Serialization Management Driven Performance in Best-Effort Hardware Transactional Memory Systems

by

Matthew Gaudet

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science
University of Alberta

Abstract

Serialization Management is the Best-Effort Hardware Transactional Memory (BE-HTM) counterpart to Software Transactional Memory (STM) Contention Management. A serialization manager uses non-speculative serialization to provide a forward-progress guarantee while simultaneously attempting to provide high application performance. Historically, non-speculative serialization management has been done through a simple policy of allowing a fixed number of retries. This thesis investigates the proposition that application performance can be improved through better Serialization Management.

This thesis explores seven serialization managers and their tuning parameters on Blue Gene/Q's BE-HTM system using the Stanford Transactional Applications for Multi-Processing (STAMP) and the Recognition, Mining and Synthesis (RMS-TM) benchmark suites. It presents the first large-scale investigation of Serialization Management for BE-HTM in the literature. This investigation experiments with a large number of values for each tuning parameter on multiple platforms. The main finding is that program performance can be improved by changing the serialization manager. However, performance is actually dominated by the tuning of parameters for each manager and this tuning depends on the benchmark, the thread count, and the platform.

Preface

Parts of Chapter 4 have been previously submitted to the 2013 ACM Student Research Competition Grand Finals, titled *Transactional Event Profiling in a Best-Effort Hardware Transactional Memory System*.

Appendix A is a collaborative work, with authorship shared among Amy Wang, myself, Peng Wu, Martin Ohmacht, José Nelson Amaral, Christopher Barton, Raul Silvera, and Maged M. Michael. I ran the experimentation and data analysis for the paper, with the majority of writing having been done by the remaining authors. It has been accepted for publication in the *IEEE Transactions on Computers*, however it has not yet appeared at the time of thesis submission, and so is included as an appendix.

It's like, French is a great idea, but nobody is going to invent French if they're constantly being attacked by bears. Do you see? SYSTEMS HACKERS SOLVE THE BEAR MENACE. Only through the constant vigilance of my people do you get the freedom to think about croissants and subtle puns involving the true father of Louis XIV.

— James Mickens, *The Night Watch*

Acknowledgements

First and foremost, I must thank my parents. They raised me well, and helped guide me just enough to let me find my path.

Andrea, my long-time partner, has been a wonderful support throughout this process.

My supervisor José Nelson Amaral has guided me continuously for more than six years now, ever since he took me on as a summer student – a program that set me on the path I have followed until today. He has been an excellent mentor, and I do not believe I could have achieved a fraction of what I have without him.

Thanks to Peng Wu, for her excellent guidance and insights, and to Amy Wang for laying the ground work upon which this thesis was built.

I would also like to thank a number of others who each helped me in ways large and small to get through my Masters program: Kit Barton, Martin Ohmacht, Maged Micheal, Wang Chen, Marcel Mitran, Jerry Zheng, Ian Gartley, Alan Li, Joran Siu, Yan Luo, Yi Zheng, Arthur Zimek, Ricardo Campello, Jörg Sander and Manon Gaudet.

IBM must be thanked for being generous in their support. Hundreds of thousands of experiments were performed over the course of this thesis, and would have been impossible without them.

This thesis has been funded in part by an Alberta Innovates Graduate Student Scholarship from Alberta Technology Futures as well as a Canada Graduate Scholarship from the Natural Sciences and Engineering Research Council of Canada.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

For almost five decades computer architecture has followed a trend dubbed Moore's Law, under which the number of transistors in a computer processor has doubled roughly every two years [54]. For much of the lifetime of Moore's Law, gains in transistor counts were accompanied by increases to processor clock speeds leading to a long period in which software would be made faster purely though hardware improvements. In the last decade, however, frequency scaling slowed dramatically due to concerns about power usage and heat dissipation [12, 60]. While Moore's law has continued to dramatically increase the transistor count in microprocessors, the new transistors are being funnelled into multiple cores as opposed to the improvement of performance of a single monolithic core.

The transition from frequency-scaling to 'core-scaling' has necessitated that software begin to change — and become parallel — in order to exploit new generations of processors.

Parallel programming, however, has a reputation for being difficult. Paul E. McKenny divides the tasks a programmer faces when programming in parallel into four categories [51]:

1. **Work Partitioning:** This is the task of splitting the algorithm and its inputs into pieces that can be run in parallel across threads, processes or computers. This thesis discusses parallel *threads*, which are streams of execution that happen concurrently in the same memory space.

2. **Resource Partitioning & Replication:** This is the task of ensuring that the

required resources are partitioned or replicated for the parallel tasks to remove the requirement of access control.

3. **Interacting With Hardware** refers to the necessary understanding of the underlying hardware and it's limitations when programming in parallel. Depending on the model of parallel programming, this interaction ranges from a very small to very large fraction of the parallel programming task.

4. **Parallel Access Control**, which is required to avoid **race conditions** on shared resources. A race condition is an uncontrolled interleaving of parallel access to a shared resource, which is considered an error in most domains because the result being computed will vary on seemingly random timing fluctuations that change the interleaving of accesses occurring between threads[1].

For the purposes of this thesis *shared-memory* machines are assumed. A shared-memory machine is a type of parallel architecture where all the processors working on a problem work in a shared address space, as opposed to distributed parallel computation where processors working in parallel do not share memory and, instead, must communicate explicitly through messages.

In a shared-memory machine, a *race-condition* interleaving typically involves **instructions accessing memory** across two or more threads in the machine. These race conditions are called *data races* and are discussed further in Section 2.1. Higher-level programming abstractions may also be subject to race-conditions, and thus semantically require similar parallel access control, however this thesis focuses on memory-level data races.

In order to avoid data races some form of control must be applied to shared resources. There are a number of synchronization primitives that can be used, however a *lock* is the most common [51]. Lock variables allow a single thread to declare ownership of a resource. While a lock is held, other threads attempting to acquire the same lock should either wait for the lock to become unlocked or proceed down an alternate path of execution that does not require access to the resource.

[1]Algorithms can be created to be race-tolerant so that either races do not affect results or the results are acceptable even in the face of races. However, it is difficult to ensure that such algorithms are correct.

Lock-based programming has been successfully used for parallel programming, however, the paradigm does have some challenges:

- From the perspective of correctness, locks can be misused in such a way that progress can no longer be made. This failure condition is called a *deadlock*, and is caused by a cycle of lock dependencies created by two or more threads. In the simplest version of this situation two threads each hold a lock that the other thread requires for progress. Another related failure condition — that also applies to a number of other programming systems — is called *livelock*, which describes a situation where threads become looped in a cycle of actions without making progress.

- From the perspective of productivity, locks require large amounts of programmer effort to achieve maximum performance.

  A pathology in lock-based programming is *lock contention*. This is when multiple threads require the same lock simultaneously to make progress. The other threads must simply wait to acquire the lock in order to proceed, because, by definition, only one thread can hold the lock. Lock contention can dramatically affect program performance if the contended lock is on the critical path. While lock contention can be inherent in an algorithm or data-structure, sometimes lock-contention can be the result of protecting too much data with the same lock. If multiple threads are waiting for the lock, but all the threads are to access different data, then the lock contention is spurious.

  Figure 1.1 illustrates the problem with a hash-table that has a single lock. The accesses to be performed by fifteen threads are indicated by arrows pointing to the hashtable buckets. Of the fifteen accesses, all but three are to different buckets, and thus such accesses do not require synchronization. The bottom left part of the figure shows the resulting lock contention: all the threads wait to acquire the single lock. This is true even for threads that have independent accesses. The bottom right of Figure 1.1 shows how performance can be dramatically improved by locking only where conflicts can occur, as only three of the fifteen locking operations are *required*.

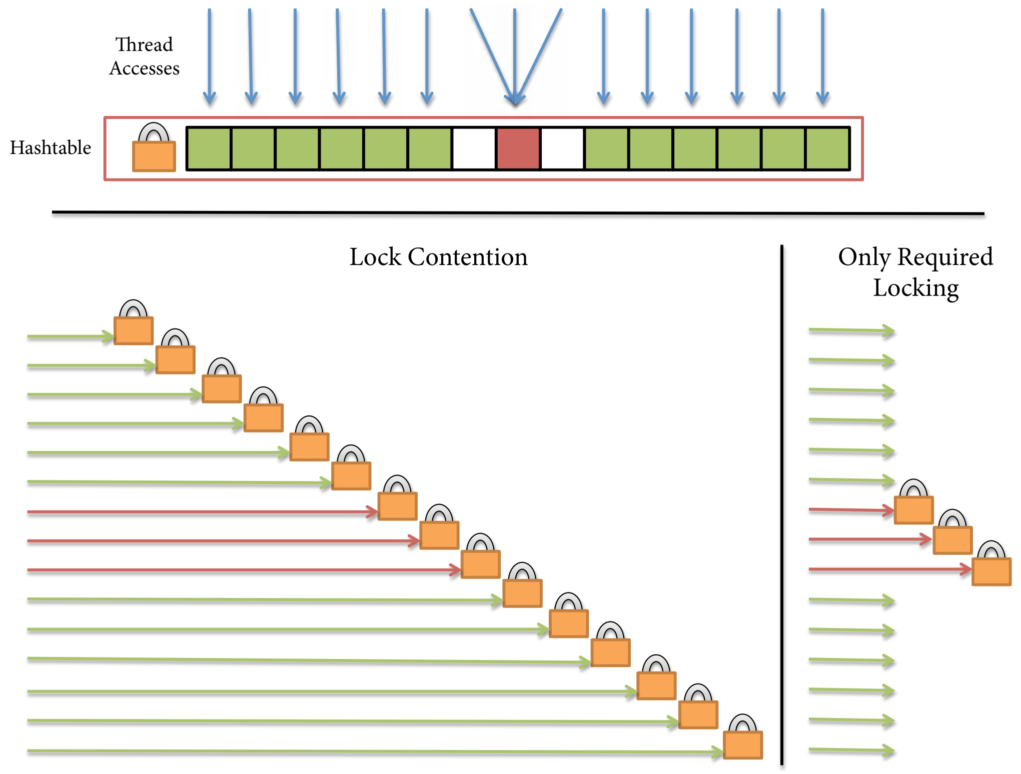

Figure 1.1: Lock Contention visualized on a hashtable. The accesses in green would produce spurious locking, but the accesses in red require some synchronization. White buckets are not accessed by any thread.

To work around spurious lock contention in a program, typically the single contended lock must be broken into multiple locks, each protecting a set of hash table buckets, in order to still guarantee safety. For example, if the contended lock is guarding access to a large array or hashtable, then *lock striping* could be applied to break the array or table into chunks, each guarded by its own lock [41]. Splitting locks must be carefully considered, however, because multiple locks combined with poor lock-discipline can introduce bugs that did not exist in the single-lock version of the algorithm.

- From the perspective of performance, locks add computation and memory accesses to memory locations that are not needed by the algorithm. These accesses are a waste of resources and time in the absence of contention on the data.

The difficulties imposed by deadlock and contention raise the burden of parallelization on the programmer. However, reducing the burden of parallelization is highly desirable because parallelization appears to be the main driver of future performance.

1.1 Transactional Memory

One proposed approach to parallelization with a reduced programming burden is called **Transactional Memory** (TM) [40].

Transactional Memory introduces a high-level construct called a *transaction*. A transaction is a region of code whose execution semantically cannot overlap with another transaction. A transaction must run in **isolation** from other transactions, must not be able to see partial results computed in another transaction, and must *commit* its results indivisibly (*atomically*) so that partial state cannot be seen.

Transactions are a higher-level construction in comparison with locks. Locks are akin to imperative programming, in that the user instructs the system **how** to achieve safe interleaving. In contrast TM programming is akin to declarative programming, where the user tells the system **what** is required for safe execution, leaving the actual mechanics to the system.

A side effect of this higher level of abstraction is that TM systems can provide improved performance by *optimistically* allowing execution to overlap. The TM system then can monitor the reads and writes performed by parallel executing transactions. Whenever a transaction attempts to execute an operation that would violate the isolation guaranteed offered by TM, one or more transactions can be *aborted*, and rolled-back to transaction start. A rollback resets the state of execution to what it was at the beginning of the transaction. Once the transaction has been rolled back, it may retry in hopes of succeeding on the next attempt, or it may switch to an alternate method of synchronization. This model is similar to database transactions, hence the name Transactional Memory.

First proposed in 1993 by Herlihy and Moss as a hardware modification, the idea has gained much popularity in the research community, which has been active in the area for the last 20 years [40]. In order to allow exploitation of TM for systems that already use locks, Transactional Lock Elision (TLE) can be used to replace lock pairs with transactions.

When discussing transactions, it is important sometimes to distinguish between the source code level transaction and an instance of a transaction at execution time. In this thesis I will refer to the the former, a region of code in the program text, as a *static transaction*. During program execution, a static transaction will likely be executed many times (*i.e.* being in a loop or function call), generating a unit of computation that must be executed in isolation and committed atomically. In this thesis I call this a *dynamic transaction*.

1.1.1 Software Transactional Memory

One of the popular pieces of TM research in the last two decades has been the exploration of Software Transactional Memory (STM) [66]. The promise of STM is to provide a high-level transactional construct on existing machines with no hardware changes. In STM systems the detection and rollback of conflicts is achieved through the use of a software library and the instrumentation of reads and writes in the program. This instrumentation can be done by the user, or automatically by an STM-aware compiler.

1.1.2 Best-Effort Hardware Transactional Memory

One of the most recent developments[2] in the area of Transactional Memory has been the appearance of commercially available, user-accessible, Hardware Transactional Memory (HTM) systems.

While Transactional Memory was originally specified as a hardware feature, the cost of producing hardware has meant that TM research has been done through either Software Transactional Memory or through system simulators for HTM.

In the last two years three HTM systems have shipped: IBM's Blue Gene/Q, IBM zEC12 and Intel's Haswell-based processors [44, 45, 74]. In a short while, IBM will be shipping a third platform with TM support, the POWER8 processors [14]. One characteristic that these three systems share is that they are all 'Best-Effort' HTM systems (BE-HTM). Best-effort HTM means that they do not guarantee that any transaction may be completed speculatively [25]. This is a practical side effect of these being early HTM designs adapted to existing architectures. These machines all can store only limited amounts of speculative state, and will abort a transaction that exceeds that amount of speculative state. The architectures also restrict the state changes that may occur within a transaction to those that can be easily buffered, typically in one of the data caches or buffers.

1.2 Performance in TM systems

Transactional-Memory performance often depends on perspective. While creating a new transactional program, it would be unfair to not account for the productivity gains brought by the simpler synchronization model.

However, if one is considering converting an existing lock-based program to transactions as a way of improving performance, then comparing against the performance of the lock-based version is important. This thesis does not focus on comparing against different synchronization mechanisms so much as it focuses on performance among different transactional variants, and therefore the performance

[2] Azul Systems shipped a version of their Vega processor with HTM several years ago, however it was deployed completely transparently to the user, and has not been widely evaluated in the research community.

study focuses on *absolute speedup*, *i.e.* ratio of wall-clock running time of the parallel implementation to the wall-clock running time of the fastest available sequential implementation.

In general, TM performance is dominated by two elements.

1. TM systems can optimistically run, possibly conflicting, transactions speculatively in parallel. One component of transactional performance is what proportion of transactions that started have successfully commited, as opposed to aborting and rolling back to the starting point.

2. The overhead to set up and execute a transaction speculatively in a TM system is higher than that of a simple locking solution. This overhead comes from sources such as instrumentation in the Software Transactional Memory (STM) case, or from mechanisms required to provide forward progress in BE-HTMs. This overhead varies from system to system. It is relatively high — hundreds of cycles — for STM systems and BG/Q, and quite low — tens of cycles — for Intel Haswell, and zEC12.

A transaction that rarely, or never, aborts is an excellent candidate for TM in a comparison against a lock-based implementation that suffers from lock contention. However, if a TM is used to elide a lock that is never contended, then the TM overhead could turn this lock elision into a net loss.

On the other hand, a transaction that aborts in the majority of its attempts to execute could still be a net-gain, if the alternative is a lock-based version of the program where waiting on the lock dominates the execution time.

In general, it is speculation that is key to the promise of TM's productivity and performance. Speculation is the key to TM's productivity because it allows programmers to declaratively specify the synchronization required, with the system providing the actual safety guarantee. It is key to the performance of TM systems because successful speculations allow computation to proceed in spite of the *appearance* of required synchronization, even though unsuccessful speculation may waste execution cycles.

1.3 Programming Models and Forward Progress

There are two different programming paradigms for STM systems: user-instrumented and compiler-instrumented STM systems. While the former generally has better performance because it allows a skilled user to instrument only the required accesses for safety, the latter provides an excellent programming model. In a compiler-instrumented STM system, the user needs only to indicate to the compiler which regions of code cannot overlap in execution. Instrumentation is done automatically by the compiler, and forward progress is guaranteed by the runtime system.

An important question in STM systems is: *When a conflict between two or more transactions occurs , what transaction(s) should be rolled back?* The answer to this question has important consequences. An inappropriate answer can cause starvation — where a particular thread is unable to make progress; livelock — where the system as a whole is unable to make progress because it is trapped in a cycle of execution states; or simply poor performance. Attempts to answer this question in STM systems came to be known largely as the area of *Contention Management*, which explores policies, called *contention managers*, that decide which in-flight transactions to rollback [42]. Contention Management has been a rich area of research in STM systems.

An STM system with a correctly designed contention manager can always guarantee forward progress. For example, a contention manager that always ensures that the eldest transaction involved in a conflict is not aborted will trivially guarantee forward progress.

Forward progress is a very important property for TM systems because programming without it would be substantially more difficult, requiring the programmer to provide an alternate path that doesn't require a transaction, leading to an increased amount of code and maintenance burden.

1.4 Programming Models and Forward Progress for Best-effort HTMs

When programming for a best-effort HTM system there are different possibilities for programming models that can be provided to programmers. One of the most accessible programming models for programmers is a model that provides a high-level model similar to that of a compiler-instrumented STM.

In this programming model a user is only responsible for specifying the boundaries of a transaction. In order to avoid putting platform-dependent burdens on the programmer, this model allows transaction bodies to contain any code, including *irrevocable actions* or code that generate so much speculative state that the hardware will not be able to store such state.

Irrevocable actions are those that the system cannot rollback or buffer appropriately. For instance, executing a function that prints to a terminal is an irrevocable action: once the text appears on the terminal, the action cannot be undone. Irrevocable operations can take many forms, such as a reference to a memory-mapped I/O address that cannot reasonably be buffered, or system calls that cannot be rolled back.

To understand how useful it is to allow irrevocable actions inside a transaction, consider a program where an irrevocable operation that appears inside a transaction is only rarely executed at runtime: *e.g.* a warning message printed on an exceptional result.

The code is greatly simplified by allowing the irrevocable operation to appear within the transaction region. Allowing irrevocable operations inside a transaction is also a great advantage when legacy code is converted to transactions. An unpleasant alternative would be for the runtime to signal a fatal error upon encountering an irrevocable operation, or for the compiler to require no irrevocable actions statically. Either case would likely require significant programmer's effort to rewrite code to make it transactional.

Similarly the ability to specify transactions that require more speculative state than is available in the hardware allows the creation of TM applications that can be ported to other systems, even though its performance may be affected by the hardware limitations in the new system.

The freedom to specify transactions outside the limits of the hardware in a TM programming model improves programmability but puts the burden of providing forward-progress guarantee on the platform.

1.5 Non-speculative Serialization and the Serialization Manager

In a best-effort HTM system, a contention manager would not suffice to provide a forward-progress guarantee. The nature of a best-effort HTM means that a strategy similar to STM cannot always ensure forward progress because when a transactional execution failure occurs in a best-effort HTM, there is no guarantee that any of the aborted transactions may be able to continue execution. For instance, consider a capacity-limited transaction, *i.e.* a transaction that creates more speculative state than the hardware can store. No matter which conflict-resolution policy the contention manager uses, forward progress will never occur without additional action from the system. Therefore the only failure-proof way to guarantee forward progress is to execute such a transaction non-speculatively.

Non-speculative serialization is a technique that allows a transaction to safely run non-speculatively if the transaction is deemed unable to succeed in a hardware transaction. It is called *serialization* because, to avoid a failure of isolation semantics, there is at most one non-speculative transaction running at any moment. Concurrent speculative transactions may not commit until after the non-speculative transaction completes[3].

Though non-speculative serialization has been described before, there has been very little discussion of how to decide **when** non-speculative serialization is to occur. Most previous publications that discuss non-speculative serialization for forward progress use a simple policy that limits retries to some maximum value [78].

This thesis introduces the concept of a ***serialization manager*** as the BE-HTM counterpart to the STM contention manager. Similar to the specification of contention managers, serialization managers separate the responsibility for forward-progress from the responsibility for correctness. The serialization manager is the policy, along

[3]Non-speculative serialization assumes that all threads accessing shared data structures use transactions. Non-speculative serialization will not match the hardware isolation semantics should there be a thread not using transactions when writing shared-data, as that non-speculative thread is invisible to the TM system and cannot be held-back while the transaction executes. This could be fixed at great cost by serializing the whole system, however the proportional response is to consider this case a programming error.

with any required state, that decides, when a transaction aborts, if the aborted transaction should be retried speculatively or if non-speculative serialization is required.

1.6 Serialization-Manager-Driven Performance

Improved STM Contention Management can improve program performance by making better decisions about what transactions to abort when conflicts occur [42]. This thesis argues that improved Serialization Management may also be able to improve BE-HTM performance because serialization[4] forms a key safety net in the face of high contention. The thesis also argues that better decisions about when to serialize can improve performance while guaranteeing forward-progress.

A key tension in a serialization manager design is the tradeoff between forward progress and speculation. An upper bound on execution time would be desirable to prevent inefficient use of resources and lots of wasted work. However, such upper bound comes at a cost because performance in a BE-HTM system is rooted in speculation. Enforcing the upper bound may eliminate beneficial speculation. If the upper bound is too high the program wastes work. If it is too low the overzealous serialization manager eliminates too much speculative execution.

This thesis presents six new serialization managers — and one that has been the default used in previous explorations of non-speculative serialization — along with the first through exploration of Serialization Management in the literature. The evaluation explores a large number of values for each parameter of each manager on a TM architecture that has two different modes, each having different conflict-detection granularity and access latencies. These two modes are considered for the purposes of this thesis to be two different, though closely related, platforms for TM. This TM implementation was the first commercially available offering of HTM, introduced by IBM, and equipped the IBM BG/Q machine — a machine that was the top-performing computer in the world circa 2013. The thorough evaluation of these serialization managers, along with their tuning on multiple platforms, provides insight into the amount, and type, of control afforded to a serialization manager.

[4] For the remainder of this thesis we will use the terms serialization and non-speculative serialization interchangeably.

The remainder of this thesis is structured as follows:

Chapter 2 provides more background on race conditions, Transactional Memory and the experimental platform for this thesis, Blue Gene/Q.

Chapter 3 discusses the problem further, generating some smaller research questions to help guide the study. Once the research questions are established, the research methodology is described.

Chapter 4 introduces a tool that was designed and built to help explore questions around serialization.

Chapter 5 introduces the serialization managers.

Chapter 6 introduces a pair of modifications that can be made to all serialization managers, and evaluates them on the MaxRetry manager, before recommending they both be adopted for all serialization managers.

Chapter 7 evaluates the new serialization managers across parameter values for those managers that have parameters.

Chapter 8 contains the experimental results, where our research questions are answered.

Chapter 9 discusses related work.

Chapter 10 discusses some limitations to the thesis.

Chapter 11 discusses directions for future work.

Chapter 12 concludes the thesis.

The thesis relies heavily on the contents of *Software Support and Evaluation of Hardware Transactional Memory on Blue Gene/Q* for background. For ease of reference I have included it as Appendix A.

Chapter 2

Background

This chapter provides background on race conditions, Transactional Memory and the Blue Gene/Q's Transactional Memory system to support the reader through the rest of the thesis.

2.1 Race Conditions

A race condition is an uncontrolled interleaving of accesses to a shared resource, where at least one of the accesses modifies the shared resource. In this thesis the interleaving discussed is between instructions accessing memory in a shared-memory machine that has more than one processor. Shared-memory race conditions can be classified as Read-after-Write (RAW), Write-after-Write (WAW) or Write-after-Read (WAR).

