implementation, the `TM_BEGIN` and `TM_END` macros were replaced by BG/Q TM pragmas. For the OpenMP critical-section implementation, the macros were replaced by `omp critical` pragma. The STM version of the STAMP benchmarks uses TinySTM 1.0.3 [11] and is manually instrumented to minimize the amount of tracked state. Table 1 summarizes the benchmarks and the running options. All runs use the large input set for the STAMP benchmarks.

All experiments run on a single 16-core, 1.6 GHz, compute node of a production BG/Q machine. The binaries are compiled by a prototype version of the IBM XL C/C++ compiler. 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. When reporting the speedups, the baseline is always a sequential, non-threaded version of the benchmark running with the one thread input.

To build a model of expected speedups for various critical-section implementations, we evaluate two critical section characteristics of the parallel benchmarks running in a single-thread execution.

- *Relative critical section size.* This metric measures the ratio between the time spent in critical sections and the time spent in parallel regions during a single-thread execution of the code. Relative critical section size is an indicator of how much the serialization of critical sections would limit the concurrency in the parallel execution.
- *Absolute critical section size.* This metric measures the average time spent (in cycles) per dynamic instance of critical sections during a single-thread execution of the code. The absolute critical section size is an indicator of the size of a dynamic transaction.

Fig. 3 shows the absolute critical section sizes of both benchmark suites in a log scale. This metric helps to reason about the transactional footprint of a benchmark. In general, benchmarks with a larger absolute critical section size, such as `labyrinth`, tend to have a larger transactional footprint.

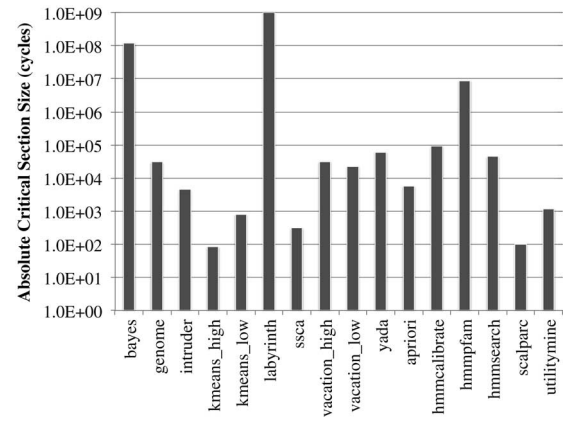

Fig. 3. Average time spent (in cycles) per dynamic instance of critical sections in the STAMP and RMS-TM benchmark suites.

As shown in Table 1, the relative critical section sizes of the two benchmark suites differ significantly. While many STAMP benchmarks spend more than 50% of the parallel region in critical sections, all RMS-TM benchmarks, except `utilitymine`, spend a tiny fraction of the parallel region in critical sections.

To better understand characteristics of applications running on BG/Q TM, we instrumented the TM runtime to collect the following statistics:

Transaction serialization ratio is the percentage of total committed transactions that are executed in the irrevocable mode. This metric is an indicator of the degree of concurrency in a TM execution.

Transaction abort ratio is the percentage of total executed transactions that are aborted. This metric is an indicator of the amount of wasted computation in a TM execution.

6 Long- Versus Short-Running TM Mode

This section focuses on understanding the performance implications of the short-running (SR) and the long-running (LR) modes of BG/Q TM. It turns out that choosing the right running mode is an important aspect of performance tuning for BG/Q TM. Altering the running mode of a BG/Q node involves a system call that requires the node to be in a certain state. Therefore, the running mode is only specified via an environment variable at the start of the program and may not be changed during the execution of the program.

Fig. 4 shows the speedup of BG/Q TM running under the SR and LR modes over the sequential baseline.

The relative performance between the SR and LR modes corresponds nicely with the absolute critical section sizes of the benchmarks. For example, the SR mode performs better than the LR mode for benchmarks `ssca2`, `fluidanimate`, `kmeans`, and `utilitymine`. All of those benchmarks use short-running transactions that are reflected as relatively small absolute critical section sizes in Fig. 3. Likewise, the LR mode outperforms the SR mode for the rest of the benchmarks that use relatively long transactions. In fact, executing a long-running transaction in the SR mode may result in serious performance degradation from the LR mode as shown in the case of `vacation` and `genome`.

The rest of the section examines three factors that contribute to the performance difference between the LR and the SR

TABLE 1
Benchmark Descriptions

Suite	Benchmark	Description	Running Options	Relative critical section size
STAMP	bayes	Machine learning. Learns a Bayesian net.	-v32 -r4096 -n10 -p40 -i2 -e8 -s1	100%
	genome	Genomic sequencing	-g16384 -s64 -n1677721	99.7%
	intruder	Network security simulation	-a10 -l128 -n262144 -s1	66.6%
	kmeans (low)	Clustering	-m40 -n40 -t0.00001 -i ⟨input⟩	2.8%
	kmeans (high)	Clustering	-m15 -n15 -t0.00001 -i ⟨input⟩	5.06%
	labyrinth	Maze solver	-i inputs/random-x512-y512-z7-n512.txt	100%
	ssca2	Kernel 1 from SSCA2 in HPCS [1]	-s20 -i1.0 -u1.0 -l3 -p3	16.6%
	vacation (low)	Simulates travel reservations	-n2 -q90 -u98 -r1048576 -t4194304	94.6%
	vacation (high)	Simulates travel reservations	-n4 -q60 -u90 -r1048576 -t4194304	95.0%
	yada	Delauney Mesh Refinement	-a15 -i inputs/ttimeu1000000.2	100%
RMS-TM	apriori	Association rule mining algorithm	⟨input⟩ -s 0.0075	0.05%
	fluidanimate	Hydrodynamics simulation	⟨threads⟩ 5 in_300K.fluid	NA
	hmmcalibrate	Calibrates genome sequence profile model	–num 500 –seed 33 globin.hmm	1.3%
	hmmpfam	Hidden Markov Model Database search	Pfam_ls_300 7LES_DROME	8.7%
	hmmsearch	Finds similar sequences from a database	globin.hmm 2000_uniprot_sprot.fasta	0.5%
	scalparc	Decision Tree Algorithm	F26-A32-D125K.tab 125000 32 2	0.01%
	utilitymine	Rule mining algorithm	⟨input⟩ logn1000_binary 0.01	35%

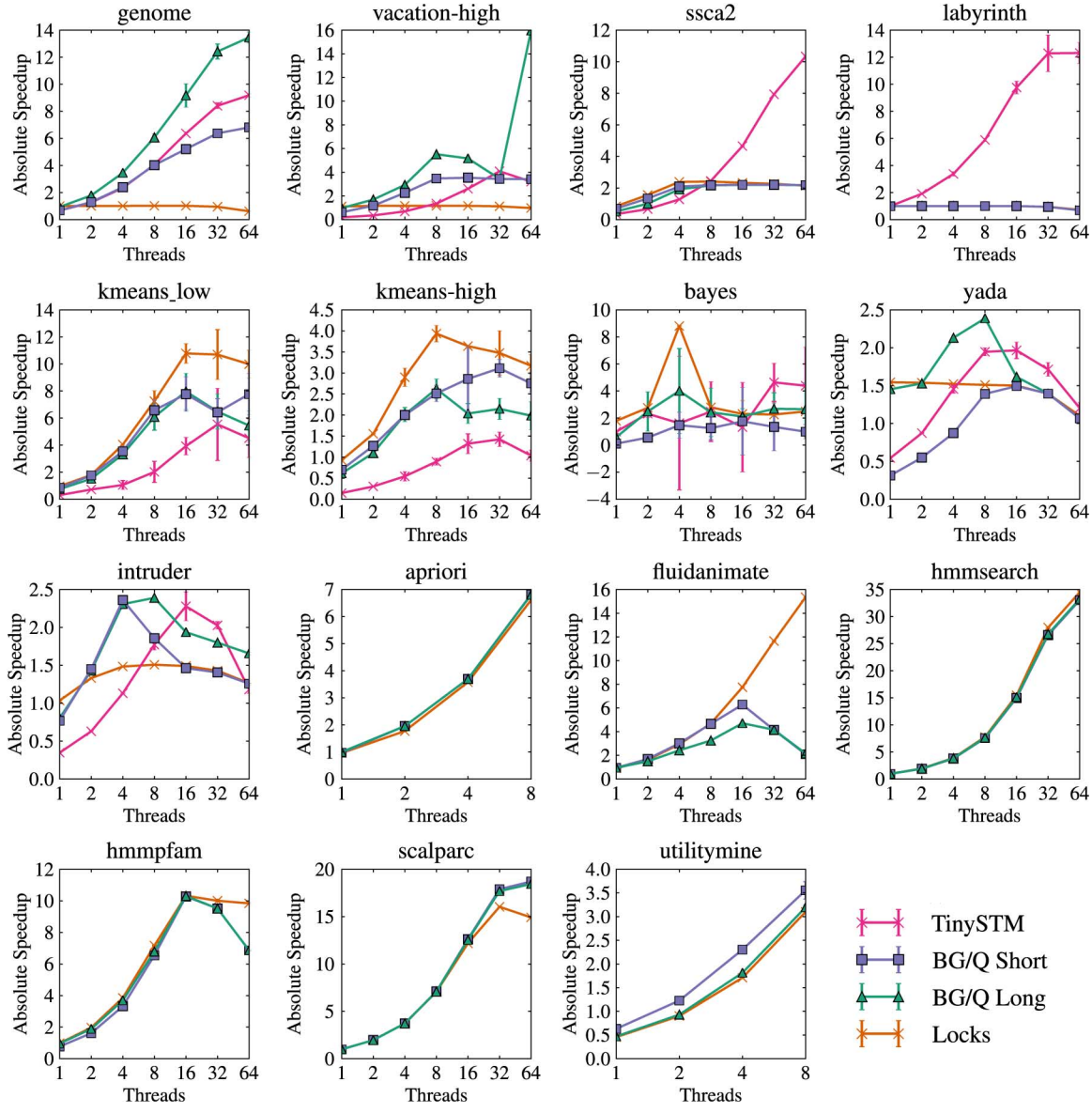

Fig. 4. Speedup of different critical section implementations of the STAMP and RMS-TM benchmark suites over the original sequential version of the benchmarks (vacation−low results were similar to vacation−high and hmmcalibrate were similar to hmmsearch—both are omitted).

TABLE 2
Hardware Performance Monitor Stats

Benchmark	# L1 misses per 100 instructions (thread=1)			Instruction path length relative to serial (thread=1)		
	sequential	BG/Q Short	BG/Q Long	omp critical/lock	BG/Q Short	BGQ Long
bayes	1.0	12.2	0.8	71.5 %	231.7 %	219.3 %
genome	0.5	1.8	0.7	99.9 %	101.0 %	101.3 %
intruder	1.2	2.6	2.1	96.7 %	105.2 %	108.1 %
kmeans_low	0.1	0.3	2.6	101.7 %	104.8 %	106.1 %
kmeans_high	0.2	0.7	3.2	103.8 %	110.6 %	113.5 %
labyrinth	0.9	1.0	1.0	99.5 %	100.2 %	100.2 %
ssca2	3.1	1.8	4.5	122.4 %	175.9 %	193.3 %
vacation_low	1.9	8.1	3.0	89.3 %	92.8 %	93.9 %
vacation_high	2.0	8.5	2.9	89.2 %	91.7 %	92.5 %
yada	0.9	19.9	0.7	73.5 %	74.6 %	74.9 %
apriori	2.4	2.4	2.4	105.6 %	103.3 %	100.2 %
fluidanimate	0.2	0.2	0.2	118.6 %	106.6 %	106.6 %
hmmcalibrate	0.5	0.6	0.5	100.0 %	100.0 %	100.0 %
hmmpfam	1.0	1.9	1.0	100.0 %	100.7 %	100.7 %
hmmsearch	0.5	0.6	0.5	100.0 %	100.0 %	100.0 %
scalparc	0.5	0.5	0.5	101.9 %	102.3 %	99.9 %
utilitymine	2.4	2.1	4.6	174.7 %	141.5 %	145.0 %

modes: loss of L1 cache locality, capacity overflow, and conflict-detection granularity.