Listing 2.1: Race Condition Demonstration

```
int main(int argc, char** argv) {
    int iterations = atoi(argv[1]);
    printf("Running with %d iterations\n",iterations);
    int counter = 0;

#pragma omp parallel for
    for (int i=0; i < iterations; i++) {
        counter = counter+1; // Racy Update
    }

    printf("%s: Counter is %d, expected %d\n",
            counter == iterations ? "PASSED" : " FAILED",
            counter,
            iterations);
}
```

Listing 2.1 shows a simple program that suffers from a data race that may cause updates to the variable `counter` to be lost, leading to a mismatch between the expected and actual value in the output of the program, contained in Listing 2.2.

Listing 2.2: Race Condition Output

```
> g++ -fopenmp race_condition.c -o race
> OMP_NUM_THREADS=1 ./race 90000000
Running with 90000000 iterations
PASSED: Counter is 90000000, expected 90000000

> OMP_NUM_THREADS=2 ./race 90000000
Running with 90000000 iterations
FAILED: Counter is 50095286, expected 90000000
```

To show in more detail how this race affects the values computed by the program, Listing 2.3 contains a pseudo-assembly of the loop body that contains the race condition. Figure 2.1 shows an interleaving of instructions between two threads where the update intended by Thread 1 is lost. An update was lost because both threads loaded, incremented, and stored the same value, a WAW race.

Listing 2.3: Pseudo Assembly of Race Condition loop

```
/* Initialize variables */
    ST      [counter],0
    ST      R0,0 /* R0 will be the loop variable i */
L:
    LD      R1,[counter]
    ADD     R1,1
    ST      [counter],R1
    ADD     R0,1
    CMP     R0,[iterations]
    BEQ     L

    ...
```

Figure 2.1: Snippet of execution showing a data race with lost updates.

(a) Thread 1

```
T1: LD      R1,[counter]
T2:
T3: ADD     R1,1
T4: ST      [counter],R1
T5:
```

(b) Thread 2

```
T1:
T2: LD      R1,[counter]
T3: ADD     R1,1
T4:
T5: ST      [counter],R1
```

2.2 Transactional Memory

Transactional processing has existed for a long time, with one of the earliest widely deployed examples being the SABRE airline reservation system of the early 1960s [9]. Since SABRE the area has gone through much evolution, with large amounts of formalization driven by the development and evolution of relational databases.

Part of the formalization of relational databases was the description of four properties to guarantee consistent processing of transactions: Atomicity, Consistency, Isolation and Durability, known as ACID [34, 35, 37].

Atomicity A transaction is atomic if the changes it makes to storage appear to happen indivisibly: either all the changes must be seen at once, or none of the changes can be seen.

Consistency The system remains consistent after the execution of a transaction. If a transaction would leave the system in an inconsistent state, then the transaction is not allowed to complete. For instance, a transaction that changes a set of records in a database must appropriately change all required records.

Isolation Transactional isolation is the inability of any component of the system to see partial results from an in-progress transaction. Isolation is a key part of proving consistency because, without isolation, invalid results could be produced by using invalid temporary results produced during the execution of a transaction.

Durability Once a transaction commits the results are preserved even in the event of a system crash.

As can be seen from the D in the ACID definitions, historically transaction processing has referred to persistent data, mostly stored in files on hard drives and tapes. The first three properties — Atomicity, Consistency and Isolation — are powerful concepts and were adapted to transient storage and parallel programming.

Though atomic updates for single variables had existed in various forms in computer architecture (such as Compare-and-Swap, Test-and-Set, or Load-linked/Store

Conditional) previously, the Oklahoma Update[1] was different in providing a form of atomic updates to multiple values [73]. In an example of independent discovery, Herlihy and Moss described *Transactional Memory* at almost the same time, differing from the Oklahoma Update mostly in implementation and terminology [40].

Herlihy and Moss' description of Transactional Memory has been more persistent than the Oklahoma update. Herlihy and Moss added six instructions to an architecture's ISA: *Load-transactional* (LT) that loads a memory location and marks the location as transactionally read, *Load-transactional-exclusive* (LTX) that loads a memory location marking it as read and hinting that it will be updated, *Store-transactional* (ST) that writes speculatively to a location, *Commit* (COMMIT) that commits a transaction if all the values written and read are still valid. In addition, the Herlihy and Moss paper also added two instructions that are less important for our discussion here: *Abort* (ABORT) that discards transactional state, and *Validate* (VALIDATE) that tests whether or not the currently executing transaction's read- and write-sets are valid, aborting if not. A transaction is implicitly started at the first transactional access and continues until COMMIT, ABORT or a failed VALIDATE.

Similar to Compare-and-Swap, a transaction can fail if another thread has altered any of the values in the read set or the write set while the transaction is in flight. In Herlihy' s and Moss' paper validation occurs on the execution of a COMMIT instruction that can return TRUE, if the transaction is committed, or FALSE if the transaction commit failed and the speculative state has been discarded. If a transaction fails it is up to the programmer to handle this case, either retrying or choosing an alternate method of synchronization.

Both the Oklahoma Update and Herlihy's Transactional Memory provide Atomicity, Consistency and Isolation[2] to those values loaded and stored transactionally.

Though the advantage of transaction-inspired parallel programming could be seen, both the Oklahoma Update and Transactional Memory were specified as hardware features. Hardware being both complicated and expensive to build meant that research

[1]Referring to the song *All 'Er Nuthing* from the musical *Oklahoma!*

[2]To be precise, the Oklahoma Update does not provide Isolation because the authors describe a situation under which a failed update could lead to variables becoming inaccessible. However, the authors also sketch a fault-tolerance system that can work around this situation and provide isolation.

on Hardware Transactional Memory work was done on hardware simulators until much later.

2.2.1 Software Transactional Memory

However, Transactional Memory was not forgotten. Effort simply moved towards providing the transactional memory paradigm using software, leading to the research area of Software Transactional Memory (STM), which has remained active to this day [66]. The goal of STM systems is to provide the transactional programming model on un-modified hardware by using a runtime system and instrumentation of reads and writes, done either by an STM compiler or by the user. An example of instrumentation is contained in Figure 2.2.

Figure 2.2: An example of STM instrumentation

(a) Before instrumentation

```
stm_transaction {
 A[i] = B[i] + C[i]
}
```

(b) After STM instrumentation

```
stm_begin();
 b_tmp = stm_load(&(b[i]));
 c_tmp = stm_load(&(c[i]));
 stm_store(&(a[i]), b_tmp + c_tmp);
stm_commit();
```

Software Transactional Memory has followed its own evolution, and has introduced a number of concepts and design dimensions. One addition to the TM model was the idea of guaranteed forward progress. This addition meant that programmers no longer had to handle the case where the transaction aborted explicitly — rather the STM system would deal with retry and guarantee that the system would eventually commit every transaction.

One dimension in STM designs is the distinction between *strongly atomic* and *weakly atomic* STMs. This distinction refers to the STM system's guarantees regarding non-transactional access to memory locations in the read sets and write sets of in-flight transactions [10].[3] A strongly atomic STM system acts like Herlihy's and Moss' TM design: a non-transactional access causes a transaction to abort if the transaction

[3]Blundell *et al.* argue that the programming-language research area has historically used the term Atomicity to refer to what would be known in databases as Atomicity and Isolation. Thus, while the strong vs weak atomicity dichotomy appears to be an isolation issue, the name is nevertheless appropriate.

has the accessed location in its read set or write set. A weakly atomic STM system does not specify what occurs when there is a non-transactional access to a transactionally tracked location — typically the argument in defence of weakly atomic STMs is that the existence of such an access is a programming error.

Strongly atomic STM systems make it easier to interact with non-transactional code by ensuring that the TM execution remains correct in the face of conflict with non-transactional code. However, strong-atomicity increases the overhead incurred by a TM system. As a result, most high-performance STM systems are weakly atomic.

One of the keys to STM performance is the notion of *privatization*, which is the ability to have non-instrumented accesses inside a transaction. Privatization is important because instrumentation in an STM is costly in comparison to normal memory accesses. Privatized accesses are a powerful performance tool. However, they are difficult for an STM compiler to achieve and also STM programmers find it difficult to use them correctly.

The combination of automatic (or no) privatization and instrumentation creates a very simple programming model for transactions, as in Figure 2.2(b), where the programmer indicates the scope of the transaction, the contents are executed in isolation and the results are committed atomically. This programming model has been empirically validated as easier than locks for novice parallel programmers [61].

Though STM has existed for almost twenty years now, it has produced relatively little impact on day-to-day programming. A number of challenges faced by STM systems have prevented broad adoption. The instrumentation and the computation required for conflict detection introduce overhead, in STM systems, causes some concern about performance [17]. As well, interaction with non-transactional code and non-transactional libraries and kernels can be complicated depending on the atomicity model and guarantees provided by the TM system [10, 17, 24]. Efforts continue to improve STM performance through new TM systems such as SwissTM. Other efforts aim at shoring up the guarantees provided by TM systems through improved sandboxing and further investigation of strongly-atomic STM systems [8, 24, 29].

Contention Managers

The contention manager is the software component in a Software Transactional Memory system responsible for determining, when a conflict occurs, what subset of transactions need to be aborted. Contention management was first proposed by Herlihy *et al.* as a generic interface to guarantee progress in transactional systems [42]. One of the key concepts behind their original specification of contention managers was that the responsibility for correctness, and for forward progress, ought to be kept separate.

There has been a substantial amount of work done in this area because Contention Management can change the performance of STM systems [3, 43, 64, 70].

More recently, there have been a number of efforts surrounding Transactional Scheduling, a technique that takes a slightly different approach to forward-progress and performance [28, 56, 79]. Where contention managers control what occurs when conflicts are detected, Transactional Schedulers attempt to proactively avoid conflicts by delaying or moving transactions to avoid interleavings expected to conflict.

2.2.2 Hardware Transactional Memory

HTM research initially was limited to the development of system simulators due to the cost of implementing new designs in real silicon. These HTM simulators range vastly in capability and structure, from very limited forms of TM similar to the original Herlihy and Moss proposal all the way to unbounded HTM systems that allow arbitrary transaction sizes by allowing conflict detection at much higher levels in the memory-hierarchy: from evicted cache lines to swapped out pages [1, 55, 67].

In the last five years Hardware Transactional Memory has begun to appear outside of simulators. The first implementation was by Azul Systems, though it was not broadly described outside of a presentation and some patents, and was only used for Transactional Lock Elision in Java [21]. The next implementation was to be Sun's Rock processor, which was cancelled before it became commercially available, though not before some research into its HTM facilities [26].

IBM's Blue Gene/Q was the first commercially available HTM available. It is on this HTM that we based our study. Further details on the implementation of

HTM in the BG/Q appear in Section 2.3 and Appendix A. IBM has since produced two more HTM implementations, one on zEC12, the current mainframe processor, and one on the upcoming POWER8 processor [14, 45]. Intel has also shipped an HTM implementation called 'Restricted Transactional Memory' (RTM) in its Haswell micro-architecture [44].

All existing HTM systems are strongly-atomic. In contrast to the original specification by Herlihy and Moss, all existing HTM systems provide a Begin instruction to start a transactional region. Within the transactional region *all* loads and stores are transactional, similar to the STM compiler programming model. Only zEC12 supports limited privatization in the form of a *Non-transactional Store instruction*

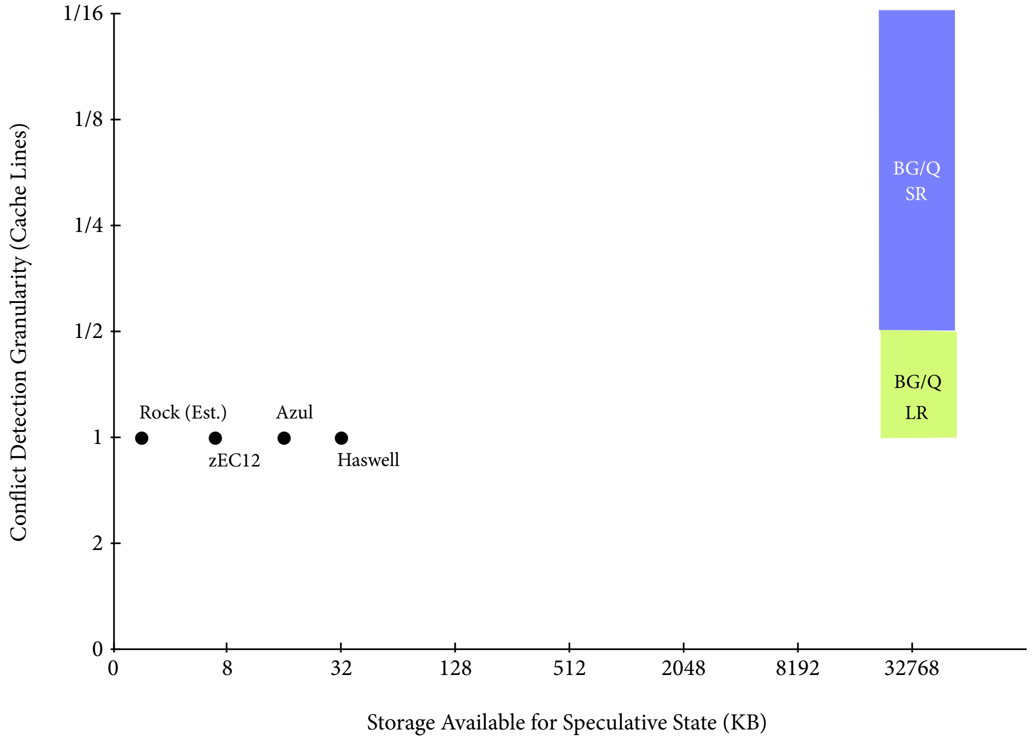

Figure 2.3: A rough summary of existing TM systems in the TM design space. This graphic does not address the additional dimensions of TM induced memory latency or TM overheads.

Figure 2.3 positions the existing TM systems on two dimensions of the design space: conflict-detection granularity and amount of storage available for speculative state. The conflict-detection granularity refers to how close together two accesses from different threads may be before the hardware signals a *false conflict* — a conflict that is not present in the actual application but that occurs in the HTM because of the inability of the hardware to determine that two referenced locations are not the same location. In an ideal system there would be no false conflicts. However, in most HTM systems the conflict-detection granularity is some fraction of the cache-line size because this detection uses existing data caches. The storable speculative state in Figure 2.3 is the maximum space that can be occupied by data written by speculative writes.

The two shaded areas for BG/Q in Figure 2.3 indicate the ranges of conflict-detection granularity are a side effect of a configuration parameter supported by BG/Q, the *running-mode*. The running mode changes the behaviour of the TM system to favour either short-running or long-running transactions. The running modes of BG/Q are discussed in more detail in Section 2.3.1, but in brief the *long-running mode* (BG/Q LR in Figure 2.3) reduces the latency — the time it takes a cache to serve a request — of accesses within a transaction at the cost of affecting cache locality across transactions, while the *short running mode* (BG/Q SR in Figure 2.3) preserves locality across transactions at the cost of increased latency within transactions.

Latency is just one dimension not addressed by Figure 2.3. Other design parameters that are not included in the figure include when conflicts are detected or signalled (this can be either at the point where the conflict occurs, or when the transaction attempts to commit) and the overhead of the TM system and the code required to provide a forward-progress guarantee.

2.3 Blue Gene/Q

IBM's Blue Gene/Q is a supercomputing solution designed for the solution of peta-scale computational problems [39]. The core of the Blue Gene/Q system is the *compute node* consisting of a 1.6 GHz POWER A2 chip with 16 GB of memory running the Compute Node Kernel (CNK) [32]. These compute nodes are combined together in larger groupings to form the total Blue Gene/Q supercomputing system.

Blue Gene/Q shipped with a Transactional-Memory programming system that functions within a single compute node[4]. Blue Gene/Q's TM support is implemented in four layers, shown in Figure 2.4, and described below.

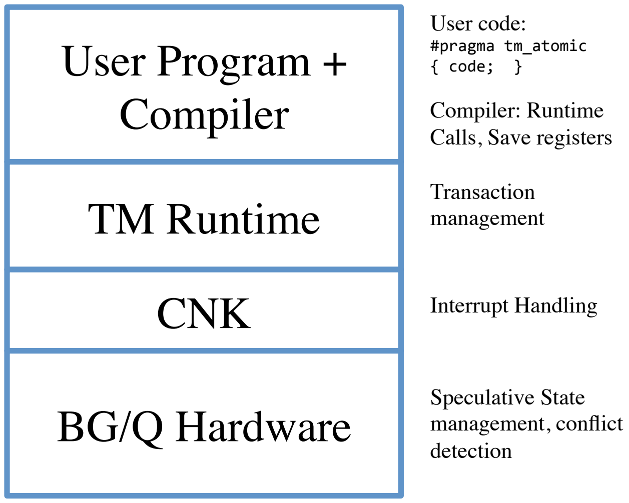

Figure 2.4: BG/Q Transactional Memory Stack

2.3.1 Hardware Support

Each BG/Q compute node has 16 user-accessible POWER A2 cores, and each core has 16K of private L1 cache that is 8-way set-associative, with 64-byte cache lines. The cores all share a 32MB L2 cache that is 16-way set associative.

BG/Q's TM support is implemented in the L2 cache, which has been modified to be *multi-versioned* so that it can store multiple versions of the same physical line. Each version occupies a different L2 way [57]. By default (and in all our experimentation) six of the sixteen sets are reserved for holding non-speculative state. In this configuration the hardware guarantees that speculatively written ways lines will not be evicted by non-speculative writes[5]. The hardware supports *sandboxing* that allows hardware exceptions and access to memory-mapped I/O to be trapped and reduced to a transaction abort.

Transactions are identified by one of 128 Speculation IDs that form part of the cache tag. These Speculation IDs are managed by the L2 cache, which must reclaim

[4] Across-node synchronization must therefore be achieved with a different method.

[5] The hardware can be configured to allow speculative writes into up to 15 of 16 ways, however, if less than six ways are reserved for non-speculative state then non-speculative ways may evict speculatively written ways, aborting the corresponding transactions

them by walking the cache contents. The reclamation process, called *ID scrubbing*, happens at a fixed interval. Therefore the starting of a transaction may have to wait for a Speculation ID to be reclaimed. Reclamation occurs when all the lines marked with a Speculation ID are marked as invalid, or are merged into the non-speculative state.

When the cache directory finds that two transactions have conflicted, it updates the *conflict register*, a status register for the current associated Speculation ID, and triggers an interrupt that is handled by the kernel. The conflict register contains a status bit to indicate if the transaction failed because it would have required evicting a speculative line — called the capacity bit – and a bit that indicates if there was one, or multiple. conflicting transactions. If there are conflicting transactions, the Speculation ID of the conflicting transaction is also available in the conflict register.[6]

Only memory state is tracked transactionally. Thus, support to restore register values must be provided through compiler-generated code for register save and restore.

The L1 cache is unmodified from a standard A2 core. Thus BG/Q supports two transactional execution modes that change how the L1 cache is handled in order to correctly track speculative state:

Short-running mode evicts speculatively written lines from the L1, forcing subsequent reads and writes to go to the L2 cache. Short-running mode is named so because it preserves the contents of the L1 cache across transactions, at the cost of extra read-after-write latency inside the transaction — therefore this mode is suitable for short-running transactions.

A side effect of the short-running mode is aliasing on the bit that indicates that the transaction was aborted because it required another speculative line. The signal aliasing means that the contents of a conflict bit are not accurate in short-running mode, as shown experimentally in Section 6.2.

Long-running mode removes the transactional read-after-write latency of short-running mode by allowing lines to remain in the L1 after writes, at the cost of

[6]We did not explore using conflicting Speculation ID data for policies because the mapping from Speculation ID to transaction is volatile and available infrequently.

flushing the L1 cache at the beginning of a transaction. Long-running mode is suitable for transactions that run for a longer time.

BG/Q supports both eager and lazy conflict resolution for transactions. Eager conflict resolution raises an interrupt to the kernel as soon as a conflicting access is detected. Lazy conflict resolution on BG/Q raises an interrupt to the kernel when either a *jail-mode violation* or a capacity overflow occurs. However, for a memory-reference conflict the interrupt is suppressed until commit time. The *jail mode* is used to sandbox transactions. A transaction that attempts to execute an operation that it is not allowed to execute, such as printing to a terminal, is said to incur a jail-mode violation.

Lazy conflict resolution can allow some transactions that would have aborted in the case of eager conflict resolution to commit safely, when the conflicting transaction has itself been aborted and invalidated before commit is attempted.

All the experiments in this thesis use eager conflict resolution because it was previously found to perform as well as, or better than, lazy conflict resolution [74].

2.3.2 Software Support

Compiler Support

Software support for BG/Q's TM begins in the compiler. IBM's XL C, XL C++ and XLF Fortran compilers for BG/Q provide support to create transactions in a program in the form of compiler pragmas, as shown in Listing 2.4

Listing 2.4: A TM pragma example

```
#omp parallel for
for (int i=0; i < N; i++) {
  #pragma tm_atomic
  {
    A[i] = B[ C[i] ];
  }
}
```

The code region marked by the pragma as a transaction is compiled into register save code and calls into the TM runtime.

TM Runtime

The TM runtime manages transactional execution, starts speculation, handles rollbacks, controls non-speculative serialization and commits state. The runtime consists of functions to initialize the TM system, a transaction-begin function, a transaction-end function, and a rollback handler. The begin and end functions are inserted by the compiler. By design the vast majority of state is thread local to avoid the need for synchronization inside the runtime.

When the kernel aborts a transaction, the rollback handler in the TM runtime is called. Inside the rollback handler, the TM runtime can query the conflict register for more information on the rollback, and can decide whether or not to run the transaction with non-speculative serialization.

Non-speculative serialization in BG/Q is accomplished by acquiring a lock, known as the irrevocable token. The irrevocable token is used to prevent new transactions from committing while that token is held. Allowing a transaction to commit while the irrevocable token is held would violate the isolation property.

Let T_N be a transaction executing non-speculatively and T_S be another transaction that starts speculative execution while T_N is executing. There are two possible solutions to prevent the commit of T_S while T_N is running:

- With *Eager Lock Checking* T_S reads the status of the token after it begins speculative execution, but before any of the T_S user code has executed. T_S has to abort with a conflict because the irrevocable token is both in the write set of T_N and in the read set of T_S. With eager lock checking any time the irrevocable token is acquired (written) by a non-speculative transaction all in-flight transactions must abort because all transactions have the irrevocable token in their read-set.

- With *Lazy Lock Checking* T_S reads the status of the irrevocable token immediately before commit. Checking the token at the end allows T_S to continue executing simultaneously with T_N. However, T_S will abort if the lock is held at its commit time to prevent violations of atomicity.[7]

[7] For example: 1) A T_N could write a value x. 2) T_S could then read x, use it in a computation, write the result of that computation, and commit. 3) If T_N subsequently updates the x, then T_S would have computed based on an internal state of T_N, violating the atomicity guarantee of T_N.

In BG/Q the short-running mode uses eager lock checking while the long-running mode uses lazy lock checking. This choice of lock-checking policy is based on prior investigation into lock checking, though that investigation found only a small effect from changing lock checking from eager to lazy in real benchmarks. In both cases, the lock is also checked prior to transaction start to avoid wasting Speculation IDs.

The TM runtime collects summary statistics on transactional events, counting transactions committed, aborted transactions, transactions serialized for JMV and transactions serialized by the serialization manager[8]. The counters are collected thread-locally and summarized across all threads in our presentation.

Kernel Support

In BG/Q the kernel support for TM consists of a handler for the conflict interrupt raised by the hardware. The conflict handler examines the relevant conflict registers and calls the rollback handler with a return code to indicate why the transaction has been rolled back.

For more details on BG/Q's TM design, please see the included Appendix A.

[8]In some cases the serialization cause is split into further counters.

Chapter 3

The Effect of Serialization Management on Performance

This thesis studies the effect of Serialization Management on performance. To understand this effect, a through investigation of both existing and new serialization managers must be undertaken.

An ideal serialization manager would have the following properties besides high performance:

1. Consistent Performance: Different applications will likely have different demands for serialization. Ideally a single tuning would accommodate as many applications as possible because most users will not tune their setup. Therefore, the default parameters should be performant.

2. Simple Tuning: In cases where tuning is required, the tuning should be as simple as possible. Ideally there would be a single parameter. A single parameter makes tuning reasonably easy, even for relatively inexperienced programmers. Multiple parameters introduces complexity in the search for a good set of values, while also dramatically increasing the search space.

3.1 Evaluation Pitfalls

The evaluation of serialization managers in order to compare and contrast them faces a number of pitfalls that could mislead a simple exploration.

The first pitfall is the existence of a number of new BE-HTM systems. As was seen (partially) in Figure 2.3 different HTM implementations occupy different positions in the TM design space. Different placement in the TM design space will change TM execution, and adjust the requirements on a serialization manager. For instance, systems with finer-granularity conflict detection may see less aborts and require less serialization leading to lower demands on the serialization manager.

A second pitfall is TM applications themselves. Serialization-manager tuning is an important element in the comparison of managers because real-world applications using TM are likely to have varying contention levels, as are found in TM benchmarks.

A very desirable property of serialization management would be configuration portability, referring to the question of how a particular set of choices for serialization management adapts to different programs or platforms. A portable solution to serialization management should provide similar results for similar programs, or similar scaling on new platforms.

How these elements of BE-HTM systems interact has not been widely studied, and thus their combination raises a number of new research questions that this thesis answers:

1. How much does a serialization manager's performance depend on the tuning of its parameters?

2. How stable is a particular serialization manager's performance across different programs with the same tuning?

3. How stable is a serialization manager's performance across different BE-HTM systems for the same tunings?

3.2 New Serialization Managers

To understand how Serialization Management affects performance, this thesis needs to explore the design space of serialization managers. Serialization Management is the BE-HTM counterpart to STM Contention Management. Thus the question "Can advice or inspiration be obtained from STM Contention Management to develop HTM serialization managers?" drove part of the exploration presented here.

One dimension of the serialization-manager design space is whether or not the manager is static or dynamic. A dynamic serialization manager reacts using the history of the program being executed, making different decisions in the same circumstance based on changed history. A static manager makes the same decision in the same circumstance, regardless or history. While dynamic policies are intuitively attractive, this thesis asks the question: Is there evidence that dynamic serialization managers should be pursued?

While trying to determine how Serialization Management relates to TM system design, another question was raised: Are there serialization related policies that are independent of the manager, and how do they affect performance? For example, one policy followed by all managers in this thesis is 'Serialize on Sandboxing Violation'.

Once the serialization managers are built, some design decisions that were rooted in intuition can be studied. For instance, is it important for a serialization manager to minimize wasted work? While intuition drives us in this direction, careful thought seems to argue that this is not a good metric because the performance in TM systems is partially provided by speculation that is, in a sense, wasted work.

3.3 Answering the Research Questions: My Approach

In order to help answer all the research questions, a large-scale study of serialization managers was performed on BG/Q, consisting of the following elements.

1. To help understand transactional program performance in more detail I developed a tool that gives deep insights into transactional execution on BG/Q. We call it the Transactional Event Profiler (TEP), and it is described in Chapter 4.

2. I also designed and built seven new serialization managers. Some of these serialization managers were inspired by STM contention managers, however, they required adaptation to the unique controls available to a serialization manager.

3. A very large-scale examination of the serialization managers used the methodology described in Section 3.4.

   To help answer the research questions, whenever a serialization manager presented a meaningful choice of parameter tuning, I explored a broad range of values for each parameter. This is not to suggest that the average programmer should have to pursue a tuning strategy similar to this. The goal of this exploration is to provide bounds on performance across tunings.

4. All experiments were run in both long-running and short-running mode to allows us to investigate the generalizability of tunings. These modes are two separate (though nearby) points in the BE-HTM design space (Figure 2.3) with differing conflict-detection granularity and latencies.

3.4 Experimental Methodology

Blue Gene/Q is the experimental platform because the forward-progress-guarantee programming model was already implemented. Thus, exploring serialization policies required minimal additional implementation effort.

All programs were compiled with the IBM XL C/C++ compilers, using a pre-release version, with the compiler flags `-O3 -qhot -qtm`. The flags activate the third level of optimization, 'Higher-Order Transformations' on loops, and transactional memory.

Timers are inserted into the programs that measure the wall-clock time to execute only the parallel section. This is done so that the ideal speedup would be linear – *e.g.* 4x for 4 threads – making scaling comparable across benchmarks.

In a parallel system, it is expected that there will be run-to-run variance in the execution time: A lucky execution could avoid waiting on synchronization and complete sooner than an unlucky execution that spends time waiting. To account for run-to-run variance, each benchmark was run 5 times, and in many plots the mean-speedup is plotted, along with a 95% confidence interval computed from the t-distribution, assuming that speedups are normally distributed. In other plots where the mean is not meaningful, all data points are plotted to show general trends.

All runs used the BG/Q Hardware Performance counters to collect statistics about L1 misses and the number of instructions executed [49]. The counters collected were:

- `PEVT_L1P_BAS_MISS`: Counts prefetchable load misses in the L1P unit.
- `PEVT_LSU_COMMIT_LD_MISS`: Counts loads that missed the L1 data cache[1]. In general, the number of L1 misses reported in this thesis is the sum of this and the previous counter.
- `PEVT_INST_ALL`: Counts instructions completed. This counter is useful for measuring the amount of work done, and will be used to investigate the nature of 'wasted' work.

3.4.1 Metrics

We computed two metrics, measures of transactional execution, to help describe and compare the execution of our benchmarks at a high level.

Serialization Fraction This could also be described as serializations-per-commit, and is computed as the ratio of serialized transactions to those committed. By the nature of serialization, this value is bounded within the $[0, 1]$ interval.

Abort Fraction The number of aborts-per-commit, computed as the ratio of the number of serialized and rolled-back transactions to committed transactions. This value is not bounded within any interval because there can be more aborts than commits.

Capacity Fraction The number of capacity-aborts-per-commit, computed as the ratio of the number of serialized and rolled-back transactions where capacity overflow was reported to committed transactions. This value is not bounded within any interval because there can be more capacity aborts than commits.

[1] According to the BGPM documentation, the `PEVT_LSU_COMMIT_LD_MISS` event does not include cache-inhibited loads. However, it is unclear from the documentation if the event captures the forced L1 misses caused by short-running mode.

3.5 Benchmarks

This evaluation uses the STAMP benchmark suite, version 0.9.10 [52], and the RMS-TM benchmark suite, version R3 [47].

3.5.1 STAMP

The STAMP suite is the de-facto standard in TM evaluations, containing eight applications that span a variety of transactional workload characteristics and domains:

bayes A machine-learning benchmark that learns a bayesian network.

genome A genomics benchmark that reconstructs a genome from gene-sequences.

intruder An implementation of a signature-based network-intrusion detection system.

kmeans A version of the k-means clustering algorithm from machine-learning.

labyrinth A parallel maze-solving algorithm.

ssca2 Kernel 1 from the HPCS graph benchmark [6].

vacation A simulation of a travel reservation system.

yada (Yet Another) Delaunay mesh refinement Algorithm.

The STAMP benchmarks are used for two reasons: First, since it is the de-facto standard suite, this allows relatively easy comparison against other work due to the common deployment of this suite. Second, the STAMP benchmarks as a whole reflect one particular belief about what transactional memory programs will look like — relatively large transactions with long running times, where the majority of time in the program is spent inside of transactions.

The benchmarks were run with the options recommended by the suite authors for real hardware, presented in Table 3.1.

Benchmark	Running Options
`bayes`	`-v32 -r4096 -n10 -p40 -i2 -e8 -s1`
`genome`	`-g16384 -s64 -n16777216`
`intruder`	`-a10 -l128 -n262144 -s1`
`kmeans`	`-m40 -n40 -t0.00001 -i inputs/random-n65536-d32-c16.txt`
`kmeans-high`	`-m15 -n15 -t0.00001 -i inputs/random-n65536-d32-c16.txt`
`labyrinth`	`-i inputs/random-x512-y512-z7-n512.txt`
`ssca2`	`-s20 -i1.0 -u1.0 -l3 -p3`
`vacation`	`-n2 -q90 -u98 -r1048576 -t4194304`
`vacation-high`	`-n4 -q60 -u90 -r1048576 -t4194304`
`yada`	`-a15 -i inputs/ttimeu1000000.2`
`apriori`	`<input> -s 0.0075`
`fluidanimate`	⟨threads⟩ `5 in_300K.fluid`
`hmmcalibrate`	`-num 500 -seed 33 globin.hmm`
`hmmpfam`	`Pfam_ls_300 7LES_DROME`
`hmmsearch`	`globin.hmm 2000_uniprot_sprot.fasta`
`scalparc`	`F26-A32-D125K.tab 125000 32 2`
`utilitymine`	⟨input⟩ `logn1000_binary 0.01`

Table 3.1: Benchmark Options

Though the STAMP suite provides eight benchmarks, most of the experimental evaluation reports only results for five benchmarks. Three benchmarks are very uninformative for the purposes of the investigation because their behaviour is dominated by effects that are not within the scope of fallback policies:

1. `bayes` is excluded because it is a non-deterministic benchmark. The time to complete execution depends on the *order* in which transactions are committed. A side effect of bayes' non-determinism is that run-to-run variance is very high. This high variance prevents accurate description of any effects from changing the TM runtime without a larger number of repetitions than was feasible.

2. `labyrinth` is excluded because its running time is dominated by a transaction that never completes on BG/Q. This transition copies a large array inside the transaction. This copy exceeds that amount of speculative state that can be stored by BG/Q.

3. `ssca2` is excluded because its bottleneck is waiting for new SpecIDs to be made available, as its high rate of transaction creation exceeds the rate of SpecID reclamation in the L2 cache [74].

`kmeans` and `vacation` have two variations described in the STAMP documentation, one for low contention and the other for high contention. However, most of the performance study presents only the results for the low-contention version because, from the perspective of Serialization Management, the low and high variants typically show similar behaviour. When they differ, results for both variants are presented or discussed.

We have attempted to account for many of the weaknesses of the STAMP benchmark suite [62], though have shied away from major changes to the suite in order to attempt to preserve comparability between different publications.

Normally, the sequential version of the STAMP suite uses the system allocator `malloc` but the parallel version uses a parallel allocator (`memory.c`). STAMP's parallel allocator does not free memory, and is simpler than `malloc`, leading to misleading speedups. It is important to specify the allocator when discussing STAMP results because the suite is known to be sensitive to allocation patterns [7]. STAMP's speedups presented in this thesis are relative to a sequential version of the benchmark, using the included parallel allocator for both parallel and sequential runs. The sequential time used to compute speedups is the minimum of five trials. In our experimentation some benchmarks — `yada` , `vacation` and `intruder` — showed a substantial performance difference due to the allocator, that introduces a constant-factor difference when compared to the results published in other papers where `malloc` is used for the sequential baseline.

3.5.2 RMS-TM

RMS-TM is a newer benchmark suite that takes its applications for the field of "Recognition, Mining and Synthesis":

aprori An association rule mining algorithm.

fluidanimate A hydrodynamics simulation.

hmmcalibrate Calibrates a genome sequence profile model

hmmpfam Searches a Hidden Markov Model database.

hmmsearch Find similar sequences from a database.

ScalParC A decision tree algorithm.

utilitymine A rule mining algorithm.

The options used to run RMS-TM are shown in Table 3.1. Results for `utility mine` and `apriori` only exist for 1-8 threads because the input file needs to be split among threads according to an 'offset file', however offset files exist only for 1-8 threads, and the method for creating them does not appear to be documented. Similar to STAMP, speedups presented are relative to the fasted sequential execution with no TM overhead of five trials.

Most benchmarks from RMS-TM exhibit a low relative critical-section size (the ratio of time spend inside a transaction vs outside during parallel execution) and have low contention. Therefore there is little opportunity to exercise serialization management.

This benchmark suite was used to reflect another view of future transactional memory programs, essentially programs where fine and medium grained locking has been replaced with transactions.

3.6 Benchmark Characterization

Each benchmark has particular characteristics that affect the response to changing the serialization managers.

This section is supplemented by Section A.5 and Table A.1 in Appendix A.

3.6.1 Contention

The amount of contention experienced by the STAMP benchmarks vary dramatically. To quantify the amount of contention, Table 3.2 shows the ratio of aborted to committed, in long- and short-running modes. This percentage can be greater than 100% because of transaction retry.

One particular stand-out data point is `vacation`, that sees dramatically fewer aborts in short-running mode than in long running mode. This variation is due to the

Table 3.2: Aborts per commit and Capacity-overflow aborts per commit in the STAMP and RMS-TM benchmarks run with MAX-RETRY set to allow 10 retries with capacity-induced serialization and rollback delay enabled. The results for 2 threads are elided because they have negligible counts even in short-running mode.

Name	threads	Abort Fraction (Short)	Abort Fraction (Long)	Capacity Fraction (Short)	Capacity Fraction (Long)
apriori	4	0.4405	0.3294	0.0069	0.0000
	8	1.0441	0.8118	0.0364	0.0000
fluidanimate	4	0.0000	0.0001	0.0000	0.0000
	8	0.0001	0.0001	0.0000	0.0000
	16	0.0002	0.0001	0.0000	0.0000
	32	0.0012	0.0006	0.0001	0.0000
	64	0.0010	0.0011	0.0000	0.0000
genome	4	0.0034	0.0213	0.0008	0.0000
	8	0.0134	0.0420	0.0018	0.0000
	16	0.0393	0.1073	0.0034	0.0000
	32	0.0744	0.1185	0.0040	0.0000
	64	0.1559	0.1187	0.0088	0.0011
hmmcalibrate	4	1.3582	0.3447	0.0057	0.0000
	8	2.6838	0.7622	0.0405	0.0000
	16	3.6589	1.4703	0.1311	0.0000
	32	4.1091	3.4674	0.2593	0.0000
	64	4.4687	5.9864	0.3972	0.0000
hmmpfam	4	3.5068	1.8278	0.0063	0.0000
	8	5.3166	3.9553	0.0420	0.0000
	16	5.9488	6.5306	0.2276	0.0000
	32	5.9438	7.1801	0.3700	0.0000
	64	6.4394	7.3492	0.3350	0.0012
hmmsearch	4	0.3667	0.1712	0.0027	0.0000
	8	1.0680	0.5204	0.0075	0.0000
	16	2.0539	1.1161	0.0171	0.0000
	32	3.5236	2.4567	0.0353	0.0000
	64	4.7928	4.0839	0.0630	0.0060
intruder	4	0.1233	0.1140	0.0082	0.0000
	8	0.3433	0.2491	0.0224	0.0000
	16	0.6386	0.4487	0.0427	0.0000
	32	0.8874	0.7142	0.0521	0.0000
	64	1.0254	0.8166	0.0598	0.0000
kmeans	4	0.0129	0.0030	0.0000	0.0000
	8	0.0780	0.0402	0.0000	0.0000
	16	0.5073	0.4131	0.0002	0.0000
	32	1.3458	1.2705	0.0037	0.0000
	64	1.6456	1.5995	0.0078	0.0000
ScalParC	4	0.0911	0.1496	0.0002	0.0000
	8	0.2044	0.3548	0.0003	0.0000
	16	0.4709	0.8685	0.0007	0.0000
	32	1.5086	1.8051	0.0018	0.0000
	64	1.5957	1.8067	0.0025	0.0000
UtilityMine	4	0.0135	0.0043	0.0036	0.0000
	8	0.0180	0.0110	0.0013	0.0000
vacation	4	0.0001	0.1180	0.0000	0.0000
	8	0.0003	0.2261	0.0000	0.0000
	16	0.0017	0.3229	0.0000	0.0000
	32	0.0111	0.3653	0.0002	0.0000
	64	0.0375	0.2608	0.0002	0.0000
yada	4	0.5709	0.4079	0.0030	0.0000
	8	0.7418	0.5590	0.0057	0.0000
	16	1.1090	0.7756	0.0186	0.0000
	32	1.8556	1.0128	0.0676	0.0000
	64	2.4957	1.4211	0.1101	0.0000

change in the conflict-detection granularity between long and short running modes. The change in detection granularity renders vacation in short-running mode largely insensitive to changes in serialization manager and tuning.

3.6.2 Capacity Overflow

Table 3.2 also shows that the dramatic difference in capacity-overflow reporting between long- and short-running modes in BG/Q is caused largely by the aliasing on the conflict bit discussed in Section 2.3.1.

Chapter 4

The Transactional Event Profiler

Many TM implementations, including BG/Q, provide summary counters of the number of committed and aborted transactions. However, to investigate the potential efficacy of more sophisticated serialization managers, these counters are insufficient.

For instance, consider two transactional workloads each of which incurs 10,000 aborts, where the first transactional application takes 1 second to run, and the other application runs in 10 seconds. Clearly these two programs experience very different contention levels, yet this difference remains invisible in summary counters.

Disambiguating these kinds of confusing situations was the main motivation behind the design of the Transactional Event Profiler (TEP), a tool that gives us insight into dynamic conditions by capturing the time series of transactional events inside the TM system.

4.1 TEP Design and Implementation

The Transactional Event Profiler (TEP) a is a software profiling system that provides a high-resolution time-series view of transactional behaviour in programs run on BG/Q. The TEP system allows both macro and micro views of transactional behaviour that have influenced our understanding of benchmarks, serialization, and serialization managers.

Event profiles are gathered by the TM runtime, triggered by events such as begin, rollback, serialization and commit. These events are stored in per-thread buffers, and are dumped to a log-file at the end of program execution by the runtime's `atexit` shutdown handler. An event profile consists of a list of event tuples: (*TS*, *T*, *TX*, *TID*, [A_1, A_2]), where:

TS is the timestamp read from a high-resolution counter.

T is an event type.

TX is a unique identifier for each static transaction. In our implementation, *TX* is the address of the instruction that starts the transaction

TID is the event's originating thread ID.

$\boldsymbol{A_1}$ and $\mathbf{A_2}$ are optional auxiliary data that can be used to save other data to the profile. For example, A_1 could be used to save the number of rollbacks used on each serialization.

The event profiles collected are post-processed by analyzers, which are discussed in more detail in Section 4.3.

4.2 Limitations of the TEP approach

The TEP design currently requires large amounts of memory for the buffers. Applications will fail if the program's working set and TEP buffers are larger than 16GB because the BG/Q compute node does not implement virtual memory. Experimental evaluation revealed that most TEP logs for the STAMP benchmarks remain less than 1GB, however, larger programs would be expected to have larger logs.

Transaction IDs are currently derived directly from the binary file for the transactional application. This source of IDs is not ideal because comparing TEP logs between different compilations of the same benchmark requires inferring a mapping between the two IDs and it is very labour intensive to recover the mapping between a transaction ID and a source location.

Ideally, compiler support should be deployed to provide (1) smaller TX IDs, reducing log size, and (2) a constant mapping between each ID and a source file location. However such support is not yet implemented in the TEP.

4.2.1 Probe Effects

Event profiles are imprecise because *probe effects* may change the behaviour under study, as the insertion of instrumentation may change interleavings, caches and other aspects of the system. The imprecision introduced by probe effects may be acceptable for analyses that use aggregated values or that are interested in relative rates between events, transactions or applications. However the overhead and interference of instrumentation must always be kept in mind.

While probe effects are impossible to avoid completely without a hardware solution, we made design choices to try to minimize them:

1. The profile buffers are thread local to avoid any synchronization overhead for the profiling.

2. The event-profiling functions are made as short as possible and are inlined to avoid function-call overhead inside the runtime handlers.

3. The event-profiling system avoids memory allocation overhead as much as possible by allocating buffers in very large chunks. A typical profiled execution, in our experience, allocates less than ten buffer-chunks per thread.

4. To save space, and to avoid one memory write per logged event, the TEP does not explicitly store the thread origin in the memory buffer or in the saved file.

   The buffers are saved on disk, one thread buffer at a time, in thread order. Analyzers can infer the thread ID of the transactions by watching for breaks in timestamp monotonicity as they process the profiles. Each such break indicates a thread change.[1] This is safe for BG/Q as the timestamp counter we use is synchronized across cores and there is no thread-migration in BG/Q.

Figure 4.1 compares the STAMP benchmarks' runtimes (the left two columns of graphs) and rollback counts (the right two columns of graphs) when run with the

[1]Thread ID recovery can be accomplished with one line when processing the profile output with the language R: `log$thread=cumsum(c(FALSE, diff(log$timestamp) < 0))`. This R statement creates a new column `thread` on the data frame `log` that contains the thread ID. Many thanks to G. Grothendieck on StackOverflow for this elegant and idiomatic R solution.

Transactional Event Profiler (dark blue bars) and without (light blue bars). `vacation` and `yada` (long running mode, 64 threads) are at the opposite ends of spectrum for probe effects because both have their results badly distorted by the TEP in opposite directions. `vacation`'s performance at 64 threads is improved by 60% when run with TEP, and `yada` degrades by 280%. The number of aborts for `yada` is tripled by the addition of the TEP, whereas `vacation` sees up to a 30% drop in aborts.

Despite the distortions caused by the TEP, in most cases the trends as the thread count changes remain similar, but with changed magnitude.

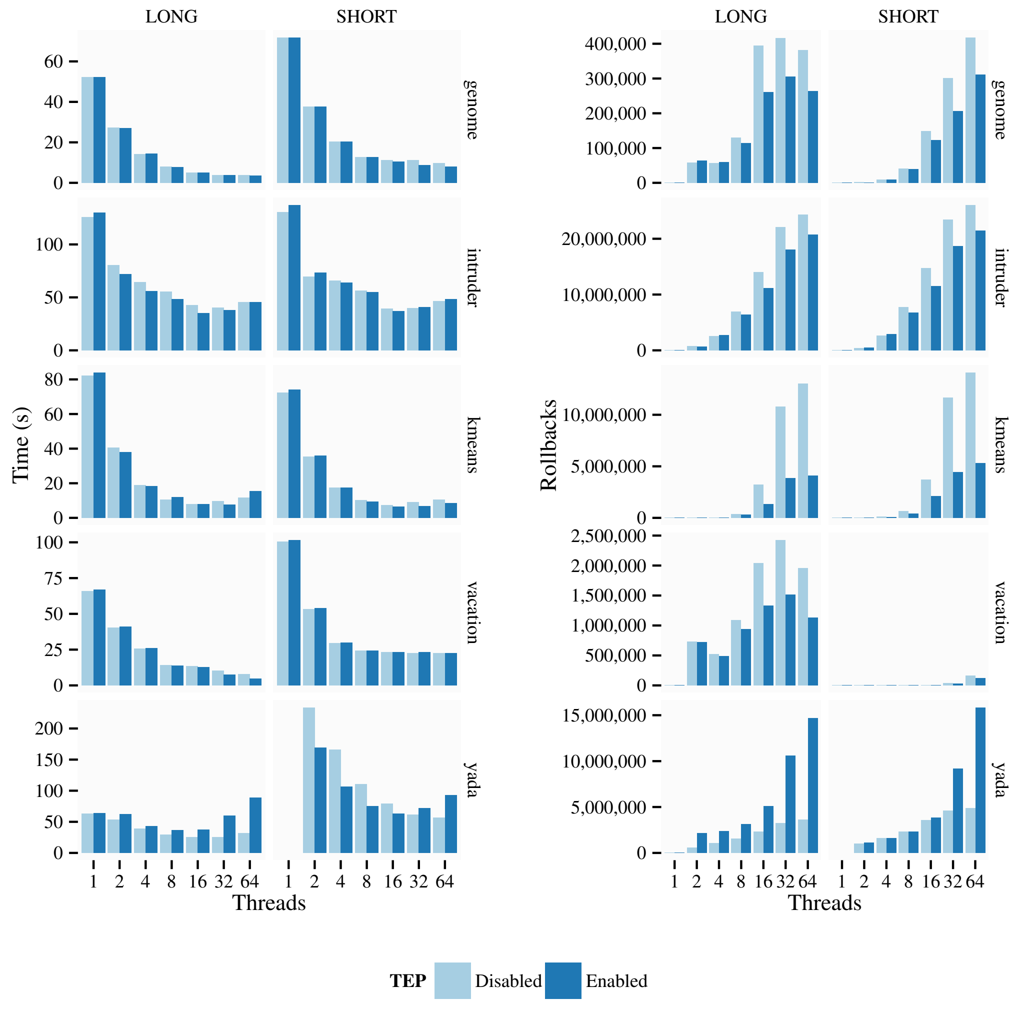

Figure 4.1: Comparing runtime and rollback counts for STAMP benchmarks run both with (dark blue bars) and without (light blue bars) Transactional Event Profiling, using Max-Retry allowing 10 retries as the serialization manager, in both Long and Short running mode.

4.2.2 Hardware Limitations

Some proposed transactional profiling systems not only track begin and end events, but also the read and write sets [48]. Read and write set information can provide interesting data on conflicting data. However, it is impossible to collect this data, in the current form of BG/Q's TM system, because read and write sets are managed in the hardware in such a way that they are not exposed to the software system[2].