6.1 Loss of Cache Locality

There are significant differences in L1 cache behaviors under the LR and the SR modes. When a transaction is executed in the LR mode, the L1 cache is flushed before starting the transaction. The L1 cache flush destroys any locality between codes executed before and after entering a transaction. For short-running transactions, the performance penalty of flushing the L1 cache can be severe, therefore the SR mode is better suited for such transactions. On the other hand, when a transaction is executed in the SR mode, the L1 cache is bypassed, which prevents locality of access within a transaction from benefiting from the L1. For long-running transactions, the performance penalty of bypassing the L1 cache during transactional execution can be severe, therefore the LR mode is better suited for such transactions.

Hardware performance counter statistics collected for all the performance runs validate this explanation. Table 2 shows the number of L1 misses per 100 instructions and the instruction-path-length statistics[6] of the benchmarks running under different configurations. As shown in Table 2, the LR mode suffers from much fewer L1 misses than the SR mode for all but `ssca2`, `kmeans`, and `utilitymine`. These three benchmarks all use small transactions according to the measured absolute critical section sizes. `Kmeans` has a significant increase in L1 misses for both the SR and the LR modes over the sequential baseline. This increase is because `kmeans` has locality of access both within and across transactions.

6.2 Capacity Overflow

Due to hardware implementation differences of the two running modes, the SR mode triggers significantly more capacity overflows than the LR mode. Figs. 6 and 5 show the percentage of total transactional executions that are aborted due to capacity overflow for the SR and LR modes, respectively. Benchmarks without any capacity overflow are omitted from the figures.

As shown in Fig. 5, under the LR mode, only two benchmarks, `labyrinth` and `bayes`, experience significant capacity overflow. The capacity overflow in `labyrinth` is persistently triggered in one of its two main transactions that involves the copying of a global grid of 14M bytes. As a result, 50% of the transactions in labyrinth experience capacity overflow. For `bayes` only 3% of committed transactions trigger capacity overflow. However, each of the aborted transaction with capacity overflow is retried up to 10 times, resulting in close to 25% of the transactions in `bayes` with capacity overflow. The percentage of executed transactions with capacity overflow in `bayes` and `labyrinth` decreases as the thread count increases. This is because transactional conflicts become the leading cause for a transaction to abort. An insignificant amount of capacity overflow occurs in `genome`, `intruder` and `yada` running with more than 16 threads. This is due to the limited number of ways in L2 for speculative writes by concurrent threads.

As shown in Fig. 6, the SR mode exhibits significantly more capacity overflow than the LR mode because of hardware implementation issues. Under the SR mode, the hardware state used to indicate capacity overflow is also used to indicate another hardware event: a race at the L2 between hit notifications from the L1 of multiple cores. In such a situation, an

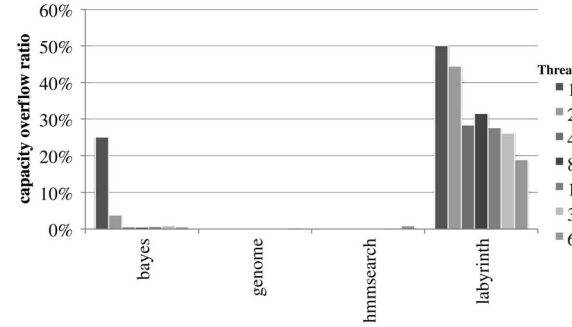

Fig. 5. LR mode: ratio of total transactions aborted due to capacity overflow.

6. Instruction path length is measured as the total number of dynamic instructions executed in the parallel region.

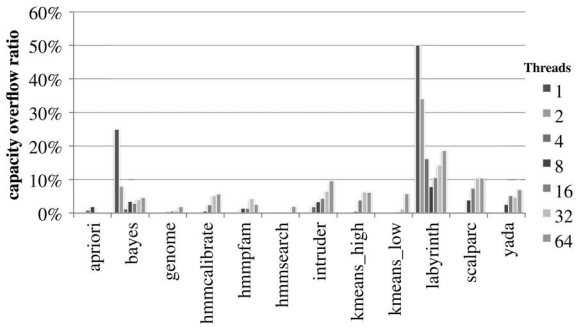

Fig. 6. SR mode: ratio of total transactions aborted due to capacity overflow.

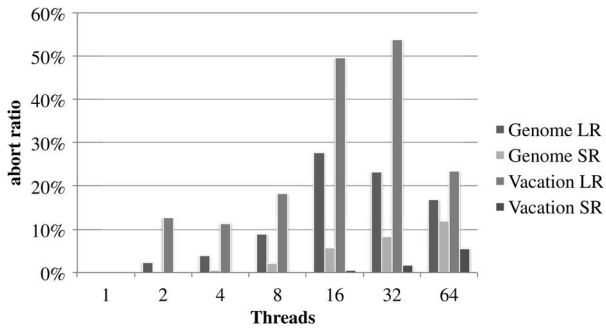

Fig. 7. The abort ratio of `genome` and `vacation`.

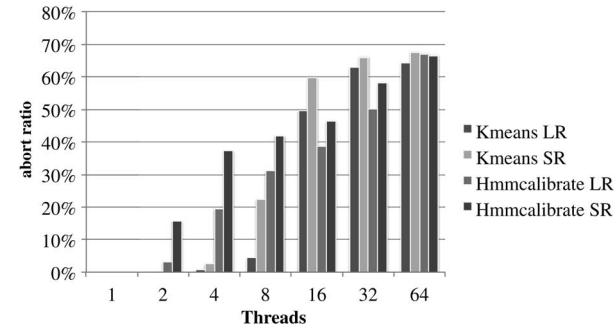

Fig. 8. The abort ratio of `kmeans` and `calibrate`.

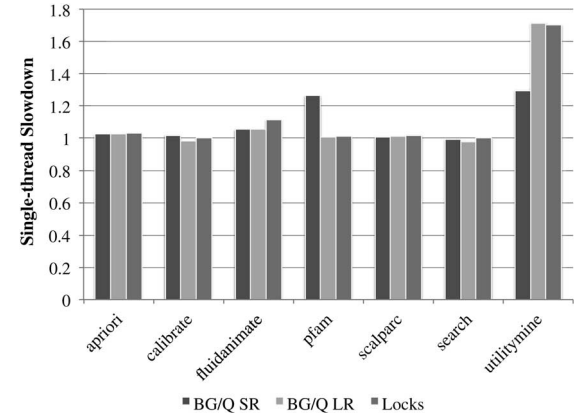

Fig. 9. Single-thread slowdown of the RMS-TM benchmarks.

abort is triggered because the hardware cannot determine the precedence between the hits. This abort occurs because the hardware must establish the ordering amongst committing transactions. Even though such ordering is not required for TM, it is implemented as such because the same hardware is also used to support TLS where such ordering is necessary.

6.3 Conflict Detection Granularity

The SR and the LR modes use different conflict detection granularity that could result in different number of transactional aborts. For instance, the SR mode detects conflicts at an 8- or 64-byte granularity depending on the number of concurrent accesses to the same cache line. The LR mode detects conflicts at a 64-byte granularity at best.

One would expect that the SR mode with a finer conflict detection granularity would trigger fewer transaction aborts than the LR mode. This is the case for `vacation` and `genome` where the abort ratio (measured as the percentage of total executed transactions that are later aborted) of the two benchmarks under the LR mode is several times higher than that of the SR mode. This also means in this case, it is more prudent to preserve intra-transactional locality offered by the LR mode, despite of its coarser granularity. Fig. 7 shows the abort ratio of `vacation` and `genome`.

However, for the rest of the benchmarks, the abort ratio of the SR mode is in fact higher than that of the LR mode, especially on benchmarks using small transactions such as `kmeans` and `hmmcalibrate`. The abort ratio of the latter two benchmarks is shown in Fig. 8. This may seem counter intuitive because we expect that, with a finer conflict detection granularity, the SR mode should reduce the number of false conflicts and consequently the number of aborts. There are three other factors that may affect the abort ratio. First, the SR mode may trigger more capacity overflow (as described in Section 6.2), as is the case for `kmeans`, `hmmcalibrate`, `hmmpfam`, and `scalparc`. Second, the SR mode may run slower because of the longer latency to satisfy read-after-write dependences, resulting in a longer overlapping window among transactions, thus causing more aborts. Third, runtime adaptation may affect how many times a transaction is retried, especially for those aborted due to capacity overflow.

7 Single-Thread TM Overhead

When parallelizing a program, one needs to be mindful of the overhead introduced by parallelization and synchronization. While this is true for parallel execution, such overhead may also manifest in the single-thread execution of a parallel code, especially when TM is used for synchronization. The slowdown caused by a single-thread execution of a parallel code over the execution of the sequential code is the *single-thread overhead*. This section studies the single-thread overhead of BG/Q TM in comparison to those of STM and locks.

Fig. 9 shows the single-thread overhead of the RMS-TM benchmarks. The single-thread overhead of both BG/Q TM and locks is insignificant except for `pfam` running under the SR mode and `utilitymine`. This is because the critical sections of the RMS-TM benchmarks are relatively small compared to the overall parallel regions (as shown in Fig. 3). There is an anomaly in `utilitymine` where the lock implementation increases the instruction path length by more than 70% in a single-thread execution (as shown in Table 2).

Fig. 10 shows the single-thread overhead of parallel implementations of the STAMP benchmarks. The single-thread overhead of TinySTM is significantly higher than that of other

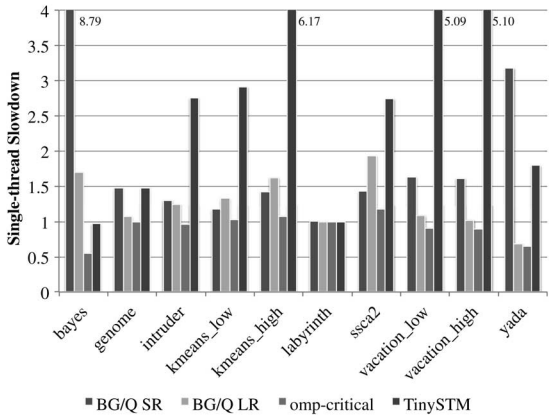

Fig. 10. Single-thread slowdown of the STAMP benchmarks.

implementations. This is because STM overhead is usually proportional to the number of memory accesses in transactions and many STAMP benchmarks use large transactions. Interestingly, `yada` and `bayes`, under the lock implementation, experience an improvement in the single-thread performance because the compiler outlines OpenMP regions into functions. Function outlining sometimes can result in reduced register pressure and better code generation. The single-thread speedup of `yada` running under the LR mode is the result of a similar code outlining effect.

The rest of this section examines the single-thread overhead of BG/Q TM in detail. There are three causes to the single-thread overhead in BG/Q TM due to increase either of L1 cache misses or of instruction path lengths.

7.1 Cache Performance Penalty

The loss of L1 cache locality due to L1 cache flush or bypass is one of the most dominant source of the BG/Q TM overhead. Table 2 shows the number of L1 cache misses per 100 instructions in both running modes of BG/Q TM relative to that of the sequential baseline.

When running a large transaction in the SR mode, the locality loss is especially severe because there is significant locality within a large transaction. When the L1 is bypassed this locality of access does not benefit from L1's lower latency. For instance, `yada`, under the SR mode, suffers from 20 times as many L1 misses as the sequential version does (Table 2), which in turn causes a three-fold single-thread slowdown (Fig. 10). The L2 cache and L1P buffer load latencies are 13 and 5 times higher than the L1 load latency, respectively.