4.3 A Sample of Event Log Analyzers

The raw event profiles can undergo a variety of analyses and visualizations, limited only by the imagination of the implementor. I have made substantial use of the event profiles, and have created a number of bespoke analyzers to answer specific questions about implementations. Here I discuss analyses and visualizations that I believe should help in answering our research questions.

4.3.1 Visualizing Event Rates

By looking at the rate of events over time we can discover interesting properties of benchmarks, such as relative rates between transactions, or phased behaviour in benchmarks.

Figure 4.2 shows the rate of each transactional event type aggregated across all threads. Phased behaviour can be seen in the dramatic change in transaction starts, and the small drop in MaxRetry-induced serializations that occurs about 9 seconds into execution. The rate of retry in Figure 4.2 never drops beneath the rate of transaction starts. Given that the retry rate is between two and three times the start rate, it can be inferred that the average transaction is retried between two and three times over the course of execution, indicating that `yada` is a high-contention benchmark.

Event rates can be split per-thread, such as in Figure 4.3, to show how transactional behaviour need not be uniform across threads: For example, `genome`'s thread 0 has a

[2] Read and write sets on BG/Q can be read, however they must be read before commit, and need to access the cache via a backdoor that is very slow to access. The capability was intended for debug and is far too slow for profiling use.

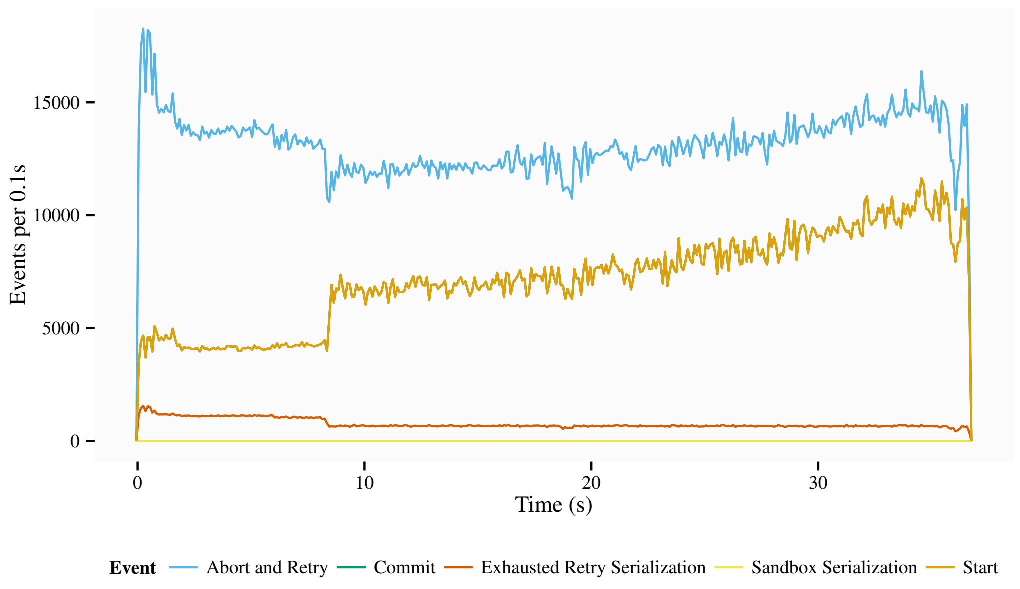

Figure 4.2: Event rates for `yada` demonstrating program phases and consistent rate of serialization. Run with 16 threads in long-running mode with MAX-RETRY and 10 allowed retries

substantially different profile than thread 1 during genome's second phase (3 seconds onward). Different profiles for different threads could be rooted in the application algorithm, as appears to be the case for `genome`, or could provide a visual indicator of starvation by showing reduced rate of commits for a subset of the executing threads.

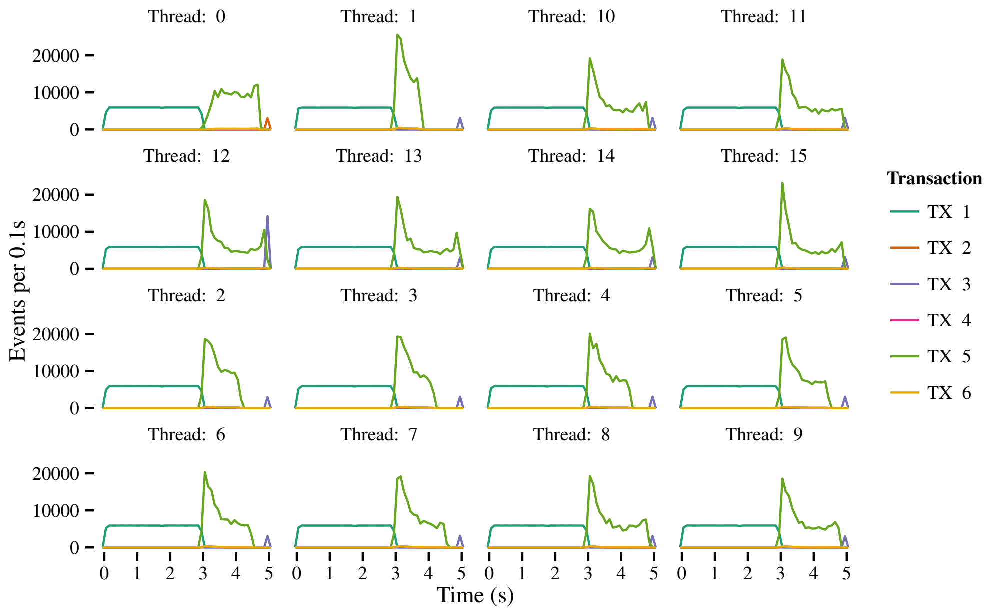

Figure 4.3: Transaction rates for genome showing how transaction activity can vary across application threads. Run with 16 threads in long running mode, using the MAX-RETRY with 10 retries allowed

4.3.2 Visualizing and Comparing Dynamic Transaction Execution Lengths

An event log can be processed to provide the execution length of each dynamic transaction, from start to commit. This data can be aggregated in histograms by each static (source-code level) transaction as can be seen for `genome` in Figure 4.4. This figure shows the dynamic transaction lengths for the six transactions in genome.[3]

Histograms of dynamic transaction lengths reveal the variability in the runtime of transactions, both static and dynamic. For instance, the histograms in Figure 4.4 indicate that transaction TX 1 runs 10 times longer (notice the different scales for frequency), on average, than any other transaction in `genome`, while also being one of the most common by frequency.

Some transactions exhibit multi-modal behaviour with regards to execution time. This is the combined result of multiple control flow paths through transactions, varying amounts of contention and changing cache contents leading to abort and

[3]For better reading of this presentation the transaction IDs reported by TEP were replaced by labels TX 1, TX 2, *etc.*

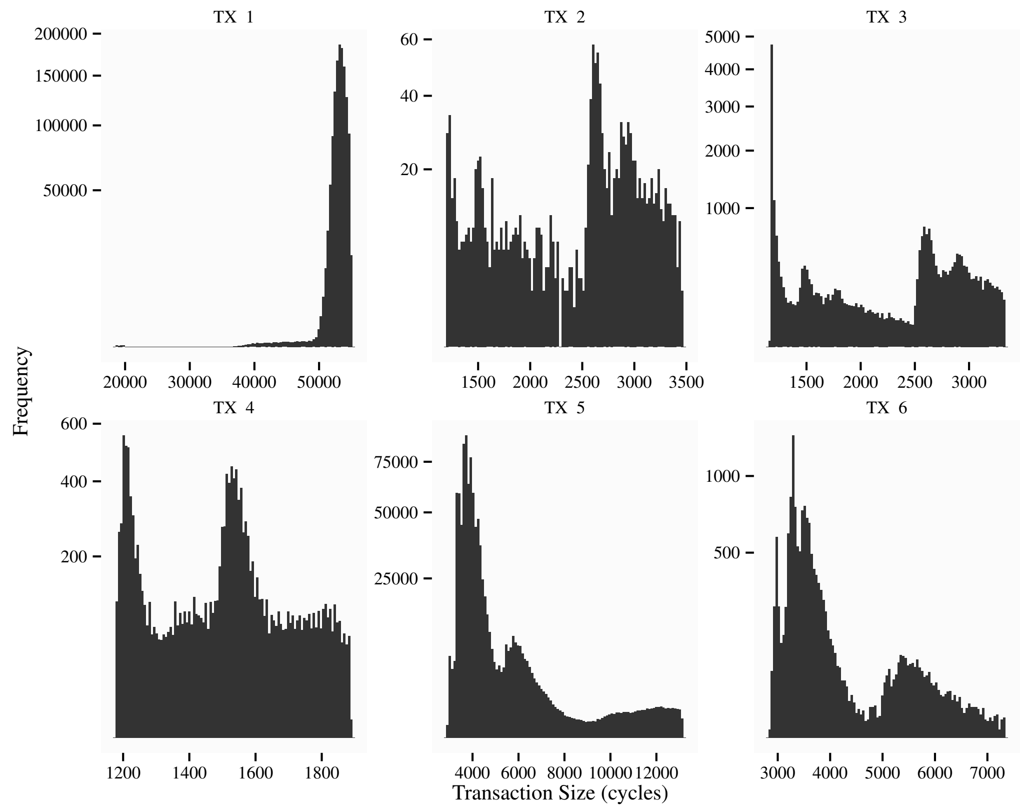

Figure 4.4: Histogram of dynamic transaction execution times for the six static transactions in `genome`. Run with 16 threads in long running mode, using the MaxRetry with 10 retries allowed. Each histogram drops the top 10th percentile of data to avoid the plots being swamped by outliers, and has 100 bins. Note the non-linear y-axis and the different scales in the plots.

restarts, possibly many.

4.3.3 Micro-level analysis and visualization

Analysis of the behaviour of individual dynamic transactions allows insight into the process by which an individual transaction is committed, and the timeline of events that lead to that point.

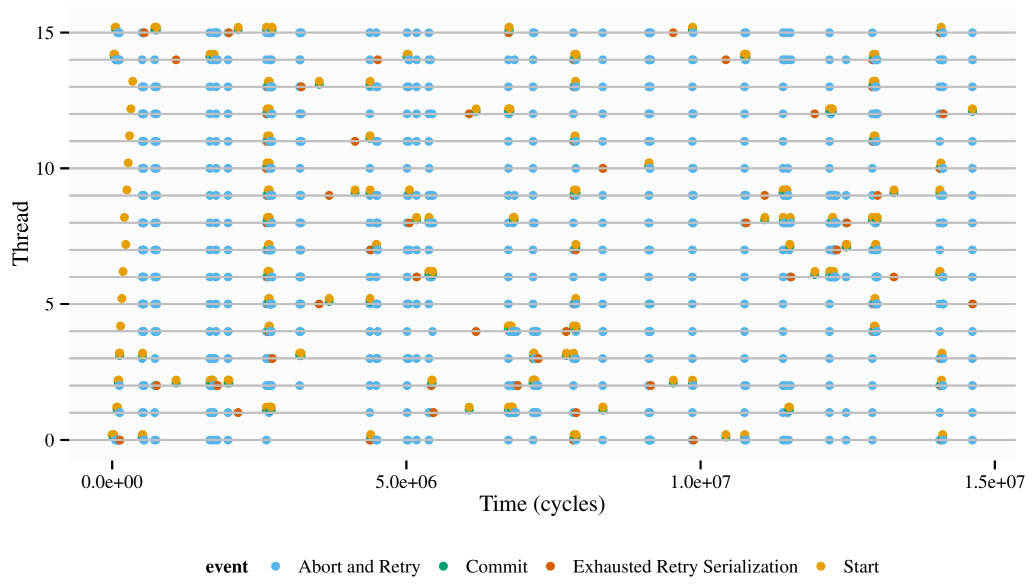

Figure 4.5: Individual events plotted for the first $\frac{1}{4000}$th of transactional execution time for yada Run with 16 threads and no rollback delays in Long Running Mode.

One possible microscopic analysis for an event log is to show the timeline of execution, such as the graph in Figure 4.5 which contains the timeline for the first $\frac{1}{4000}$th of transactional execution time for the benchmark yada. Each row corresponds to a single thread, and each dot corresponds to a single transactional event. Time 0 is set to the time of the earliest transactional event.

The sea of blue dots — corresponding to aborted transactions retrying — and red dots — corresponding to transactions serializing to complete — in Figure 4.5 indicates a high-contention portion of `yada`'s execution. This particular log was made with no attempt to space out transactions on retry through backoff. Thus, the vertical columns correspond to the Convoy effect described by Bobba *et al.*. The convoy effect appears because after a first abort the transactions restart sufficiently close to each other that the same conflict repeats and slows progress [11].

4.3.4 Visualizing Parameter Evolution

We have made use of the event profiles to understand the behaviour of new serialization managers with a variety of bespoke analyzers. Event profile tuples can be annotated with any value from the runtime to visualize parameter values as they fluctuate over time. This visualization has helped discover flaws in some designs and inspired new approaches.

Section 5.4 introduces a serialization manager called LimitMeanST that models the mean transaction length — μ_{exec} — during program execution. Dumping the calculated mean to the event log enabled the generation of the plot in Figure 4.6, that shows the serialization manager modelling the mean for each static transaction[4].

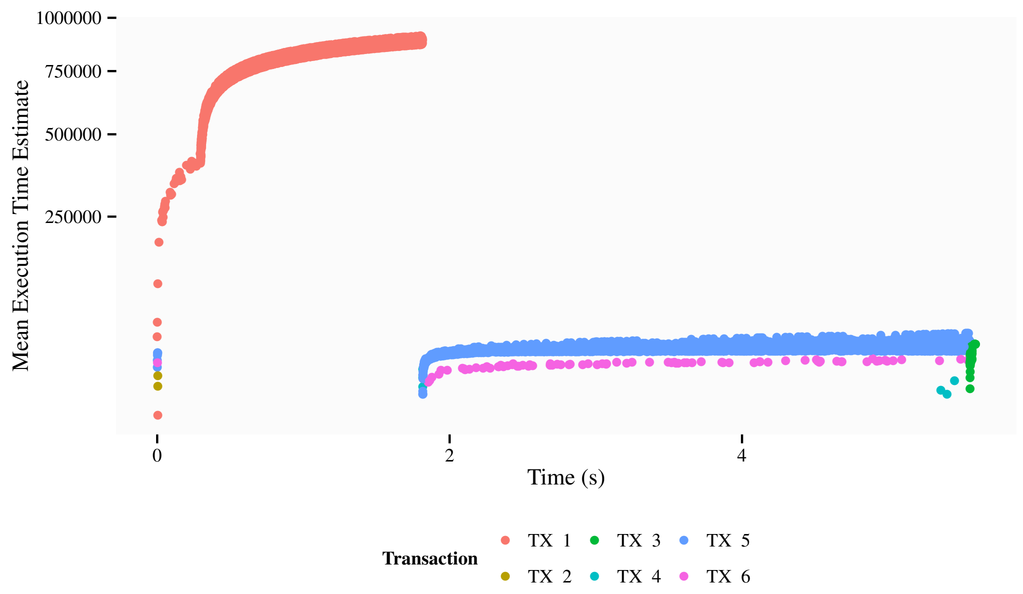

Figure 4.6: Showing LimitMeanST modelling μ_{exec} for genome run with 64 threads, Long running mode. Note the non-linear y-axis. Only 0.06% of data points were plotted to avoid exceeding technical limits in the document.

LimitMeanST adapts to changing execution time for transactions, as can be seen by the kink in the curve for transaction TX 1 occurring at around $\frac{1}{4}$ seconds.

[4]The transaction IDs in Figure 4.6 are incomparable with the transaction IDs in Figure 4.4 because these profiles are generated from different binary files, as described in Section 4.2.

4.4 Lessons from the TEP

The phasing of retry rates with drastic changes seen at certain points in execution seen in both Figures 4.2 and 4.3, as well as the large variance in transaction execution time recorded in Figure 4.4 suggest that dynamic policies that can change their behaviour over time may be required to extract maximum performance from BE-HTM.

Dramatic changes in abort rates indicate that there are likely points in transactional execution that demand different responses from the serialization manager. Higher contention could demand more retries before serialization to reduce the chance of serialization convoy, or could demand more serialization because less work is being accomplished.

The large variance in transaction execution time shows that some transactions may benefit from being treated differently. This applies both to static (source code level) transactions, where some transactions may demand serialization more often, and dynamic (a particular instance) transactions, where some transactions run much longer than average.

Chapter 5

Explored Serialization Managers

We developed a number of serialization managers, some from scratch and others inspired by approaches from the STM literature. Often the inspired policies look quite different from the original STM ones because there is little overlap in the information available to BE-HTM and STM systems. However we attempted to preserve the fundamental insights of each particular policy, as best as possible, when translating to BE-HTM systems.

For those policies that can be tuned through a parameter, this chapter also contains the results of that tuning. A more detailed investigation and across-policy comparisons are presented in Chapter 8.

5.1 Max-Retry

Max-Retry is perhaps the simplest serialization manager imaginable. Max-Retry allows a fixed number of rollbacks before forcing a transaction to serialize. This policy trivially ensures forward progress and accepts different tunings for different programs.

Most previous work in BE-HTM systems have used policies very similar or identical to Max-Retry.

5.2 SerializationControl

SerializationControl is built upon Max-Retry, but encourages fast serialization for transactions that are predicted to have little chance of committing speculatively.

Each thread has a local *serialization table*[1] to support SerializationControl. For each static transaction, this table keeps track of the number of times that the transaction is committed, the number of times it is serialized, and a black-list counter that flags transactions for immediate serialization on retry.

The *serialization-commit ratio* for a static transaction is the ratio between the number of times the transaction is serialized to the number of times the transaction is committed. When a transaction serializes by retrying more than N times, the transaction entry in the serialization table is used to compute its serialization-commit ratio. If the serialization-commit ratio exceeds a predefined threshold, then the black-list counter in the serialization table entry is set to N. In the event of a transaction failure when the black-list counter is greater than zero, the transaction is serialized immediately and the black-list counter is decremented. Serializations that happen while a transaction is black listed do not contribute to the serialization count, thus allowing the serialization-commit ratio to decrease while the transaction is black listed.

The goal of SerializationControl is to improve performance in the worst cases for non-speculative execution when feedback is limited. This case happens when transactions continually fail when executed speculatively. Recording the history of static transactions per-thread leads to a better predictor of whether instances of the transaction will abort in the future. The per-transaction predictor should help in benchmarks where a subset of the static transactions require serialization more often than others.

5.3 Limit, inspired by Karma

Karma is a classic contention manager from the STM literature [64]. The goal of Karma is to increase throughput by aborting the transaction that has done the least amount of work. Karma estimates the amount of work done by a transaction by the number of objects that the transaction has accessed. The intuition behind Karma is

[1]The serialization table is keyed by static transaction address in the program text and uses linear probing to attempt to resolve collisions. However, if collisions are not resolved after two probes, then the element is recycled, leading to table aliasing.

that it is preferable to abort a smaller transaction that has done little work than to abort a larger transaction that could soon commit and has done larger amounts of work. In order to be fair, however, a dynamic transaction accumulates its completed work — called *karma* — across aborts to provide an element of *karmic fairness*. The idea is that small transactions should be allowed to build up enough karma so that they eventually complete even when they conflict with larger transactions.

Karma requires an estimation of the amount of work done by a transaction. This estimate is provided by the transaction read-write set size so that Karma can favour transactions that have done more work. A serialization manager inspired by Karma will need an alternative way to estimate the transaction size because existing HTMs do not expose the size of the read or write sets to software.

One solution is to budget an amount of execution time (in cycles) for each dynamic transaction to attempt to execute speculatively. We call this policy Limit. In Limit each dynamic transaction is given a budget of time to complete. While that budget is not exhausted, an aborting transaction may continue to retry. Once the budget is exhausted, the transaction must execute non-speculatively.

When Limit is run with exponential randomized backoff on retry, it could be said that Limit emulates the intentions of the contention manager Polka, which is precisely Karma with exponential backoff [64].

The intuition behind Limit is that it would be better to allow a short-running transaction to retry more often than to allow a long-running transaction to keep retrying. The idea is that both the risk of abortion and the amount of wasted work in case of abortion are lower for a short-running transaction than for a long-running transaction. In Limit a long-running transaction will serialize after fewer retries and thus will be favoured for resource allocation, just like in Karma.

5.4 LimitMean and LimitMeanST

Limit performs well when the budget parameter is tuned correctly. However, budget tuning is difficult because it depends on many factors. We have developed two modifications to Limit to attempt to address the tuning difficulty.

The first is called LimitMean. LimitMean computes, per thread, the cumulative moving arithmetic mean of the transaction execution time $\bar{\mu}_{\text{exec}}$, computed as:

$$\overline{\mu_{\text{exec}:(N+1)}} = \frac{N\overline{\mu_{\text{exec}:(N)}} + T_{\text{exec}}}{N+1} \tag{5.1}$$

The mean execution time is updated each time a transaction completes either serially or with no aborts. At transaction start, the budget is computed as $M \times \bar{\mu}_{\text{exec}}$ for some parameter M provided at program start.

LimitMean follows similar intuition to Limit, preferring to allow small transactions to retry and large transactions to serialize quickly. However, LimitMean adjusts the definition of small and large depending on the actual transaction lengths observed in the program, effectively self-tuning the budget.

LimitMeanST (Static Transaction) modifies LimitMean by tracking $\bar{\mu}_{\text{exec}}$ at a per-static-transaction level using the same hash-table scheme as SerializationControl.

The intuition behind LimitMeanST is a little different, and relies on the fact, seen in Figure 4.4, that transactions have varying running lengths. LimitMeanST is based on the idea that the serialization manager should not unduly punish naturally large transactions, but rather it should punish only the *outliers* of transactional execution time, *i.e.* only the transactions that run many times longer than average.

In both LimitMean and LimitMeanST transactions are pre-loaded with an estimate of the execution length of 10000 cycles if the transaction has not been seen previously, or if the hash table entry has been recycled.

5.5 Best-Effort Adaptive Transactional Scheduling (BE-ATS)

Adaptive Transactional Scheduling (ATS) is an idea from the STM literature [79]. ATS is not a contention manager, but instead it is a transactional scheduler that can be combined with any contention manager. A transaction scheduler adjusts when transactions execute, as opposed to a contention manager that decides what transactions survive a conflict. The motivation for including ATS in this study is to show how other techniques, besides contention managers, can be deployed as serialization managers on a Best-Effort HTM system, broadening the pool of ideas that can be drawn upon.

In a transactional-memory system experiencing low levels of conflict, retrying an aborted transaction speculatively — instead of acquiring a lock to ensure its completion — is a high-performance choice. However, when the contention levels are high, the amount of work wasted by frequent retries may outweigh the benefits of parallel execution. ATS aims to prevent the concurrent execution of transactions that have a high probability of conflict, without completely serializing execution.

The intuition behind ATS is that threads can detect high-contention situations and those threads experiencing contention can cooperate by waiting in a queue before executing a transaction. ATS uses Conflict Intensity (CI) — a parameterizable distributed measurement of contention — to decide whether transactions executing on a thread must consult a central queue before starting.

Conflict Intensity is recurrently defined as shown in equation 5.2. The value of CC is 0 for a commit and 1 for an abort, and $CI_0 = 0$. The parameter α runs from 0 to 1 and controls the amount of history preserved, and thus the rate of adaptation. A high α value preserves more history and slows adaptation to changing circumstance.

$$CI_n = \alpha \times CI_{n-1} + (1 - \alpha) \times CC \tag{5.2}$$

When a transaction starts (or restarts), its conflict intensity is compared against a `QueueIntensity` threshold. If the `QueueIntensity` threshold is exceeded, the transaction is stalled and enqueued in a global scheduler. When the transaction reaches the head of the queue, it is allowed to proceed regardless of its conflict intensity. ATS attempts to reduce contention by preventing concurrent execution of transactions that have a high probability of generating a conflict.

ATS can be implemented on BG/Q very simply and efficiently by tracking the conflict intensity[2] and acquiring a ticket lock when the conflict intensity goes above the `QueueIntensity` threshold. The ticket lock handles waiting threads using a first-in-first-out order, and thus emulates a queue.

[2]Tracking conflict intensity is slightly different in BG/Q. The compiler liveness analysis for register-save and restore expects the runtime does not use any floating point registers, and so the FP registers are not saved or restored. Therefore CI is computed using `libfixmath`, a Q16.16 fixed point library.

ATS alone does not guarantee forward progress. Even if all threads are in the queue for the lock, a livelock may occur if the executing thread is attempting to execute a transaction that requires irrevocability to complete. For instance, the executing transaction may perform an I/O operation.