For not-so-small transactions, the LR mode preserves more locality than the SR mode. However, there are still non-trivial increases in L1 misses due to the flush of the L1 cache at the start of a transaction.

7.2 Capacity Overflow

It is possible to have capacity overflow during a single-thread execution. Among all the benchmarks evaluated, only `bayes` and `labyrinth` experience capacity overflow in a single-thread execution (Figs. 6 and 5).

Of the two, `labyrinth` incurs little single-thread overhead. This is because capacity overflow happens in consecutive transactions, in which case, the TM runtime detects a high serialization ratio and is able to retry transactions in the irrevocable mode immediately with few retries.

On the other hand, `bayes` suffers a significant single-thread overhead because capacity overflow is sporadically triggered on 3% of transactions, leading to a low serialization ratio. As a result, each aborted transaction is retried 10 times before being executed in the irrevocable mode. These retries cause more than 2-fold increases in the instruction path length (Table 2).

There is one more benchmark, `hmmpfam`, that has non-zero serialization ratio at a single-thread. But that is caused by JMV rather than by capacity overflow.

7.3 Transaction Entry and Exit Overhead

When starting or committing a transaction, the TM runtime performs the following tasks: 1) register check pointing, 2) applying for a spec-ID, 3) writing to the memory-mapped I/O to start or commit a transaction, 4) toggling kernel sandboxing via system calls, and 5) other runtime bookkeeping. These operations also contribute to the single-thread TM overhead.

To quantify this overhead, we measure the time spent in a transaction that implements a single atomic update operation. The overhead is in the order of hundreds of cycles for both BG/Q TM and TinySTM, but less in BG/Q TM. Specifically, the overhead for BG/Q TM is 44% of that of TinySTM for the SR mode, and 76% of that of TinySTM for the LR mode. The LR mode incurs a higher overhead than the SR mode because accesses to internal TM run-time data structures before and after transactional execution also suffer from L1 misses due to the L1 cache invalidation.

The overhead of entering and exiting transactions is most pronounced in programs with small and frequent transactions. As shown in Table 2, the instruction path length increase in `utilitymine`, `ssca2`, and `kmeans` is the result of this overhead.

8 Scalability

This section examines the scalability of different parallel implementations of the benchmarks using BG/Q TM, locks, and TinySTM. The speedups of these parallel implementations over the sequential baseline are shown in Fig. 4.

The relative critical section size is a good predictor of the scalability of certain parallel implementations. Therefore, the rest of the section uses the following classification of the benchmarks:

- *Loosely synchronized.* Applications whose relative critical section sizes are less than 1/64. This category includes all the RMS-TM benchmarks except for `hmmpfam` and `utilitymine`. `fluidanimate` performs no synchronization at 1-thread and hence its critical section size is shown as NA.
- *Moderately synchronized.* Applications whose relative critical section sizes are less than 1/3. This category includes `kmeans`, `ssca2`, and `hmmpfam`.
- *Heavily synchronized.* Applications whose relative critical section sizes are more than 1/3. This category includes all the STAMP benchmarks except for `ssca2` and `kmeans`.

8.1 Locks

The relative critical section size is a good indicator of the scalability of the lock implementation of a parallel code. For instance, loosely synchronized applications are expected

TABLE 3
Percentage of Irrevocable and Aborted Transactions in BG/Q TM Execution

Benchmark	Serialization ratio (# threads)							Abort ratio (# threads)						
	1	2	4	8	16	32	64	1	2	4	8	16	32	64
bayes	2 %	16 %	21 %	24 %	23 %	18 %	18 %	25 %	50 %	59 %	67 %	71 %	76 %	76 %
genome	0 %	0 %	0 %	0 %	1 %	1 %	0 %	0 %	2 %	3 %	8 %	27 %	23 %	16 %
intruder	0 %	0 %	0 %	7 %	19 %	20 %	20 %	0 %	4 %	18 %	57 %	64 %	66 %	66 %
kmeans_low	0 %	0 %	0 %	0 %	1 %	5 %	9 %	0 %	0 %	0 %	4 %	49 %	62 %	64 %
kmeans_high	0 %	0 %	0 %	1 %	6 %	18 %	20 %	0 %	0 %	5 %	38 %	64 %	66 %	67 %
labyrinth	24 %	22 %	14 %	25 %	27 %	27 %	23 %	50 %	55 %	71 %	67 %	69 %	70 %	74 %
ssca2	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %
vacation_low	0 %	0 %	0 %	0 %	7 %	13 %	0 %	0 %	15 %	13 %	21 %	57 %	57 %	21 %
vacation_high	0 %	0 %	0 %	0 %	5 %	13 %	0 %	0 %	12 %	11 %	18 %	49 %	53 %	23 %
yada	0 %	3 %	3 %	5 %	17 %	19 %	19 %	0 %	40 %	45 %	51 %	58 %	62 %	62 %
apriori	0 %	3 %	5 %	13 %	NA %	NA %	NA %	0 %	9 %	32 %	51 %	NA %	NA %	NA %
hmmcalibrate	0 %	1 %	3 %	6 %	10 %	17 %	20 %	0 %	6 %	16 %	31 %	38 %	50 %	67 %
hmmpfam	4 %	9 %	16 %	24 %	30 %	27 %	21 %	4 %	27 %	44 %	57 %	65 %	69 %	74 %
hmmsearch	0 %	1 %	1 %	3 %	7 %	15 %	23 %	0 %	8 %	13 %	28 %	37 %	43 %	53 %
fluidanimate	NA %	0 %	0 %	0 %	0 %	0 %	0 %	NA %	0 %	0 %	0 %	0 %	0 %	0 %
scalparc	0 %	0 %	2 %	11 %	25 %	29 %	29 %	0 %	5 %	40 %	51 %	57 %	60 %	60 %
utilitymine	0 %	0 %	0 %	0 %	NA %	NA %	NA %	0 %	0 %	0 %	1 %	NA %	NA %	NA %

to scale well using locks. As shown in Fig. 4, all applications in the loosely synchronized category scale to 64 threads except for `scalparc` that scales up to 32 threads.

On the other hand, heavily synchronized applications exhibit no scalability using locks except for `intruder` because all but `intruder` have a relative critical section size of close to 100%. In contrast, `intruder` has a relative critical section size of 66% and is able to scale beyond eight threads but only reaches a speedup of 2.5 times.

Moderately synchronized applications start with good scalability until reaching a plateau. The thread count at the point where the plateau is reached corresponds roughly to the inverse of the relative critical section size of the application. One exception is `utilitymine`, which scales up to 8 threads despite having a relative critical section size of 35%.

8.2 BG/Q TM

The classification according to the amount of synchronized execution provides a model to predict where BG/Q TM is likely to show benefits over conventional locking. For instance, BG/Q TM is unlikely to outperform locks for loosely synchronized benchmarks, but may improve over locks for moderately or heavily synchronized benchmarks, provided that BG/Q TM does not suffer from other serialization bottlenecks.

To quantify the amount of serialization in a TM execution, Table 3 shows the serialization ratio and the abort ratio (defined in Section 5) computed from statistics collected by the TM runtime. These ratios are determined by the amount of optimistic concurrency inherent in the program, hardware conflict detection granularity, and the retry adaptation of the TM runtime. Sometimes the abort ratio decreases with higher number of threads—as shown in Table 3 for `labyrinth` and `vacation`—because aborts caused by conflicts are highly dependent on the start and commit timing for the various transactions. Therefore, changing the number of threads may change this ratio in unexpected ways. Our runtime adaptation scheme usually limits the number of retries for failed transactions. Thus, its effects are generally to lower the abort ratio at the expense of increasing the serialization ratio. There are two groups of applications that scale well under BGQ TM:

- **Good scaling due to loose synchronization:** `apriori`, `hmmcalibrate`, and `hmmsearch` scale fine under lock and TM implementations. All three are loosely synchronized, serialization of critical sections is not a scalability bottleneck and transaction retries incur negligible overheads.

  Having a certain amount of transaction aborts, or irrevocable execution, does not necessarily limit scalability. For instance, `hmmcalibrate` and `hmmsearch` exhibit significant serialization ratio (up to 23%) and abort ratio (up to 67%) at high thread counts.
- **Good scaling via effective HTM:** `genome`, `vacation`, `scalparc`, and `utilitymine` exhibit a good scalability and a low serialization ratio.[7] Both `genome` and `vacation` are heavily synchronized leading the lock implementation to completely serialize and the TM implementation scales much better. Performance boosts beyond 16 threads come from SMT threads multiplexing on the processor pipeline and from hiding in-order processor stalls.

The rest of the applications all exhibit various scaling bottlenecks that prevent them from scaling to high thread counts under BG/Q TM:

- **Spec-ID bottleneck.** Despite zero abort and serialization ratios, `ssca2` and `fluidanimate` scale only up to 4 and 16 threads, respectively. Both benchmarks use short and frequent transactions leading the system to quickly exhaust spec-IDs. In this case, the start of a new transaction is blocked until after a spec-ID is recycled. BG/Q TM has

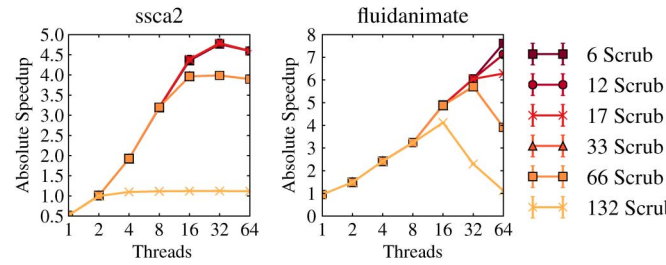

Fig. 11. Effect of varying scrub intervals for `ssca2` and `fluidanimate`.

7. `utilitymine` and `apriori` do not have inputs for 16-64 threads and hence NA's are shown in Table 3.

TABLE 4
Basic Features of Real HTM Implementations

HTM	BG/Q	Rock	Azul	zEC12	Haswell	POWER
Buffer capacity	20MB	32 lines	16 KB	unknown	unknown	unknown
Speculative buffer	L2	store queue	L1	Gathering Store Cache	unknown	unknown
Register save/restore	no	yes	no	yes	yes	yes
Unsupported ISA	none	yes	none	none (except in constrained transactions)	yes	yes
Conflict detection	8-64B	n/a	32B	265B	unknown	unknown
User-level abort	no	yes	n/a	yes	yes	yes

only 128 spec-IDs and they are recycled periodically based on a pre-determined interval called the scrub interval. Fig. 11 shows a sensitivity study on the impact of the scrub interval on the performance of `ssca2` and `fluidanimate`. With the default scrub interval of 132 cycles, `ssca2` and `fluidanimate` run out of spec-IDs beyond 2 and 16 threads respectively. As shown in Fig. 11, the scalability of both benchmarks improves significantly with a much lower scrub interval.

- **Contention bottleneck.** For `yada`, `bayes`, `intruder`, `kmeans`, and `hmmpfam`, high serialization ratios at higher thread counts are the main bottleneck for scalability. Since all 5 benchmarks are moderately or heavily synchronized, the serialization of critical sections limits the scalability of the applications. The high variability in the execution time of `bayes` is because the termination condition of `bayes` is sensitive to the commit order of the transactions.
- **Capacity bottleneck.** The main transactions of `labyrinth` are always executed in the irrevocable mode due to capacity overflow (see Section 7.2). As a result, the performance of BG/Q TM is similar to that of locks and exhibits no scalability.