To compensate for transactions that will always fail, we modified ATS into Best-Effort Adaptive Transactional Scheduling, or BE-ATS that defines a `MaxIntensity` threshold. If the CI computed for the thread is above `MaxIntensity` any transactions that execute on that thread are executed through non-speculative serialization, turning ATS into a serialization manager.

5.6 Percentage Of Effective Work (PEW)

A common hypothesis in TM systems is that it is best to minimize wasted time through some bound.

One measure of wasted time is given by the metric Percentage of Effective Work (PEW), described in a forthcoming contribution by Pereira *et al.* [59].

PEW is computed per static transaction t and, in our implementation, is computed thread-locally. In addition, PEW is specified based on an *execution slice*, a group of k attempts to execute a transaction t.

To compute $PEW(t, n)$ for a transaction t in slice n, five quantities are tracked:

1. $W_C(t, n)$ is the amount of work committed in slice n for transaction t.

2. $W_A(t, n)$ is the amount of work wasted in a slice n for transaction t.

3. $c(t)$ is a counter that tracks progress of an execution slice for transaction t.

4. $T(t, n)$ is the total work performed by the transaction, including wasted work. This value is smoothed by the history of previous slices.

5. $E(t, n)$ is the effective (committed) work performed by the transaction in slice N. This value, like total work, is smoothed by history.

Whenever $c(t) = k$ the runtime recomputes $PEW(t, n)$. First the runtime computes $W_T(t, n) = W_C(t, n) + W_A(t, n)$. $W_T(t, n)$ is the total amount of execution time for transaction t. From these two values, the total (Equation 5.3) and effective (Equation 5.4) work can be computed, with a history parameter α to provide some discounting to past values.

$$T(t,n) = \alpha \times T(t,n-1) + (1-\alpha) \times W_T(t,n) \tag{5.3}$$

$$E(t,n) = \alpha \times E(t,n-1) + (1-\alpha) \times W_C(t,n) \tag{5.4}$$

The percentage of effective work is defined as the ratio between these two values (Equation 5.5):

$$PEW(t,n) = \frac{E(t,n)}{T(t,n)} \tag{5.5}$$

On rollback, if $PEW(t,n)$ drops beneath some threshold T, then the runtime will serialize that execution, increasing $PEW(t,n+1)$.

As in BEATS, PEW was implemented using a fixed-point math library to perform the fractional calculations.

5.7 Other Investigated Serialization Managers

Though this thesis focuses on the seven serialization managers described above, they are far from the only possible serialization managers, and far from all that we investigated.

Table 5.1 contains a short discussion of a number of policies that were also investigated and rejected, sorted by the reason for their rejection. **Runtime Work** contains policies that are feasible candidates for BE-HTMs but would require more engineering resources than were available. **Insufficient Information** contains policies that do not successfully translate to BE-HTMs due to the limited feedback provided by HTMs. **Insufficient Control** are polices that require more control than an HTM provides.

There were also some policies that were implemented but that either overlapped substantially with existing policies or did not perform well enough to be worth describing in detail.

Runtime Work	
adaptSTM [58]	Could be used for switching between long– and short-running modes, however would require a redesign of the TM runtime.
Insufficient Information	
CAR-STM [28]	Would require implementing higher-order transactions that can be manipulated; moved from one core to another core for example. Would also require information on enemy transactions.
LUTS [56]	Need enemy-transaction information to model conflicts
Eruption [43]	Need enemy-transaction information to pass along priority.
Steal-on-abort [4]	Need enemy-transaction information to choose target thread, as well as requiring higher-order transactions that can be moved from core to core.
Insufficient Control	
Various contention managers [64]	Very weak control of transactions to abort on conflict

Table 5.1: Dismissed potential policies, grouped by reason for rejection

Chapter 6

Manager-Independent Policies

While trying to understand how serialization management fits together with existing HTM systems, the question was raised as to whether there are manager-independent policies that can be added to any serialization manager. In our exploration we discovered two such policies, Rollback Delay and Capacity Serialization, described below.

In the experiments in this chapter we use the serialization manager Max-Retry as a vehicle to explore the two policies and their effect on performance. The results in this chapter will justify enabling these two policies in all subsequent experiments. Section 7.1 will discuss Max-Retry itself.

6.1 Rollback Delay

Bobba *et al.* described a pathology called Convoy, where a large group of transactions abort near-simultaneously, usually because of conflicting references on the same cache line, and retry near-simultaneously, leading to waves of rollbacks and little progress or livelock [11]. To avoid a Convoy, restart after rollback can be delayed through a randomized exponential backoff [73].

Figure 6.1 shows the results of a comprehensive exploration of exponential backoff's effect on performance on the STAMP benchmarks. The horizontal axis records the number of rollbacks that were allowed and the vertical axis reports the absolute speedup over an efficient sequential execution reported for each individual execution of the benchmark as a single point in the plots. For each allowed retry value, from

1 to 39, each benchmark was executed five times. The number of threads used in each execution is shown at the top of each column of plots. The graph shows the mean of the measured speedups, along with a 95% confidence interval in grey. For instance, the plots for `kmeans` with 16, 32, and 64 threads show a higher variance in the measured speedups in comparison with all other plots. The red line represents the mean speedup without rollback delay while the blue dotted line represents the mean speedup with rollback delay.

Figure 6.1 shows that most benchmarks see performance improvements from enabling exponential random backoffs. The improvements typically are seen at higher thread counts combined with higher numbers of allowed-retries. The result can be dramatic, doubling the performance of `yada` in Long Running mode and improving the performance of `intruder` especially when a large number of retries is allowed.

However, the improvement is not universal. Some configurations see higher performance without rollback delays. This is the case, for example, when running `intruder` with two and four threads. Rollback delay can reduce performance where simultaneous retry is uncommon, or where simultaneous retry does not cause more aborts.

Degradations caused by exponential backoff are rare and of low magnitude compared to the improvements delivered by exponential backoff. Therefore all subsequent experiments in this part will have exponential backoff enabled.

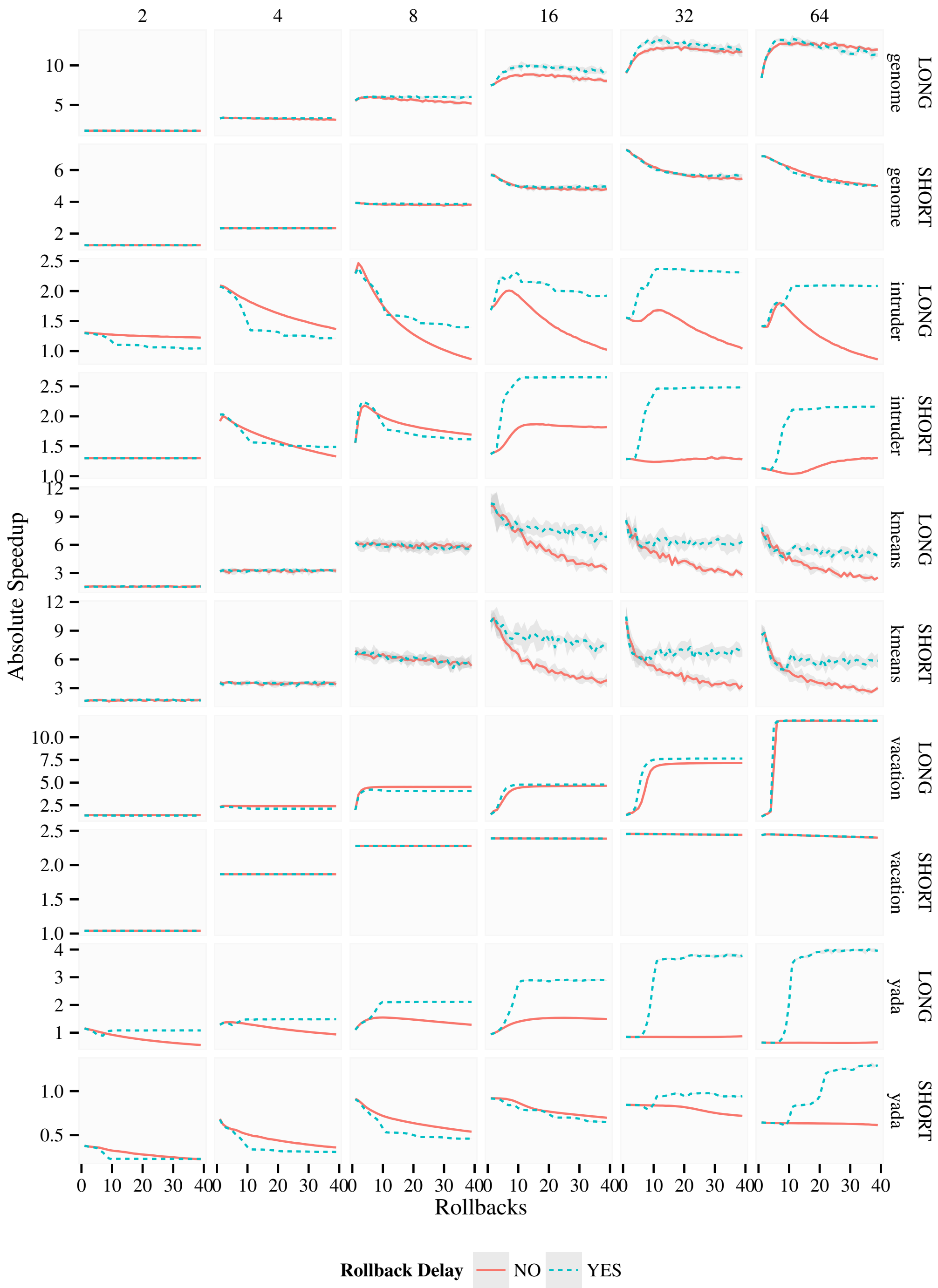

Figure 6.1: The relationship between allowed-rollbacks and absolute speedup, showing the effect of exponential backoff. Using MAX-RETRY and no capacity serialization.

6.2 Capacity Serialization

The BG/Q TM hardware provides some feedback to the TM runtime system. One element of the feedback is a capacity bit that indicates that the transaction was aborted because a speculatively loaded line was evicted, meaning that conflicts could no longer be detected safely. When designing a policy the capacity bit can be used to serialize early, ideally saving one or more useless subsequent retries.

However, the capacity bit can be misleading. In both long- and short-running mode, when a transaction is retried, other parallel threads and transactions may have changed the state of the cache or the system – other transactions may have committed, freeing speculative state for use, for example – possibly allowing the capacity-aborted transaction to complete speculatively on retry. In addition, as mentioned in Section 2.3.1, in short-running mode there is some aliasing on the capacity bit, where the hardware sets the capacity bit even on aborts that are not actually caused by capacity limitations.

Figures 6.2 and 6.3 show the results of a comprehensive exploration of the effects of capacity-induced serialization on performance on the STAMP and RMS-TM benchmarks.

Each point in the figure corresponds to a single execution. Similar to Figure 6.1 the plots here also show the mean and 95% confidence interval of the measured absolute speedup of five runs of the benchmark for each number of allowed rollbacks. The red line show the speedup for runs without capacity-induced serialization while the blue dotted-line is for runs with capacity serialization. All the plots in a column are for the same thread count, listed at the top of the column, and each row of plots is for a given benchmark at a given running mode as indicated to the right of the figure.

The results in Figures 6.2 and 6.3 indicate very small effects on performance from capacity-induced serialization for most benchmarks in either running mode. The largest effects are improvements due to capacity-induced serialization in short-running mode with large amounts of retries allowed — seen in `genome`, `intruder` and `hmmcalibrate`. The effect of capacity-induced serialization on performance in long-running mode seems to be almost entirely negligible, with a very small amount of change noticeable for `intruder` running with 16 threads.

The different responses to capacity-induced serialization in the two modes are explained partially by Table 3.2 that shows the percentage of transactions that report a capacity overflow in each benchmark.

All subsequent experiments in this thesis will have capacity-induced serialization enabled because capacity-induced serialization seems to provide more benefit than harm.

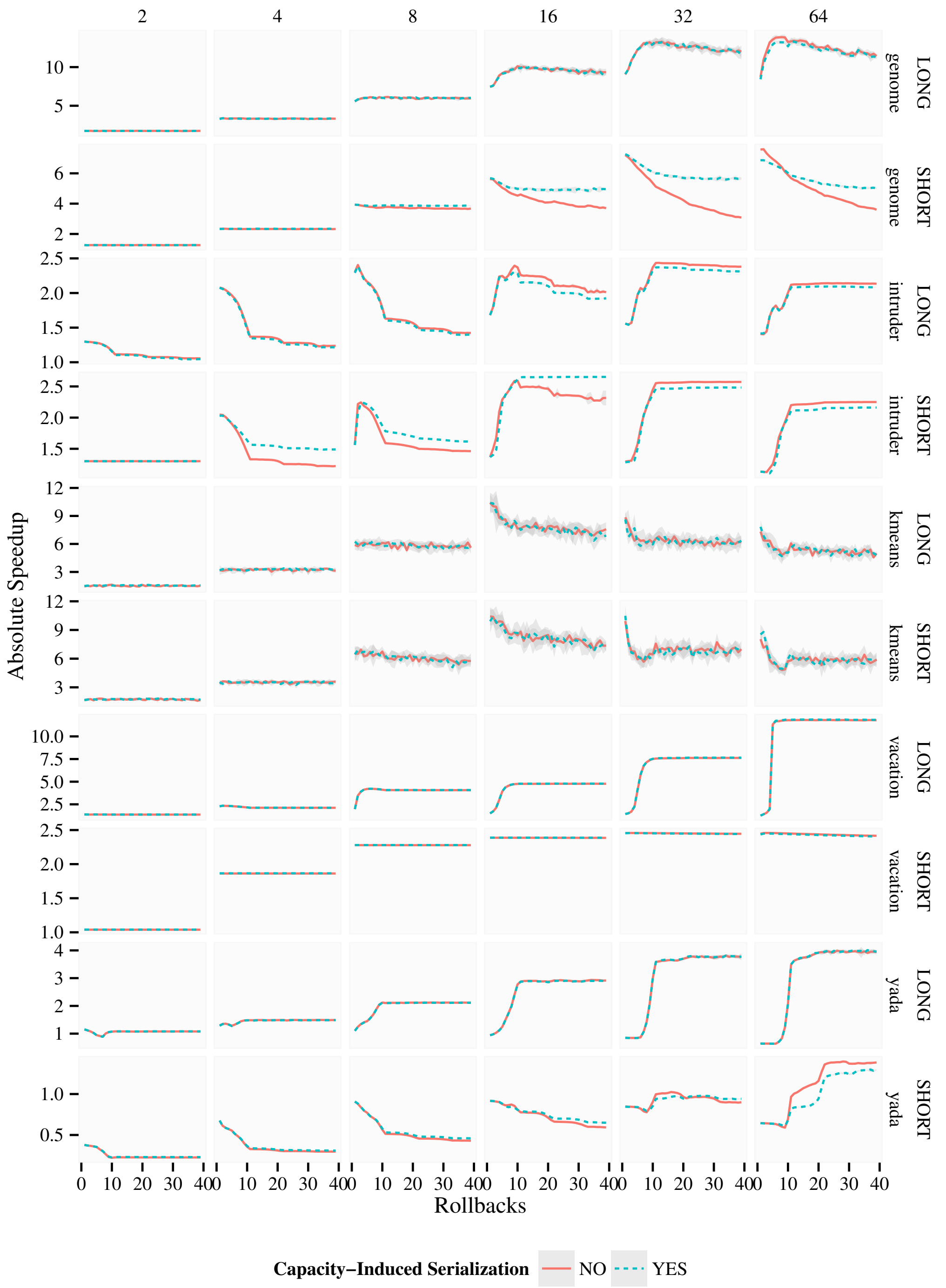

Figure 6.2: The relationship between allowed-rollbacks and absolute speedup, showing the effect of capacity-induced serialization on the STAMP benchmarks. Using MaxRetry and exponential-backoff.

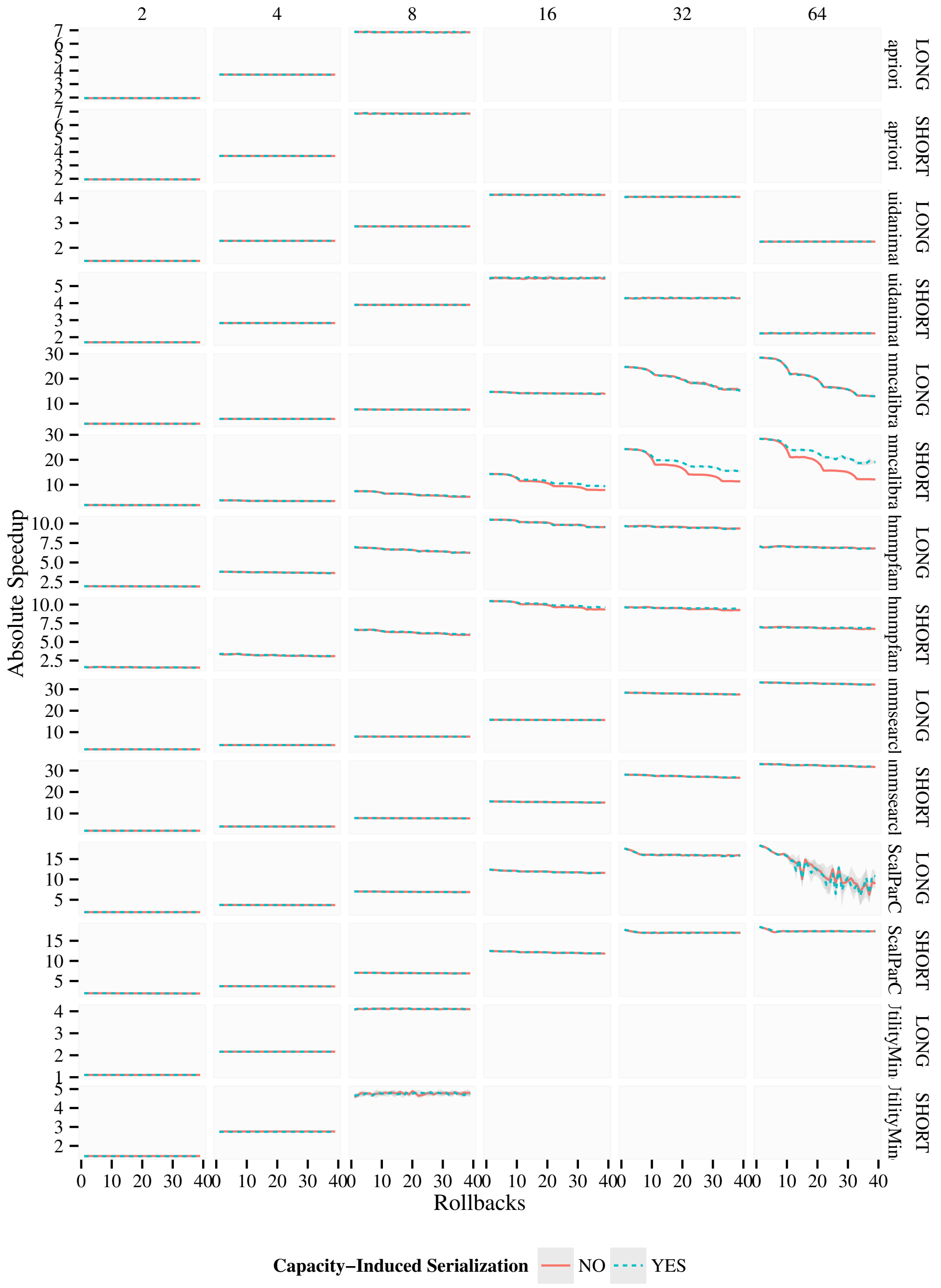

Figure 6.3: The relationship between allowed-rollbacks and absolute speedup, showing the effect of capacity-induced serialization on the RMS-TM benchmarks. Using MAX-RETRY and exponential-backoff. `apriori`, `fluidanimate` and `utilitymine` are excluded for space reasons, as well as showing no effect.

Chapter 7

Serialization Manager Tunings

All the serialization managers described in Chapter 5 have at least one tuning variable that can be changed to vary the behaviour of the manager. A generous evaluation of serialization managers requires understanding the behaviour of the manager across a wide range of its parameter values. This chapter performs that study for each serialization manager described in Chapter 5.

7.1 Tuning Max-Retry

Max-Retry requires that all transactions complete successfully within a limited number of retries. If a transaction exceeds that limited number of retries, Max-Retry serializes execution in order to enforce forward progress.

Figures 7.1 and 7.2 show how the absolute speedup achieved by the STAMP and RMS-TM benchmarks changes with both the running mode and the number of allowed retries when using Max-Retry as the serialization manager.

The two figures show clearly that tuning for the number of rollbacks allowed has some effect. However, the number of rollbacks for which the maximum speedup is achieved depends not only on the running-mode but also on the benchmark and the number of threads used for execution.

The STAMP benchmarks spend substantially more time inside transactions than the RMS-TM benchmarks. Thus, the STAMP benchmarks are much more sensitive to tuning as compared to the RMS-TM benchmarks, where only `hmmcalibrate`, `hmmpfam` and `scalparc` appear to be affected by tuning.

Where the RMS-TM benchmarks appear to universally perform best with only one retry leading to serialization, the STAMP benchmarks vary dramatically on where the best speedup is achieved. For example, `vacation`, in long-running mode performs best with a large number of retries allowed (> 10), yet for `kmeans`, any more than one or two retries reduces performance for all thread counts more than 4. `intruder` is notable for a dramatic reversal of behaviour between thread counts ≤ 8 and thread counts ≥ 16.

No benchmark in our tests improved in performance after 25 retries.

7.1.1 Mode Generalizability

According to the results for the the MAX-RETRY tuning, the benchmarks can be divided into two groups. In one group — consisting of `intruder` , `kmeans` (both low and high contention) and all the RMS-TM benchmarks excluding `scalparc` run with 64 threads — the tuning between long and short running mode largely follow the same general trend. Most of these benchmarks are mode-insensitive, seeing relatively small differences between achievable speedup between long- and short-running mode. The tuning for the second group — consisting of `scalparc` run with 64 threads,and the remaining STAMP benchmarks — follow very different trends between long- and short-running mode. `scalparc` is the exception, as it is largely mode-insensitive, yet sees a dramatic change in program behaviour for long-running mode when more than ten rollbacks are allowed in comparison to short-running mode.

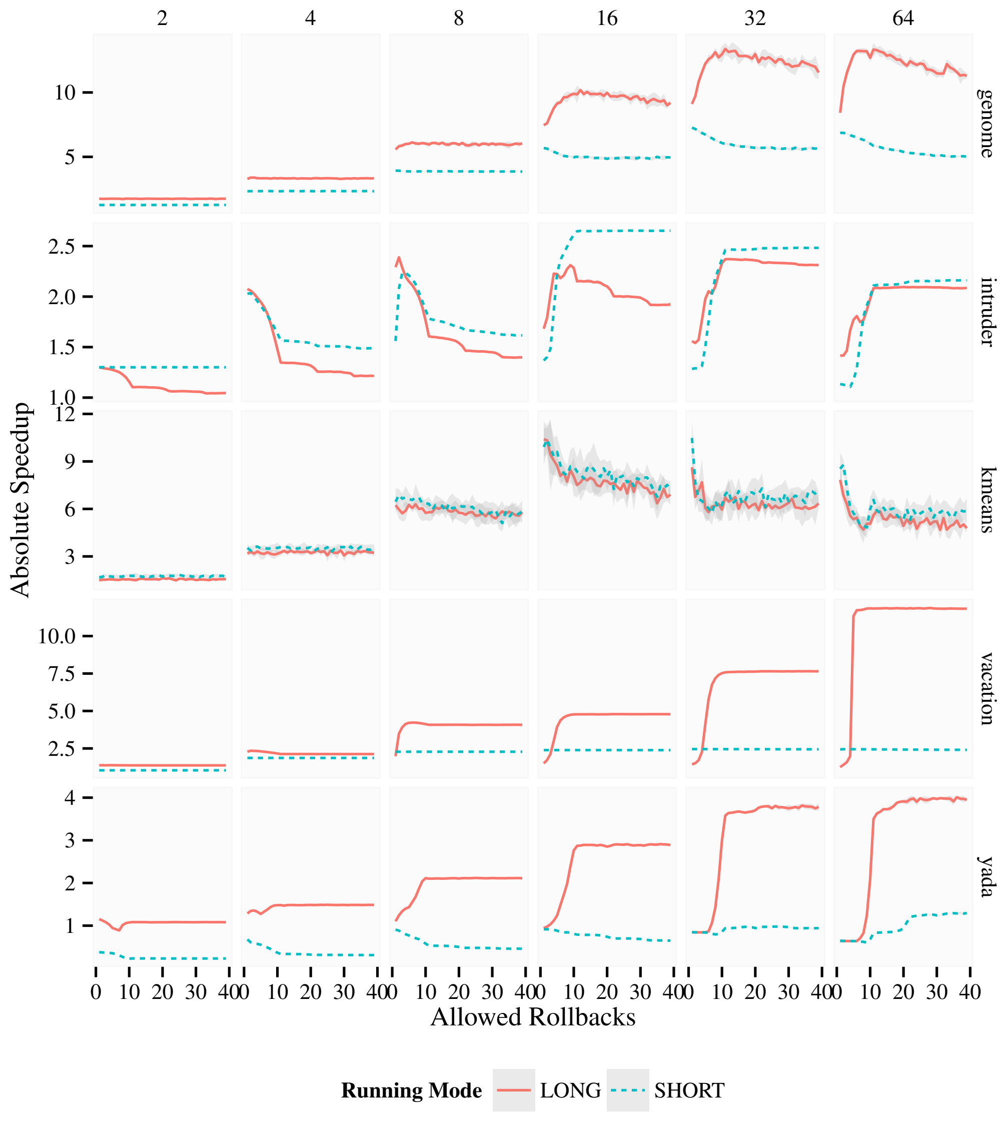

Figure 7.1: Absolute Speedup for STAMP relative to the number of allowed-rollbacks in Max-Retry for both short and long-running mode. vacation-high and kmeans-high are elided because their behaviour is indistinguishable from their low contention counterparts.