8.3 TinySTM

This section compares the scalability of TinySTM and BG/Q TM on the STAMP benchmarks. The strength of BG/Q TM is best demonstrated on `genome`, `vacation`, and `kmeans` where BG/Q TM has both a steeper and a longer ascending curve than TinySTM does. For these benchmarks, BG/Q TM does not suffer from any HTM-specific scaling bottlenecks and benefits from a much lower single-thread overhead. In addition, the lower overhead of BG/Q TM likely reduces the window of overlap among concurrent transactions which in turn may reduce transactional conflicts.

For the rest of benchmarks, BG/Q TM incurs a much lower single-thread overhead, but TinySTM exhibits a better relative scalability, that is, scalability with respect to a single-thread TM execution. The better relative scalability of TinySTM is due to its finer conflict detection granularity (word-level) and the fact that it rarely suffers from capacity overflow and does not have spec-ID issues.

The good scaling of `labyrinth` and `bayes` on TinySTM is the result of a STM programming style that relies heavily on manual instrumentation. Table 5 shows the average read- and write-set size per transaction using TinySTM. On the only two benchmarks with capacity overflow during a single-thread BG/Q TM execution, the STM executions incur no single-thread overhead because instrumented state is aggressively reduced to a tiny fraction of the actual footprint of the transactions.

9 Related Work

Despite many HTM proposals in the literature for hardware support for transactional memory [3], [14], [19], [22], [25], only recently real HTM implementations became available. Besides the earlier Rock processor [10] and Vega Azul system [8], now we have Intel Haswell [17], the IBM zEC12 enterprise server [16], and IBM TM support for the POWER architecture [4]. While all are best-effort HTMs, their design points differ drastically. Table 4 compares the key characteristics of these systems in detail.

Both Rock HTM and Vega from Azul have small speculative buffers, compared to BG/Q's 20Mbytes of speculative state. 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 instructions that use register windows [10]. In contrast, in BG/Q TM, the entire instruction set architecture is supported within a transaction and the compiler saves/restores registers.

The method used to build a software system to offer guarantee of forward progress on top of a best-effort HTM could be an elegant solution to the requirement that TM programmers provide an alternative code sequence for transaction rollbacks in Intel's Transactional Synchronization Extensions (TSX) [15], and could thus unburden the TM programmer from the need to reason about hardware limitations [17].

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. A closely related study of HTM on BG/Q corroborates our findings [24].

The implementation of HTM in the IBM zEC12 enterprise server uses the L1 and a modified MESI cache protocol to store speculative state [16]. In this machine both L1 and L2 use a write-through policy, thus the complexity of tracking dirty

TABLE 5
Average Read- and Write-Set Size (in Words) of STAMP Using TinySTM (1 Thread)

Benchmark	Read	Write
bayes	26.8	2.1
genome	36.0	0.9
intruder	23.3	1.6
kmeans_low	4.0	13.0
kmeans_high	4.0	13.0
labyrinth	116.3	177.0
ssca2	1.0	2.0
vacation_low	280.6	5.2
vacation_high	389.2	7.7
yada	42.2	10.8

lines does not exist in this machine. Contrary to the BG/Q design, the System z HTM support includes the implementation of transaction-specific instructions. That system also implements more extensive support for the testing of HTM support and for the debugging of TM code.

10 Conclusion

This detailed performance study of one of the first commercially available HTM systems has some surprising findings. The reduced single-thread overhead in comparison with STM implementations 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 the long-running mode in BG/Q, is critical to achieve performance. The end-to-end solution presented here is a programming model that supports the entire ISA and thus delivers the simplicity promised by TMs.

Trademarks

IBM, AIX, and Blue Gene are trademarks or registered trademarks of International Business Machines Corporation. UNIX is a registered trademark of The Open Group. Intel is a registered trademark of Intel Corporation. Linux is a trademark of Linus Torvalds Other company, product, and service names may be trademarks or service marks of others.

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 B554331. This is also supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by the IBM Toronto Centre for Advanced Studies (CAS).

References

[1] D. A. Bader and K. Madduri, "Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors," in *Proc. Int. Conf. High Perform. Comput. (HiPC)*, Dec. 2005, pp. 465–476.

[2] 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 *Proc. Int. Conf. Comput. Architecture*, 2007, pp. 81–91.

[3] C. C. S. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie, "Unbounded transactional memory," in *Proc. High-Perform. Comput. Architecture*, Feb. 2005, pp. 316–327.

[4] H. W. Cain, B. Frey, D. Williams, M. M. Michael, C. May, and H. Le, "Robust architectural support for transactional memory in the power architecture," in *Proc. Int. Conf. Comput. Architecture*, 2013, pp. 225–236.

[5] 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? "*Commun. ACM*, vol. 51, no. 11, pp. 40–46, Nov. 2008.

[6] S. Chaudhry, 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 *Proc. Int. Conf. Comput. Architecture*, 2009, pp 484–495.

[7] 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 *Proc. Int. Symp. Microarchitecture (MICRO)*, Dec. 2010, pp. 39–50.

[8] C. Click, "Azul's experiences with hardware transactional memory," in *Proc. HP Labs' Bay Area Workshop Trans. Memory*, 2009.

[9] L. Dalessandro, M. F. Spear, and M. L. Scott, "Norec: Streamlining STM by abolishing ownership records," in *Proc. Principles Practice Parallel Program.*, Jan. 2010, pp. 67–78.

[10] D. Dice, Y. Lev, M. Moir, and D. Nussbaum "Early experience with a commercial hardware transactional memory implementation," in *Proc. Architectural Support Program. Languages Oper. Syst. (ASPLOS)*, Mar. 2009, pp. 157–168.

[11] P. Felber, C. Fetzer, P. Marlier, and T. Riegel, "Time-Based software transactional memory," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 12, pp. 1793–1807, Dec. 2010.

[12] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *Proc. Principles Practice Parallel Program.*, Feb. 2008, pp. 237–246.

[13] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, et al. "The IBM blue gene/Q compute chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, Mar./Apr. 2012.

[14] M. Herlihy and J. E. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proc. Int. Conf. Comput. Architecture*, May 1993, pp. 289–300.