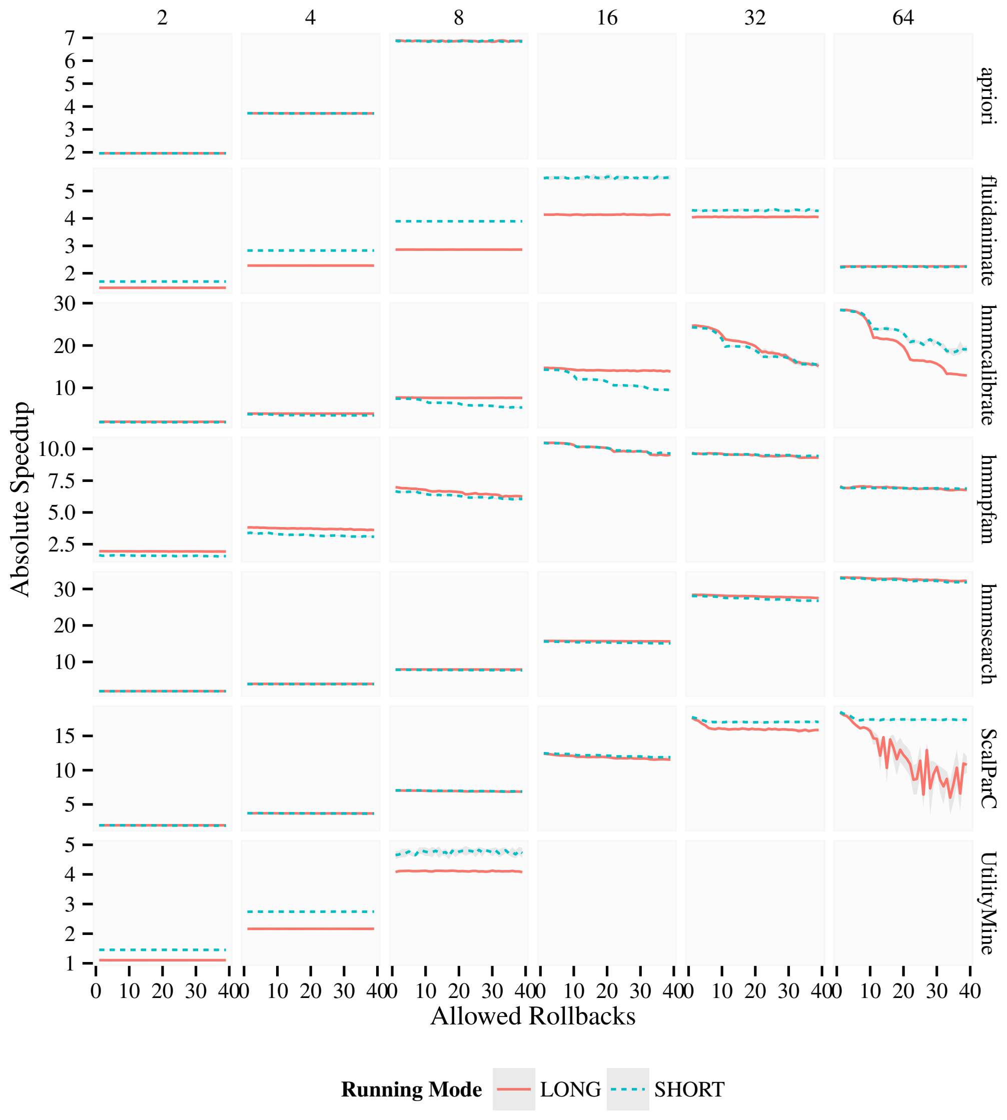

Figure 7.2: Absolute Speedup for RMS-TM relative to the number of allowed-rollbacks in Max-Retry for both short and long-running mode.

7.2 Tuning SerializationControl's Blacklisting Threshold

SerializationControl modified Max-Retry to handle transactions that serialize often by blacklisting them, forcing them to serialize immediately on subsequent aborts for a period of time.

In the vast majority of benchmarks — all of the RMS-TM benchmarks, `genome`, `ssca2` — SerializationControl does not change performance relative to Max-Retry across all the values of threshold explored. In the case of the RMS-TM benchmarks, this is due to the relatively low amount of time spent in transactions. In the case of `genome` and `ssca2` the lack of effect of SerializationControl on performance is because rare serializations never trigger blacklisting.

What is the effect of the blacklisting threshold of SerializationControl on performance? The results of an experiment that varies the blacklisting threshold from 0.05 to 1 — for 5, 10 and 40 allowed retries — for the benchmarks whose performance is affected by SerializationControl is presented in Figure 7.3. Each graph presents the raw data-points of the results for five runs. In these plots, variance is seen through the width of the lines. The darkness of a dot indicates the black-list threshold according to the key at the bottom of the figure. A black-listing threshold of 1 indicates that the blacklist will only be activated once per static transaction, if it has failed 100% of the executions. This threshold value mimics the behaviour of Max-Retry in the vast majority of situations and provides a point of comparison for very little or no blacklisting.

In most of the benchmarks a lower threshold for black listing is too aggressive, causing a reduction in performance relative to higher thresholds. The exceptions are benchmarks with high amounts of contention and serialization: `yada` running with 8 threads and `kmeans` running with more than 16 threads where the number of allowed retries is low. These are also the configurations and benchmarks that performed best with a very-low number of retries in Max-Retry, as expected. `intruder` and `yada` show discontinuities where there exists a minimum (in the case of `intruder`) or maximum (in the case of `yada`) threshold requirement. In the majority of cases it appears that it is much more important to choose the correct number of retries than the correct black-listing threshold. Black listing has a relatively minor effect in those cases where it improves performance, except for `kmeans` that sees a substantial improvement from avoiding transactional execution as much as possible.

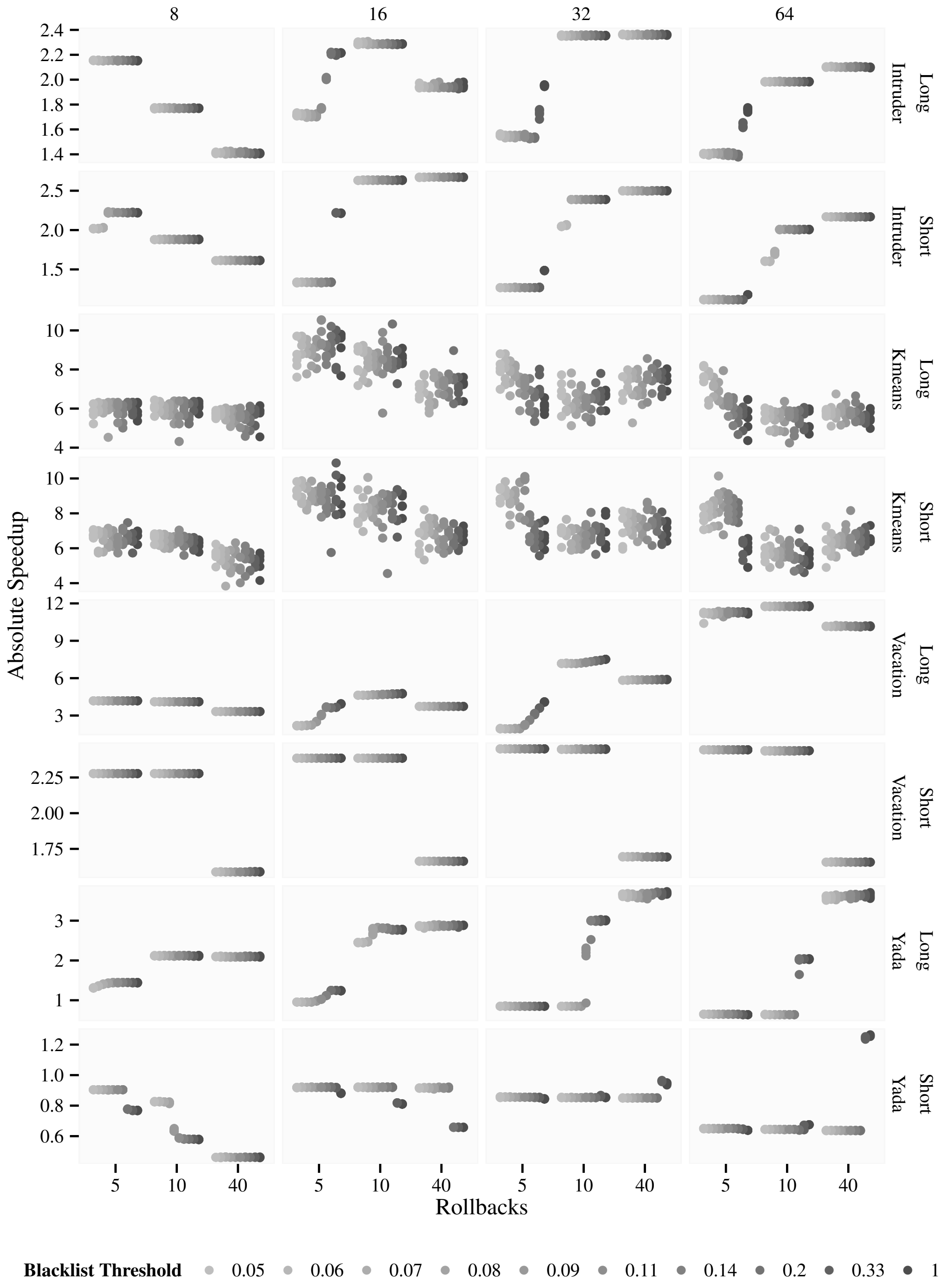

Figure 7.3: Absolute speedup of the benchmarks affected by SerializationControl relative to Threshold and 5, 10 and 40 allowed rollbacks in SerializationControl. A threshold of 1 approximates Max-Retry in almost all cases, only blacklisting transactions when the transaction has failed 100% of executions.

7.3 Tuning Limit's Per-Transaction Execution-Cycle Budget

Limit provides a static execution-cycle budget for each dynamic transaction as discussed in Section 5.3. This budget is the same for all transactions and is determined by the user at program start. The intuition is that a sufficient budget will encourage large transactions to serialize while allowing smaller ones to keep attempting speculative execution. The hope is that less computation will be wasted retrying larger transactions.

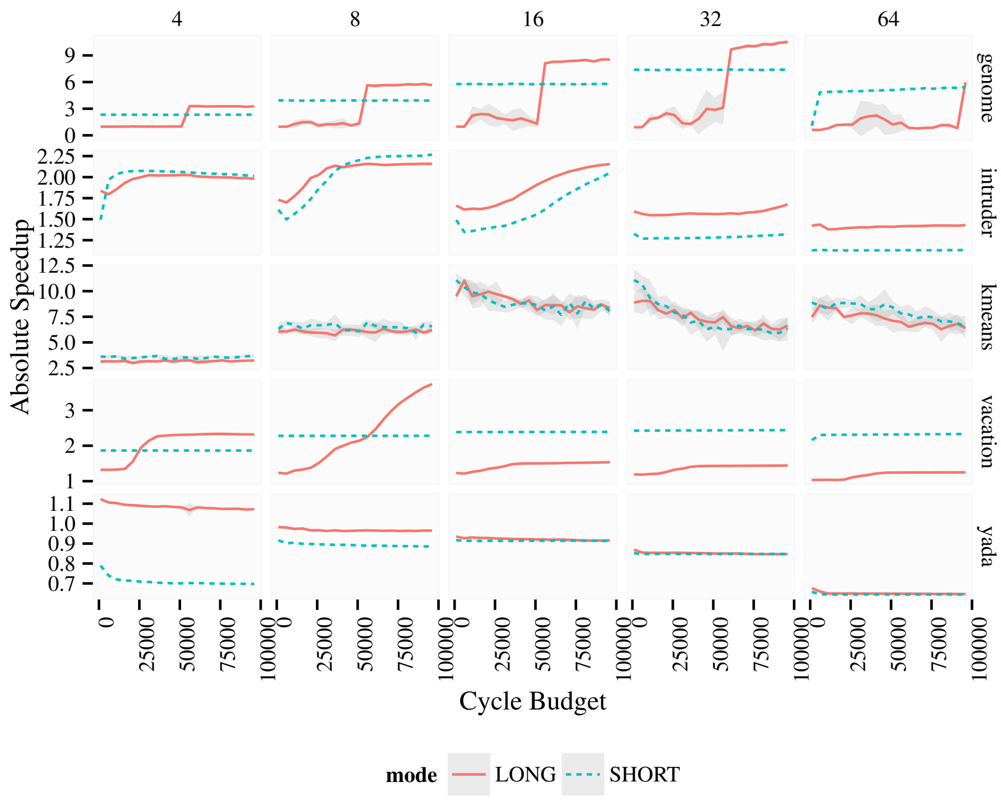

Figure 7.4: Absolute speedup of STAMP benchmarks relative to cycle budget for Limit. `kmeans-high` is excluded because it under-performs in all configurations, similar to `yada`

Figures 7.4 shows the effect of tuning the per-transaction cycle budget for the benchmarks from 1000 to 96000 budgeted cycles on the STAMP benchmark suite.

Results for the RMS-TM benchmarks are not presented because they were found to be entirely insensitive to the cycle budget except for a small amount of performance degradation with very low (< 3000) cycle budgets.

The results for `genome` show how sensitive Limit can be to tuning. `genome` has a sharp increase in speedup when the time budget passes a required minimum budget of 55,000 cycles. After the minimum budget is achieved, speedup continues to improve with a larger cycle budget.

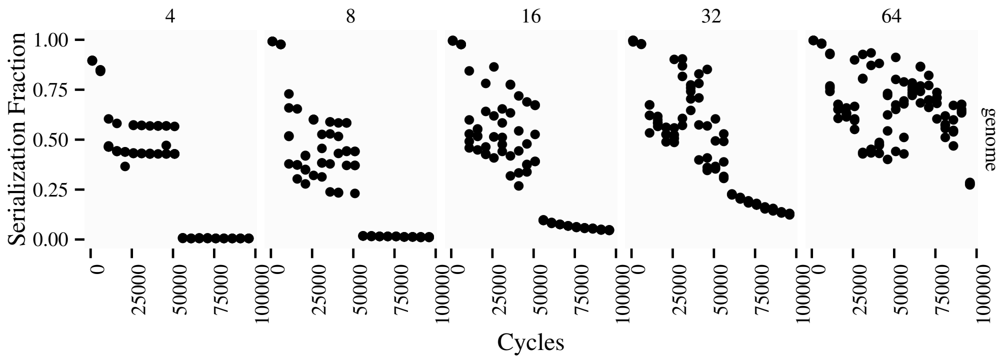

Figure 7.5: Serialization fraction for `genome` in long running mode, relative to the cycle budget in Limit showing the dramatic effect of achieving the minimum budget.

The required minimum budget for genome stays relatively constant for 4, 8, 16 and 32 threads. However at 64 threads the required minimum jumps higher, possibly due to increased pressure on shared resources in 4-way SMT mode.

The tuning curves for long- and short-running modes appear largely consistent for `kmeans` and intruder. The highest performance for `kmeans` occurs at the lowest budget because `kmeans` prefers serialization to speculation.

Figure 7.5 shows the dramatic effect the cycle budget has on the fraction of transactions serialized. The continued improvement shows up here as a decrease in serialization.

When compared to the previous serialization managers, all benchmarks, except for `kmeans`, perform far worse than the highest performance seen. The expectation here is that in most cases the range of explored cycle budgets was insufficient. The challenge of choosing the cycle budget correctly is what inspired LimitMean and LimitMeanST.

7.4 Tuning M in LimitMean and LimitMeanST

LimitMean and LimitMeanST attempt to alleviate the challenge of choosing the correct cycle budget by estimating it as some multiple M of $\bar{\mu}_{\text{exec}}$, the estimated mean execution time.

Figures 7.6 and 7.7 contain the results of a tuning study of M for LimitMean and LimitMeanST. In addition, the results of changing the allowed number of retries for Max-Retry have been included because the three serialization managers are so closely related that comparing allowed retries and M is very sensible.

For most tests the maximum speedup of the three managers is achieved by either LimitMeanST or Max-Retry, with the single exception of `yada` in short-running mode run with 32 threads. In all other cases LimitMean at best matches the performance of the other two policies.

`genome` is an interesting success case for LimitMeanST. Compared to Max-Retry, LimitMeanST's ability to distinguish based on transaction appears to dramatically improve performance over Max-Retry and LimitMean for many larger values of M. A similar benefit, at a different point in the tuning space appears to occur for intruder in long-running mode.

In contrast, `vacation` has a cliff in its performance with LimitMean and LimitMeanST that occurs consistently with more than ten retries. There is a small jump in serialization fraction that happens at that point, however, it is not clear why this occurs though it seems to be inherent in the application.

The key difference between Max-Retry, LimitMean and LimitMeanST is how they decide when to serialize. Figures 7.8 and 7.9 explore how the fraction of transactions serialized relates to absolute speedup through a scatter plot of all the samples obtained. The idea behind the figures is to show any relationship between the empirical serialization fraction and speedup.

Roughly speaking Figures 7.8 and 7.9 contain three groups of benchmarks (and configurations): those like `genome` and `intruder` (in long-running mode) that see improved results with increased serialization up to a limit, those like `kmeans` and `hmmcalibrate` where the performance keeps improving as the amount of serialization increases; and those like `yada` and `hmmsearch` that see only reductions in performance when serialization increases. These results are rooted in the algorithms and data access patterns of the benchmarks.

The notable feature of LimitMeanST is the ability to make different decisions based on what static transaction is being executed. `intruder` and `yada` appear to be the only two benchmarks that benefit from this ability as can be seen by improved performance over Max-Retry and LimitMean with the same serialization fraction.

The vertical bar for ScalParC executing with 64 threads in long-running mode corresponds to the decrease in performance when more retries are allowed in the Max-Retry policy as seen in Figure 7.7.

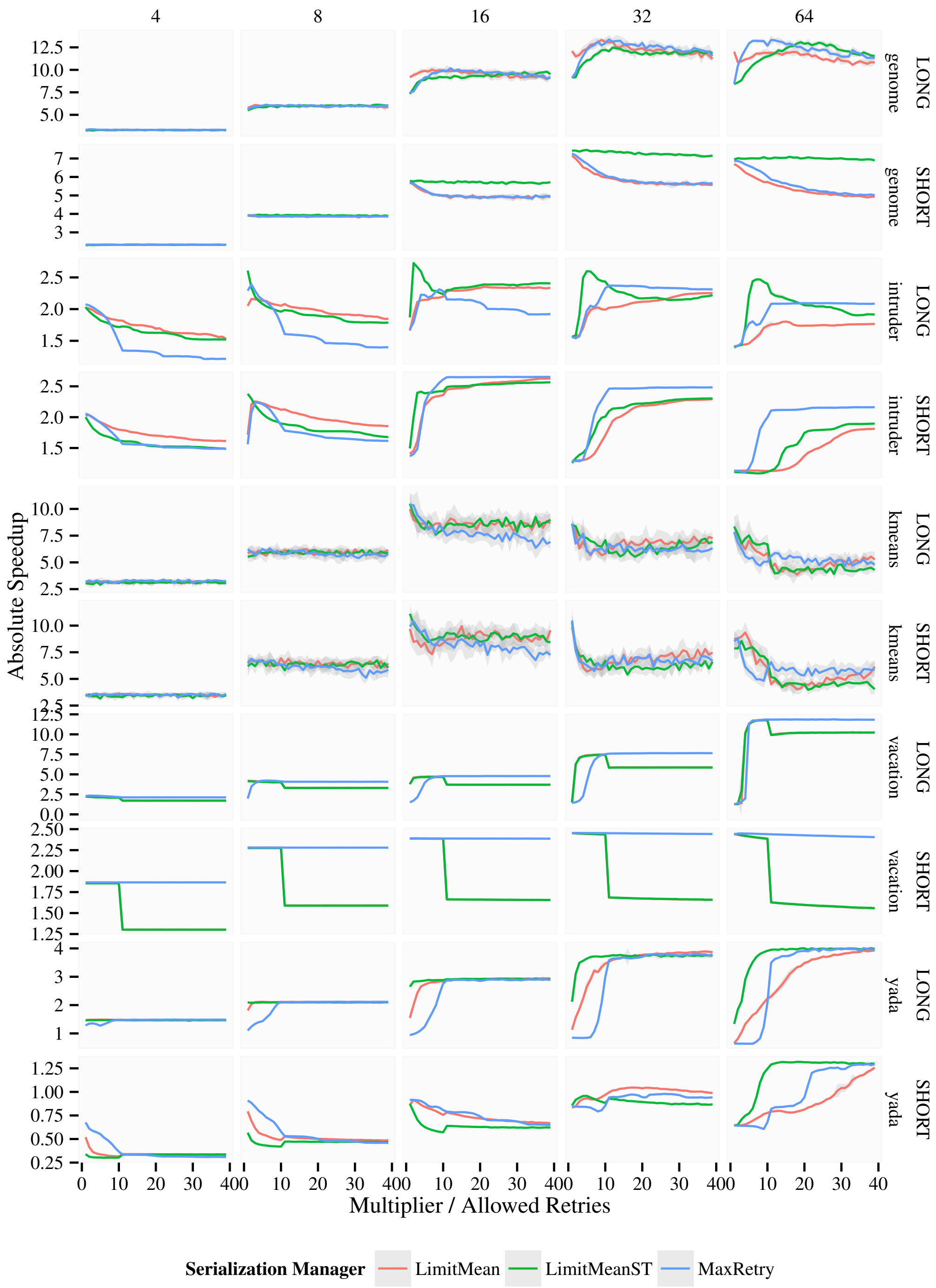

Figure 7.6: Absolute Speedup for the STAMP benchmarks relative to M or allowed retries when run with LIMIT, LIMITMEAN or LIMITMEANST. The results for 2 threads are elided because they don't vary much over the range values explored.

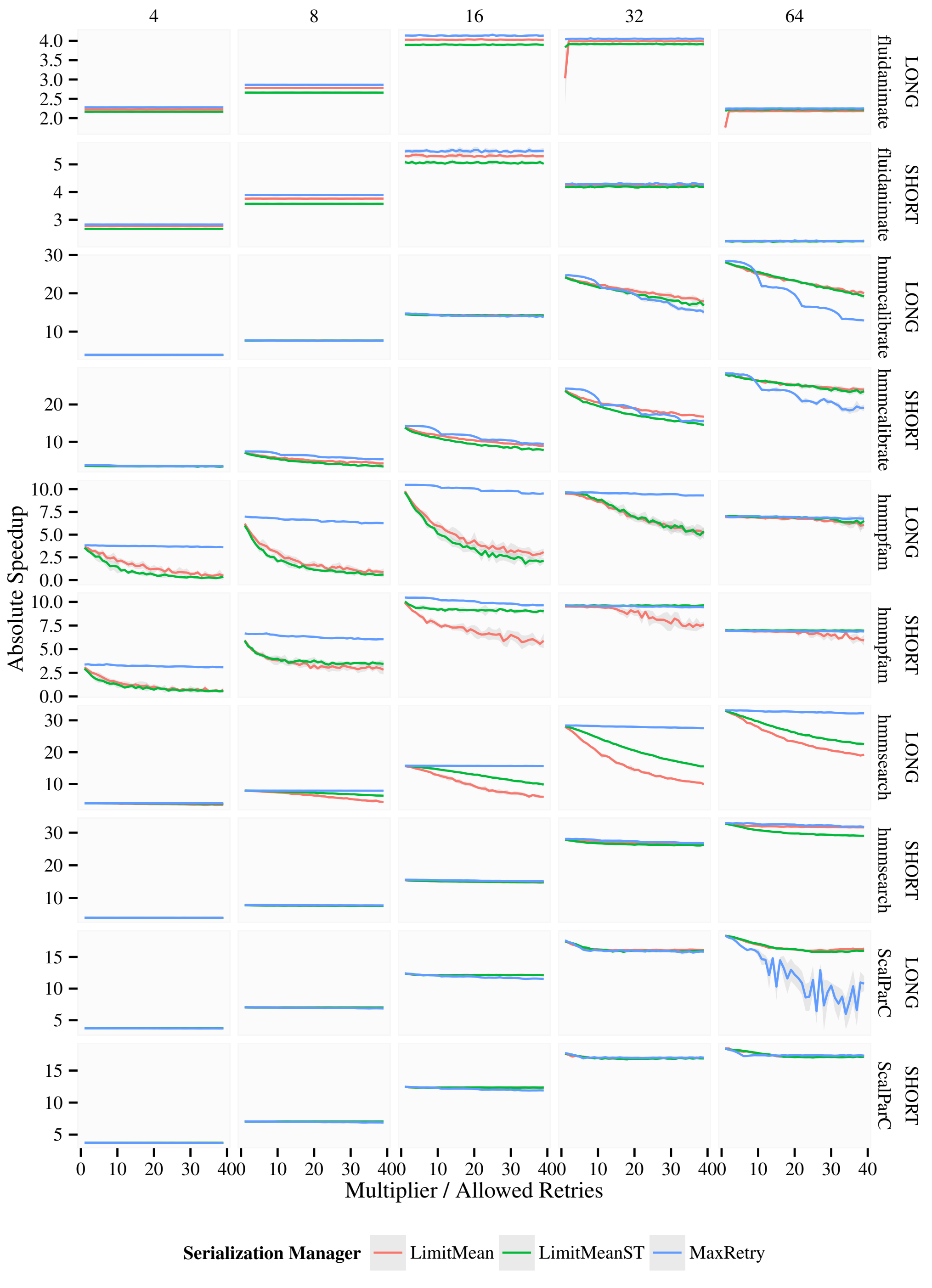

Figure 7.7: Absolute Speedup for the RMS-TM benchmarks relative to *M* or allowed retries when run with MAX-RETRY, LIMITMEAN or LIMITMEANST. The results for 2 threads are elided because they don't vary much over the range values explored. `apriori` and `utilitymine` are excluded because they don't see any change from policy or tuning.

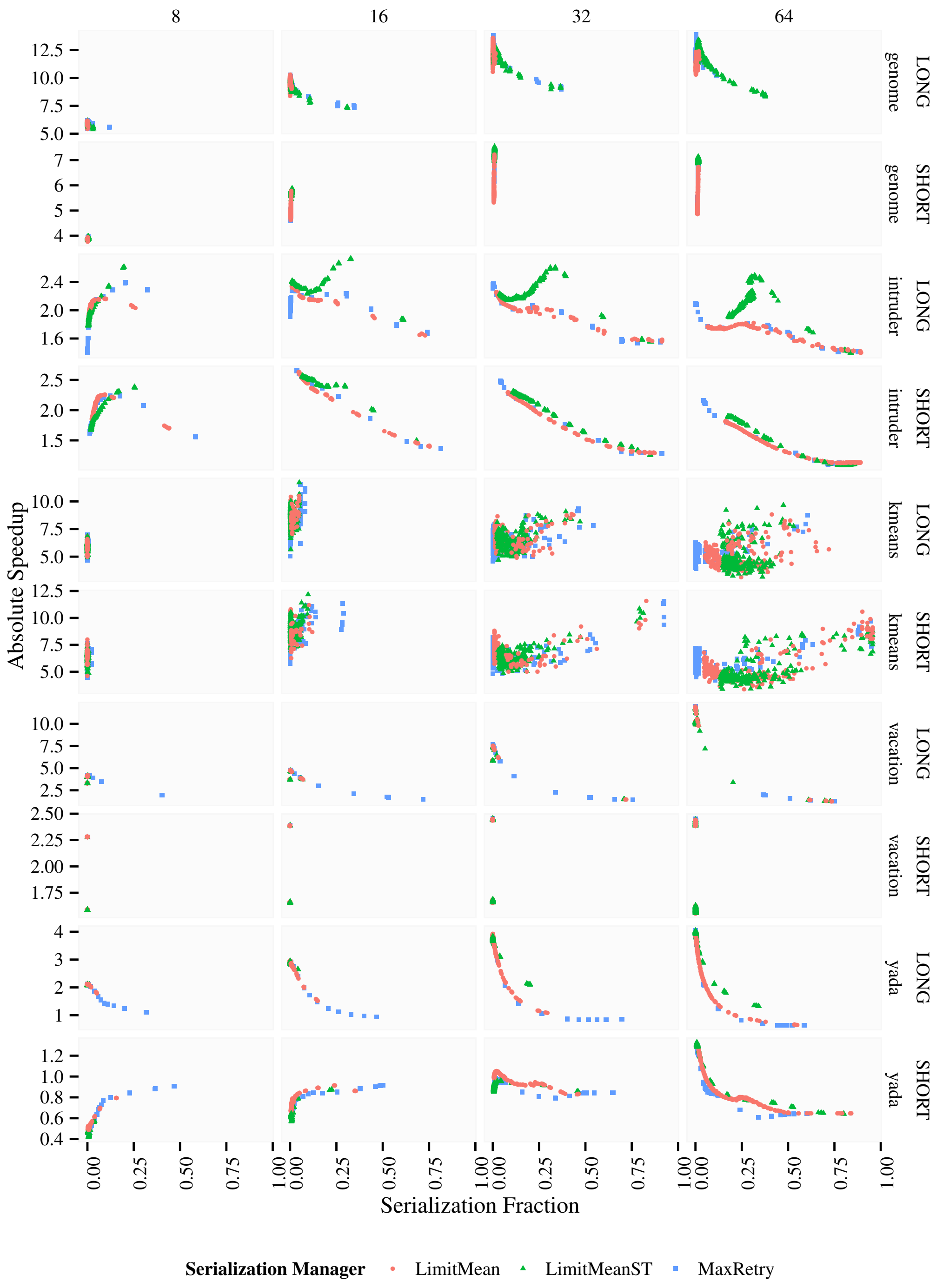

Figure 7.8: Absolute Speedup as related to serialization fraction for STAMP benchmarks run with the MAX-RETRY, LIMITMEAN and LIMITMEANST serialization managers with M or allowed retries between 1 and 39.

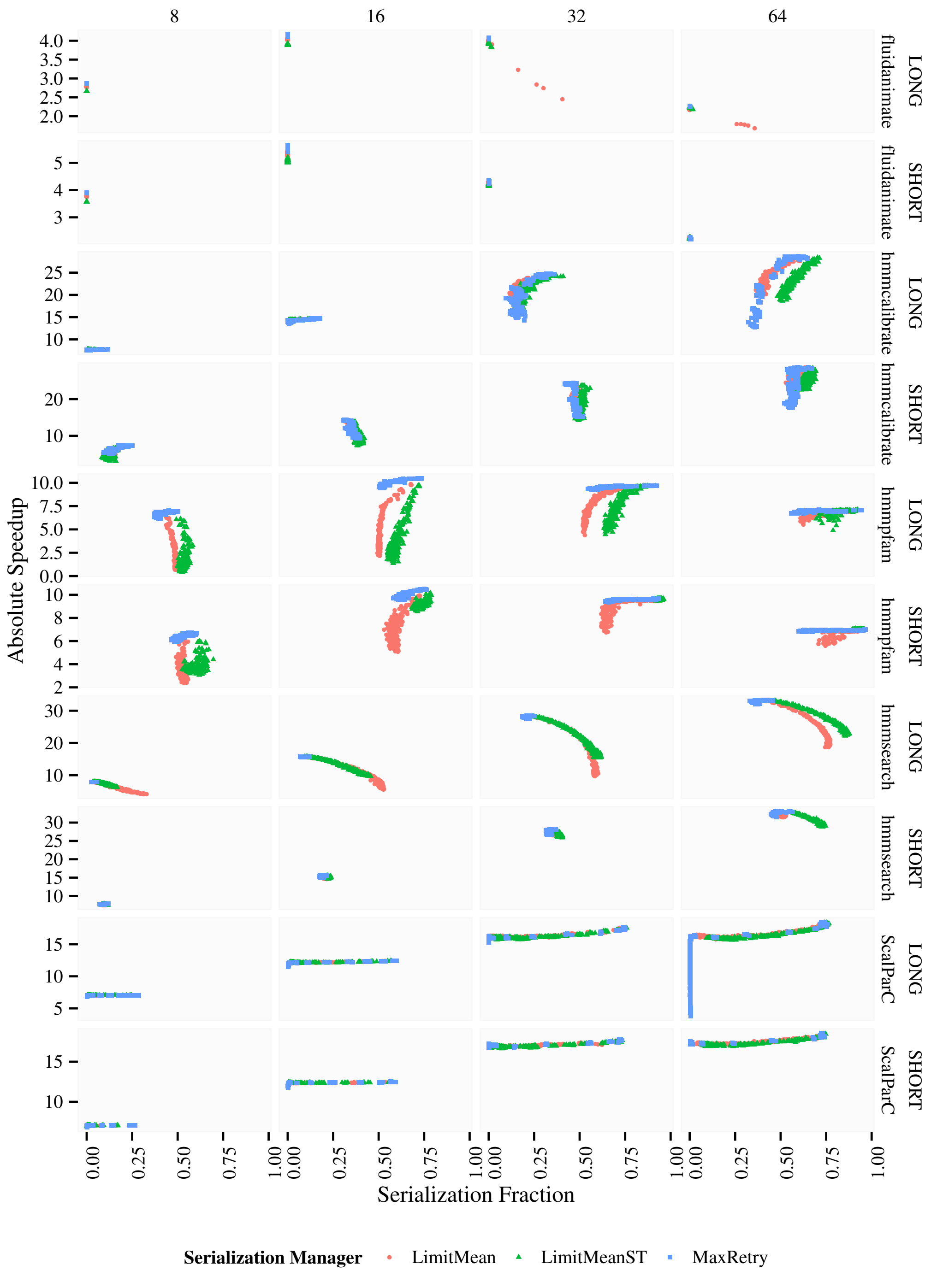

Figure 7.9: Absolute Speedup as related to serialization fraction for RMS-TM benchmarks run with the MAX-RETRY, LIMITMEAN and LIMITMEANST serialization managers with *M* or allowed retries between 1 and 39. `apriori` and `utilitymine` are elided from this figure because they see no relationship between serialization ratio and speedup.

7.5 Tuning α in BE-ATS

BE-ATS tracks contention per thread using Conflict Intensity, which is updated using a parameter α that controls the amount of history preserved. Higher α means slower adaptation to changing circumstance, and lower α means faster adaptation to changing circumstance.

When a thread's measured Conflict Intensity goes above a threshold `QueueIntensity`, subsequent transactions on that thread are queued to reduce overlapped execution. If a thread's Conflict Intensity increases beyond a second threshold, named `MaxIntensity`, then transactions are serialized upon aborting.

Figure 7.10 shows how benchmark performance is affected by varying α for both STAMP and RMS-TM. In this experimentation, `QueueIntensity` was arbitrarily set to 0.6 and `MaxIntensity` to 0.9. Though a complete examination of BE-ATS would require also examining where the thresholds are set, a static pair of thresholds gives a good overview of the range of behaviours expected modulo some shifting based on α.

Previous experimentation has showed that `kmeans` and `scalparc` perform best when serialization happens immediately after abort. This trend holds true with BE-ATS with the highest performance occurring when α is low, which ignores most of the previous value of Conflict Intensity. Similarly `hmmcalibrate`'s performance drops as α increases.

As we have seen previously, `intruder`, `kmeans` and the RMS-TM benchmarks have similar speedup curves between short- and long running-mode, while the remaining benchmarks see different behaviour curves across the values of α.

One of the fundamental elements of BE-ATS is the idea that threads can self-throttle in order to reduce contention and increase program performance. Figure 7.11 examines the relationship between the fraction of transactions that enter the queue and absolute speedup.

`intruder` in both modes and `genome`, `yada` and `vacation` in long-running mode appear to benefit from small amounts of queueing. The remaining benchmarks' performance is largely determined by serialization, with performance either entirely unrelated to the fraction of transactions queued or with their highest performance happening when no transactions are queued. While some benchmarks queue more than 20% of total executed transactions, typically large fractions of queued transactions appear to harm speedup overall, and few benchmarks queue many transactions.

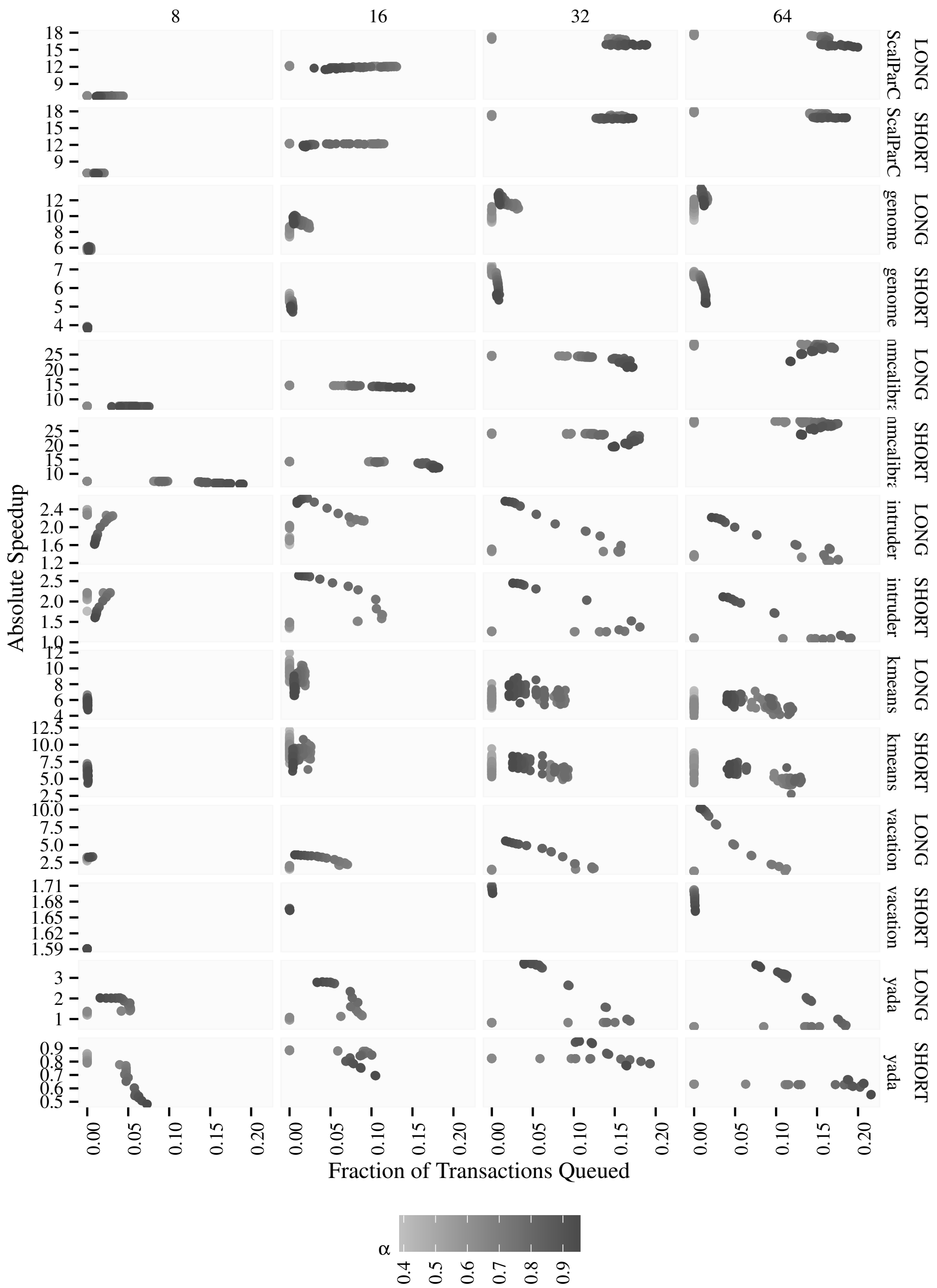

Figure 7.10: Absolute Speedup for STAMP and RMS-TM benchmarks relative to α parameter in BE-ATS. Results for thread counts less than 8 and the missing RMS-TM benchmarks are not presented because changing α has no effect. The 'high' contention versions of `kmeans` and `vacation` are excluded because they behave identically to the low contention versions presented here.

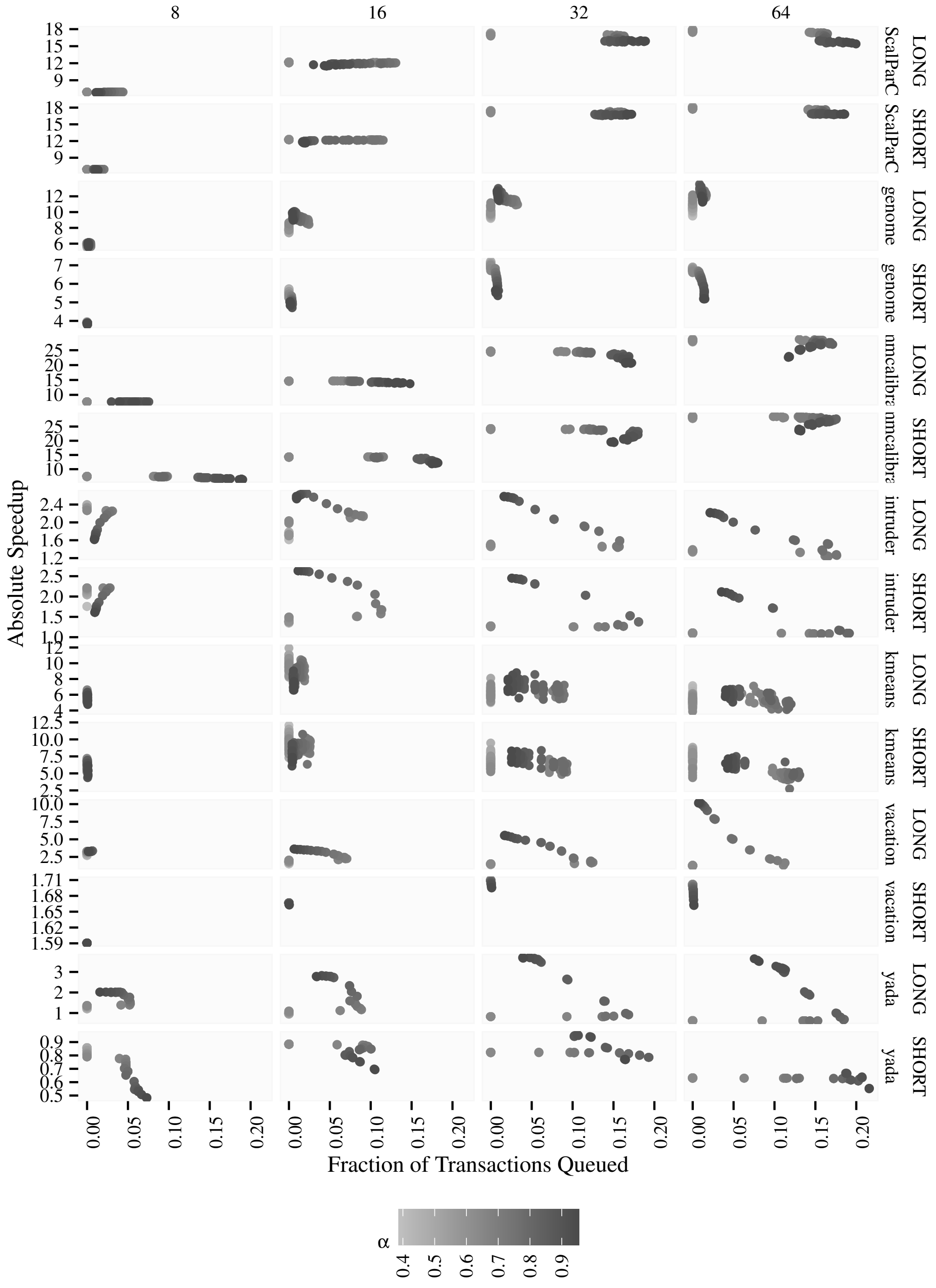

Figure 7.11: The relationship between the fraction of transactions queued and absolute speedup in the RMS-TM and STAMP benchmarks. UtilityMine, apriori, fluidanimate, hmmpfam, and hmmsearch are excluded because they see no relationship between speedup and the fraction of transactions queued.

7.6 Tuning PEW's T and α

PEW controls serialization by estimating the amount of wasted work a thread has done. This policy serializes if history (controlled by a parameter α in a similar manner to BE-ATS) indicates that the thread has been wasting too much work, determined by the ratio of committed to uncommitted work PEW dropping beneath a threshold T. A higher threshold is less lenient about wasted work, serializing more often than a lower threshold which allows substantially more wasted work.

PEW requires updating an average for every transaction start or abort. To accomplish that the serialization manager has a third parameter k that chunks the updates to reduce overhead, The computed PEW is only updated every k operations.

Figure 7.12 shows the results of an experiment that varies the threshold and α parameters for PEW while executing the STAMP benchmarks. In the plots in the four left columns $k = 0$ leading to the updating of PEW in every execution. For the plots in the four right columns $k = 10$ and therefore PEW is updated every tenth execution. The performance of the RMS-TM benchmarks does not vary when the parameters of the PEW policy are changed. Therefore, results for that suite are not presented.

Before these experiments, the expectation was that $k = 0$ would lead to higher overhead. Surprisingly, the overhead of updating PEW does not appear to have the largest effect on performance. Changing k appears to instead have a dramatic effect, in many cases reducing performance substantially, and dramatically altering the tuning curves for the STAMP benchmarks. For instance, `genome` in long running mode at 32 threads sees a complete reversal of speedup trend when varying α between $k = 0$ and $k = 10$. Updating PEW too aggressively causes overly aggressive serialization, as seen in Figure 7.13. This figure plots the serialization fraction relative to the threshold and α values. Overly aggressive serialization explains most of the poor-performing cases for $k = 0$.

One of the interesting results from Figure 7.13 is how successfully the threshold parameter is able to control the serialization rate, as designed, when using PEW and $k = 10$. Almost every benchmark has a very similar curve of serialization fraction. Performance is then dependent on the program and algorithm's requirements and resource usage.

These results provide clear evidence that speculation is one of the key elements for TM performance in most cases. The performance is terrible for `genome` in long-running mode with a high threshold T because `genome`, when run in long-running mode, needs speculation in order to perform well. The serialization-fraction results help explain the subset of the benchmarks that do not perform as expected in transactional memory: `genome` in short-running mode and `kmeans`. In both cases PEW performs best with as little speculation as possible, which is accomplished with low α and high T. These cases are pathological failures in the BG/Q's BE-HTM system.