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

[16] C. Jacobi, T. Slegel, and D. Greiner, "Transactional memory architecture and implementation for IBM system z," in *Proc. Int. Symp. Microarchitecture (MICRO)*, Dec. 2012, pp. 25–36.

[17] D. Kanter, (2012, Sep.). "Analysis of Haswell's Transactional Memory" [Online]. Available: http://www.realworldtech.com/page.cfm?ArticleID=RWT021512050738&p=1, Real World Technologies.

[18] G. Kestor, V. Karakostas, O. Unsal, A. Cristal, I. Hur, and M. Valero, "RMS-TM: A comprehensive benchmark suite for transactional memory systems," in *Proc. Int. Conf. Perform. Eng. (ICPE)*, Mar. 2011, pp. 335–346.

[19] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun, "Architectural semantics for practical transactional memory," in *Proc. Int. Conf. Comput. Architecture*, Jun. 2006, pp. 53–65.

[20] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *Proc. Int. Symp. Workload Characterization (IISWC)*, Sep. 2008, pp. 35–46.

[21] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun, "An effective hybrid transactional memory system with strong isolation guarantees," in *Proc. Int. Conf. Comput. Architecture*, 2007, pp. 69–80.

[22] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "Logtm: Log-Based transactional memory," in *Proc. High-Perform. Comput. Architecture*, Feb. 2006, pp. 258–269.

[23] 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 *Proc. Principles Practice Parallel Program.*, Jan. 2006, pp. 187–197.

[24] M. Schindewolf, B. Bihari, J. Gyllenhaal, M. Schulz, A. Wang, and W. Karl, "What scientific applications can benefit from hardware transactional memory?" in *Proc. High Perform. Comput. Network. Storage Anal. (HPCNSA)*, Nov. 2012, pp. 1–11.

[25] A. Shriraman, S. Dwarkadas, and M. L. Scott, "Flexible decoupled transactional memory support," in *Proc. Int. Conf. Comput. Architecture*, Jun. 2008, pp. 139–150.

[26] M. F. Spear, M. M. Michael, and C. von Praun, "RingSTM: Scalable transactions with a single atomic instruction," in *Proc. ACM Symp. Parallelism Algorithms Architectures (SPAA)*, Jun. 2008, pp. 275–284.

[27] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation." in *Proc. Int. Conf. Comput. Architecture*, Jun. 2000, pp. 1–12.

[28] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek, "Multiple reservations and the Oklahoma update," *IEEE Parallel Distrib. Technol. (PDT)*, vol. 1, no. 4, pp. 58–71, Nov. 1993.

[29] Transactional Memory Specification Drafting Group, Draft specification of transactional language constructs for C++ (version 1.1), 2012.

Amy Wang received the bachelor degree in applied science from the Engineering Science Program in 1999 and the master degree in applied science in computer engineering in 2001, both from the University of Toronto, Ontario, Canada. She is currently a member of the Hardware Acceleration Lab and formerly a member of the XL compiler back-end team at the IBM Canada Lab, Toronto. In 2002, she joined the IBM Toronto Lab, contributing her skills to the development of various compiler back-end optimizations, such as auto-SIMD-ization for the Blue Gene/L, Blue Gene/P, and Blue Gene/Q, cell broadband engine architecture, and VMX/VSX hardware. She developed the XLs SMP runtime support for utilizing the transactional memory and speculative execution hardware features on Blue Gene/Q.

Matthew Gaudet received the bachelors' degree in computing science from the University of Alberta, Edmonton, Canada, in 2012, where he has continued on to pursue the master's degree. He has industrial experience from working with IBM on a number of projects, and has published research in the area of data mining. His research interests include optimizing compilers, runtime systems, and transactional memory.

Peng Wu received the PhD degree in computer science from the University of Illinois, Urbana-Champaign. She joined IBM T.J. Watson Research Center as a research staff member in 2001. Her work has expanded across all layers of the system stack. Her current research interests include compilation and runtime systems for emerging workloads and software exploitation of new hardware features. She held more than 20 patents, and has co-authored more than 30 papers including a best paper award in PACT 2012. She is an adjunct assistant professor in the Department of Computer Science University of Illinois since 2012.

Martin Ohmacht received the PhD degree in electrical engineering from the University of Hannover, Germany, in 2001. He is currently a research staff member at the IBM T.J. Watson Research Center, Yorktown Heights. He has worked on memory subsystem architecture and implementation for all generations of the Blue Gene super computer project. His research interests include computer architecture, design and verification of multiprocessor systems, and compiler optimizations.

José Nelson Amaral received the PhD degree from the University of Texas at Austin, the MSc degree from the Instituto Tecnológico de Aeronáutica, Brazil. He is a computing science professor at the University of Alberta, Edmonton, Canada. His research focuses on compiler design and implementation, programming languages, high-performance computing, and related areas. He is a former associate editor of the *IEEE Transactions on Computers*, He has also served in many program committees. He is a senior member of the ACM.

Christopher Barton received the MSc and PhD degrees in computing science from the University of Alberta, Edmonton, Canada. He has worked with the XL Compiler team at IBM's Toronto Lab since 2008. He is currently the technical lead for the XL C/C++, and XL Fortran Compilers on AIX and Linux on Power systems.

Raul Silvera received the MSc degree the School of Computer Science of McGill University, Montréal, Canada. He is a senior technical staff member (STSM) at the IBM Canada Lab, Toronto. He joined IBM in 1997 and has been focused on development of compilation technology for the IBM Power and System Z platforms, including code analysis, optimization, parallelization, and code generation for C, C++, Fortran, and other static languages.

Maged M. Michael received the PhD degree in computer science from the University of Rochester, New York, in 1997. Since then, he has been a research staff member at the IBM Thomas J. Watson Research Center,Yorktown Heights, New York. His research interests include concurrent algorithms, nonblocking synchronization, transactional memory, multicore systems, and concurrent memory management. He is an ACM distinguished scientist and a member of the Connecticut Academy of Science and Engineering.