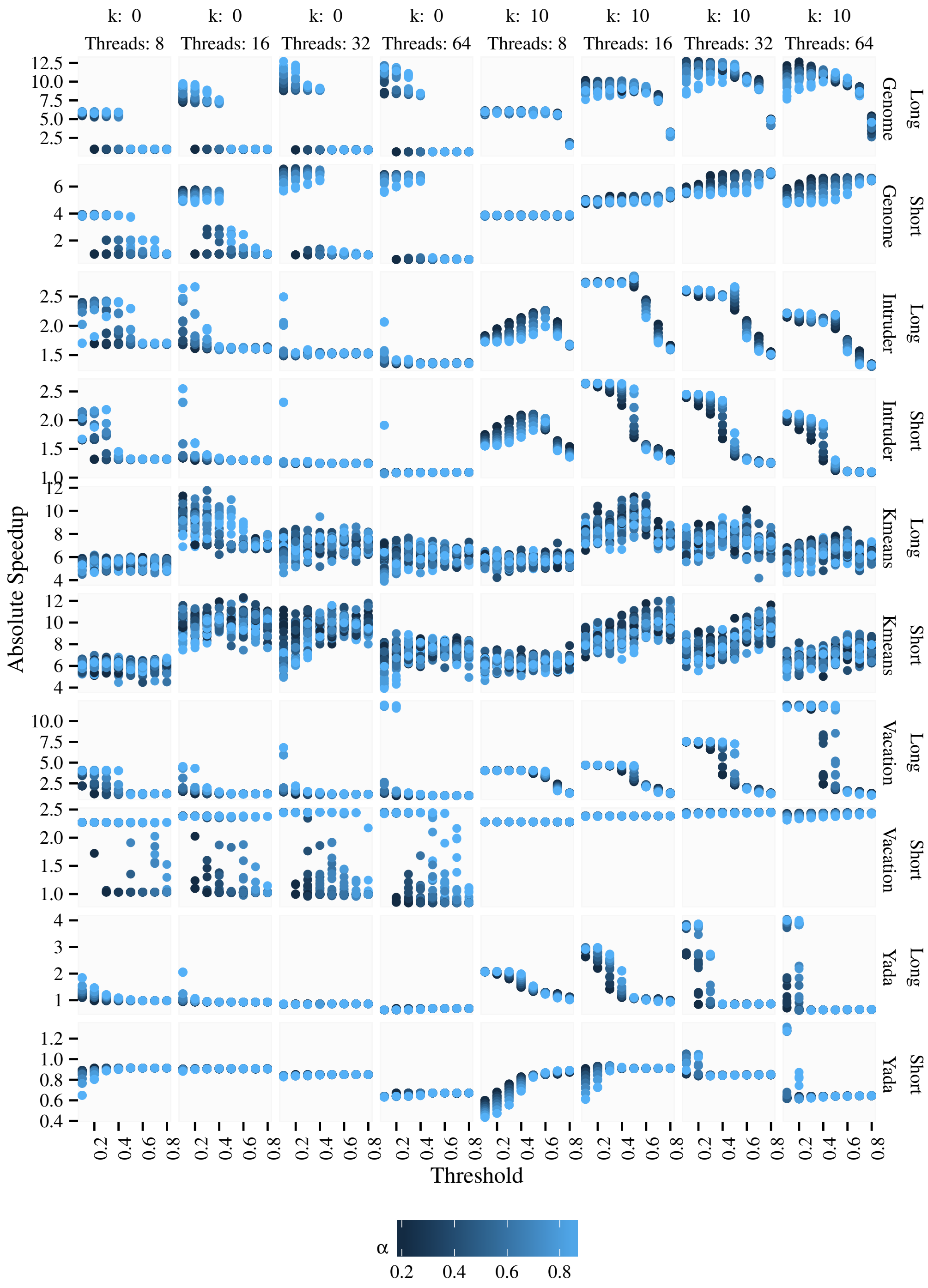

Figure 7.12: Varying the threshold and alpha parameters in STAMP running PEW for two values of *k* in Long and Short Running mode, for 8-64 threads. Results for 1-4 threads and the RMS-TM benchmarks are not presented because they are insensitive to parameter values.

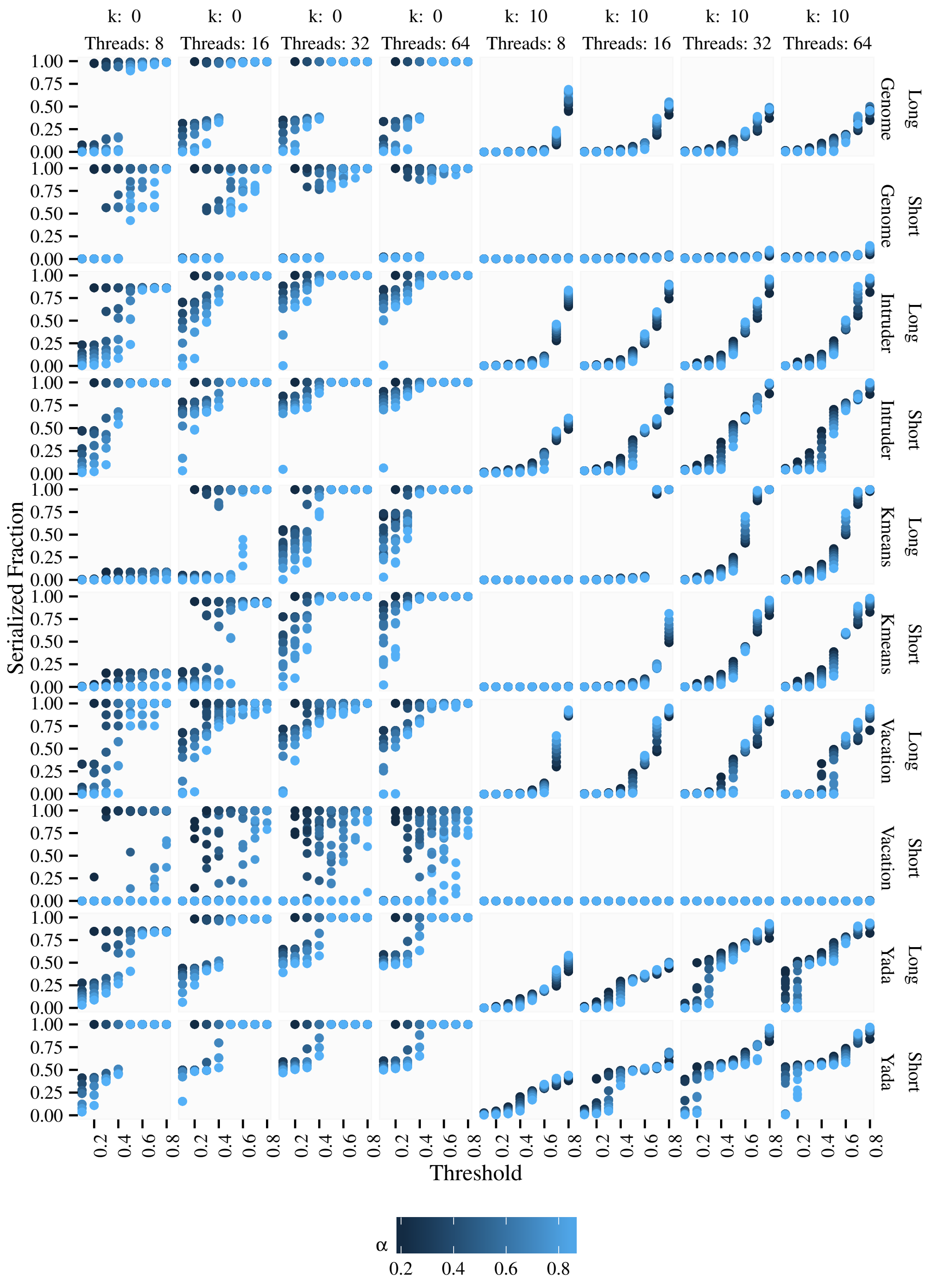

Figure 7.13: Varying the threshold and alpha parameters in PEW for two values of E in Short Running mode on STAMP, showing fraction of transactions serialized for 8-64 threads.

7.7 Tuning, a summary

The summary of the tuning exercise presented in this chapter is that manager performance is highly dependent on the combination of tuning, benchmark, and running-mode. This means that comparing performance between *managers* is difficult, as being fair requires paying attention to tuning.

The benchmarks themselves appear to have preferences as to the tuning, and the kind of behaviour the tuning promotes: benchmarks like `genome` in short-running mode and `kmeans` find no benefit from speculative execution, and almost invariably prefer a tuning that promotes serialization over speculation. On the other hand, benchmarks like `yada` prefer tunings that promote speculation, with almost any serialization reducing performance.

With the limited exception of LimitMeanST, the changed decision making *processes* explored among the differing serialization managers have little effect on changing outcomes through a 'better' serialization. LimitMeanST is the exception due to its ability to improve the performance of `intruder` in a limited sense.

Chapter 8

Serialization-Manager-Driven Performance Effects

8.1 Performance Unpredictability

Chapter 3 raised two pitfalls that need to be kept in mind before drawing any conclusions about serialization management. (1) We must consider the effect that the BE-HTM platform has on a serialization manager's performance at a particular tuning setting on a particular benchmark. (2) Applications have different transactional profiles that have different demands on the TM system and its serialization manager, therefore a broad set of applications need to be included in the experimental evaluation. Furthermore, it was discussed how desirable it would be for there to be some level of generalizability in serialization management — a good tuning for a particular benchmark should work well on the same benchmark on another platform, or on another benchmark on the same platform.

The results presented in Chapter 7 show that unpredictability in serialization management seems to be the norm, as there don't seem to be many generalizable rules. The lack of generalizabilities indicate that **1)** the platform has a very substantial effect on serialization management for many benchmarks. Across all the serialization managers explored in this section, each revealed a set of benchmarks (typically `genome`, `intruder` and `vacation`) that found opposite effects in long- and short-running mode when varying the parameters of the serialization manager. **2)** The individual benchmarks often require very different handling, with opposite reaction to parameters in the same running-mode being common. For instance, to achieve maximum performance kmeans requires aggressive serialization and speculative retries are almost never a good idea for kmeans. In contrast, yada's highest performance occurs with zero serialization. **3)** There is almost no generalizability in tuning of parameters for serialization management. Finding the correct parameter value is essential for performance, even within the same benchmark running with a different number of threads. This can be seen with intruder often changing the best tuning from one extreme at 2-8 threads to another extreme at 16 or more threads (see Figure 7.1).

8.2 Stable Trends

Though there is little in the way of generalizability in the tuning of parameter values for serialization managers that would allow for a generalization of performance trends across benchmarks or platforms, there is at least one property that appears to hold relatively stable for each benchmark across serialization managers. Figure 8.1 plots the relationship between the serialization fraction — the fraction of transactions committed in serialization mode — and absolute speedup at 16 threads for each of the RMS-TM and STAMP benchmarks across all serialization managers. By looking at the datapoint trends across the rows, corresponding to benchmarks, it can be seen that despite the varying strategies deployed by the serialization managers, the performance trend for a particular benchmark appears to remain relatively constant, aside from a small number of exceptional cases previously discussed. For example, intruder sees a different serialization fraction curve using LimitMeanST over the other serialization managers.

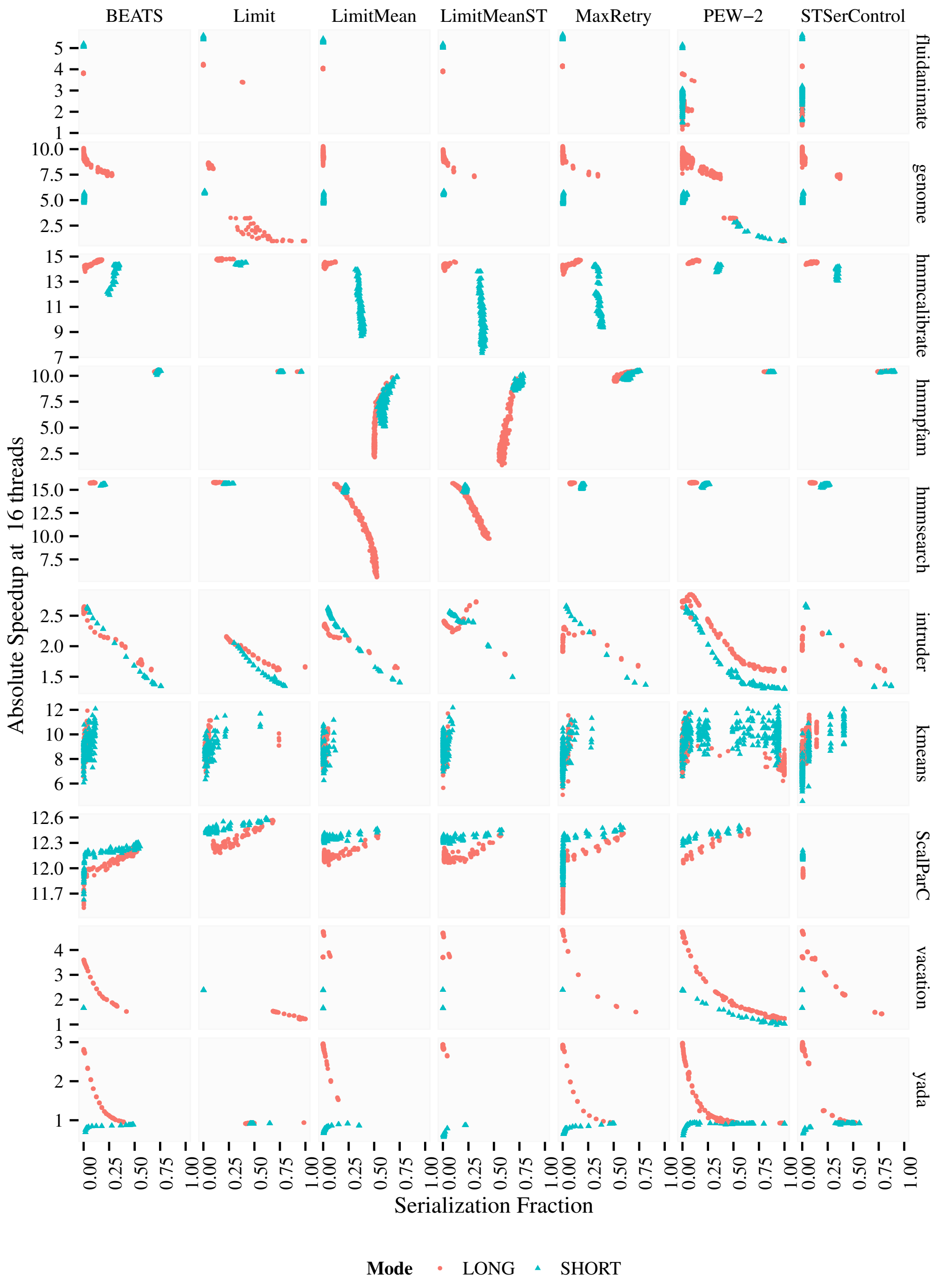

Figure 8.1: Serialization Fraction and its relationship to absolute speedup for 16 threads, run in both long and short running mode

8.3 Wasted-Work Hypothesis

Section 5.6 raised a common hypothesis in TM systems: that wasted work is important to avoid. Wasted work here means instructions executed in aborted transactions. The expectation is that performance should be best executing the minimum possible number of instructions, because it indicates that the computation was performed with maximum efficiency. Using the data on instructions executed aggregated across all threads collected using the Blue Gene Performance Monitor hardware event counters (BGPM counters) we can explore this hypothesis in more detail.

Figure 8.2 shows a strong relationship for the benchmark `genome` between the number of executed instructions and speedup. As expected, the best performance appears when the fewest instructions are executed. This relationship is not a universal trend though. Figure 8.3 plots the same data for `yada`, which finds some configurations running best with more instructions executed, with the results at 8 threads being particularly different.

Another measure of wasted work would be the fraction of dynamic transaction executions that are aborted, or the ratio between the number of aborts and the number of committed transactions. This ratio is also presented in Figures 8.2 and 8.3 in the form of the point sizes. These point sizes corroborate the conclusions drawn from the instructions executed, showing `yada` performing better when more aborts occur at lower thread counts, and `genome` performing better with less total computation and fewer aborts across all configurations.

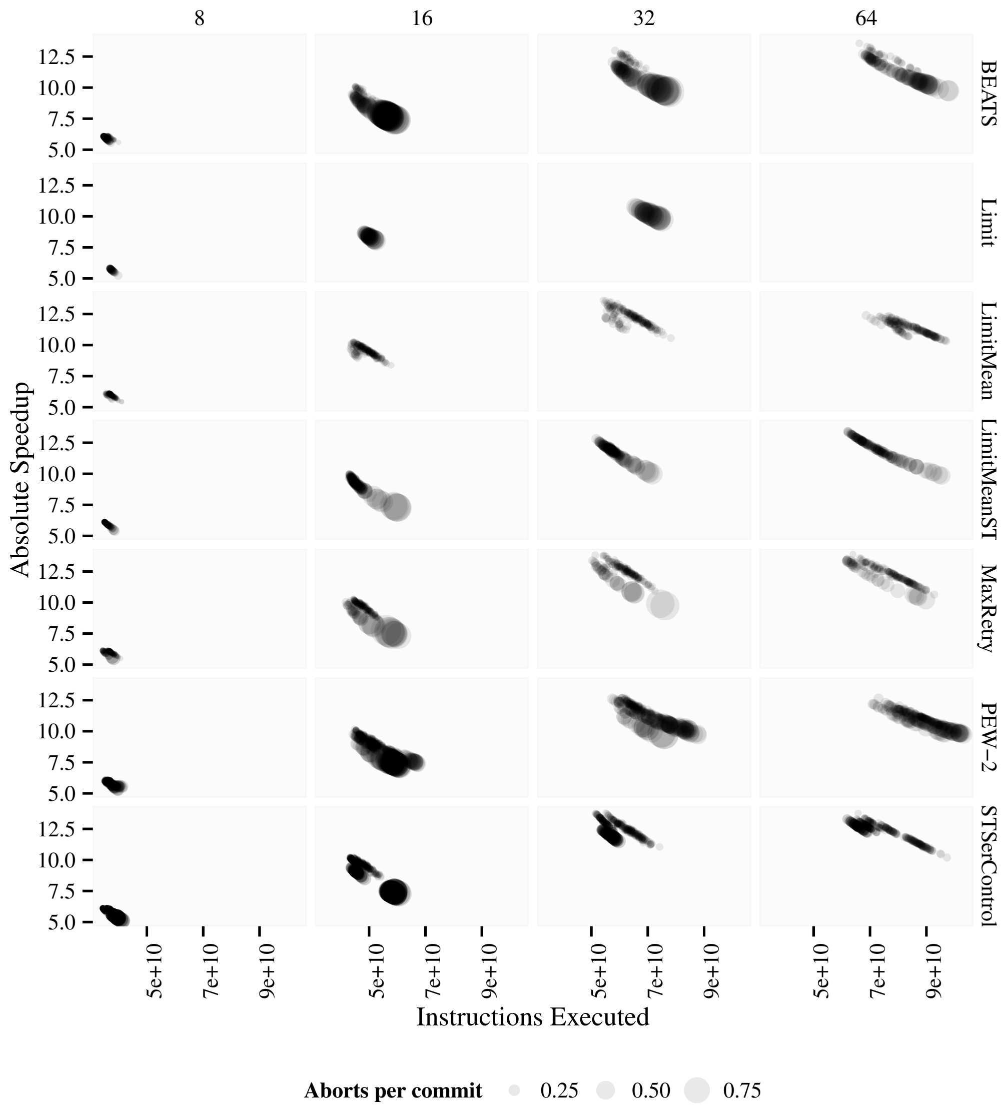

Figure 8.2: Instructions Executed compared to Absolute Speedup for genome run with 8-64 threads in long running, plotting the top 30% of executions, where the size of each point corresponds to the number of Aborts per commit, with smaller dots having fewer aborts.

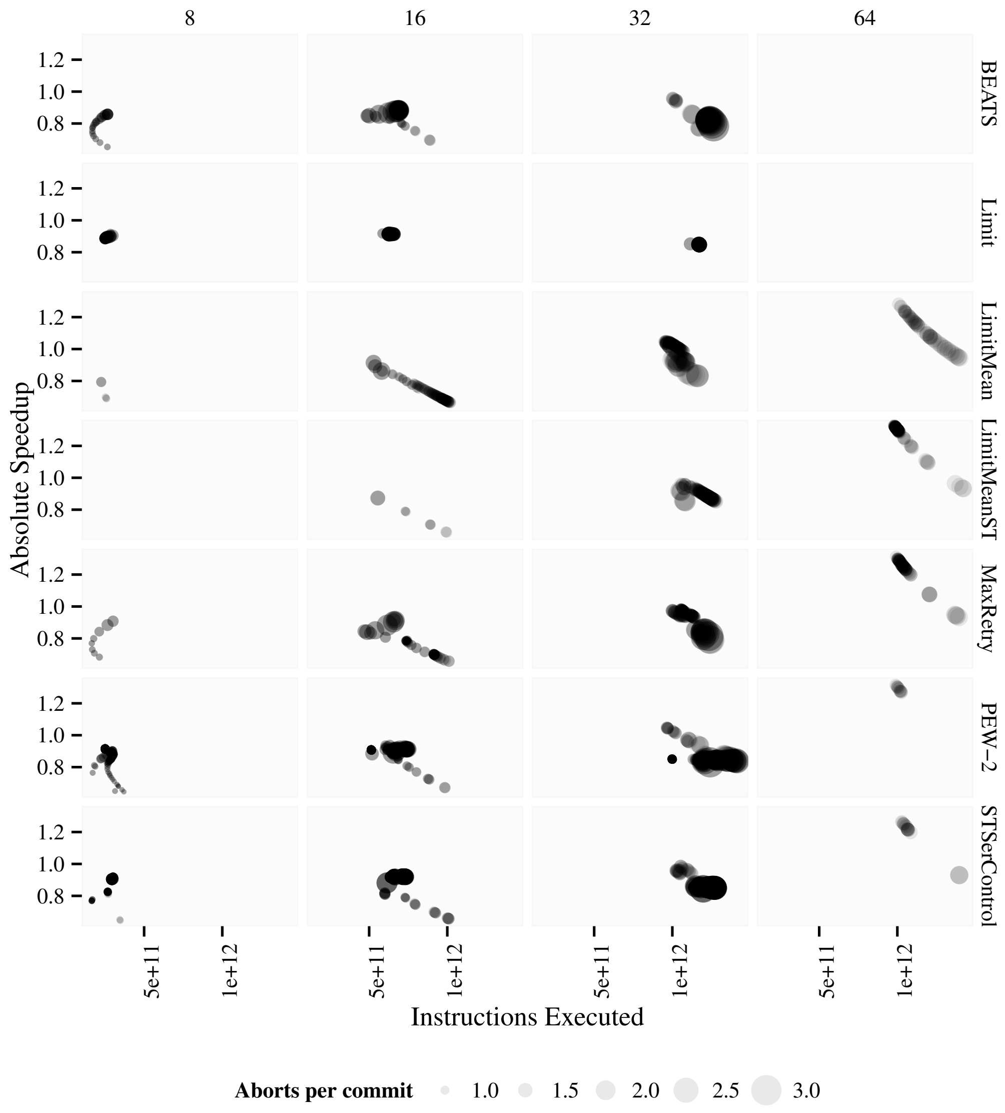

Figure 8.3: Instructions Executed compared to Absolute Speedup for yada run with 8-64 threads in short-running mode, plotting the top 30% of executions, where the size of each point corresponds to the number of Aborts per commit, with smaller dots having fewer aborts.

8.4 Serialization Management and Performance

The results in Chapter 7 showed how performance was affected for each individual serialization manager described in Chapter 5 as the parameters that control the serialization managers were varied. Figures 8.4 and 8.5 summarize the results of all the experimentation presented in Chapter 7. They show the distribution of Absolute Speedups achieved for each benchmark in each running mode by each serialization manager, across all the parameter values explored in the form of violin plots.[1] A tight distribution for a particular serialization manager indicates that the benchmark performance does not change much over the tunings explored in Chapter 7.

This thesis hypothesized that improved serialization management can improve the performance of BE-HTM parallelized programs, much in the same way that improved Contention Management can improve the performance of STM parallelized programs. The distribution of speedups in the figures show that overall the RMS-TM benchmarks (Figure 8.5) are markedly less sensitive to the effects of serialization management than the STAMP benchmarks (Figures 8.4). In the STAMP benchmarks the improvement from the worst to the best performance can often exceed 1000% (in the case of `genome` executing in long-running mode). This result can be explained by the low proportion of time spent in transactions in the RMS-TM benchmarks relative to the STAMP benchmarks. This result also indicates that for heavily transactional applications, such as the STAMP benchmarks, serialization management is a critical aspect of performance.

The best-achieved performance for each runtime is the highest point on the violin plot. Most combinations of benchmarks and mode see some change in performance across different serialization managers and tunings. However, despite these changes, most serialization managers appear to achieve equally high speedup on most bench-

[1]A violin plot uses the variable width of a "violin" to show the empirical distribution of values: thin violin sections have low density (or few data-points), and thicker sections have high density (or many data points). Figures 8.4 and 8.5 use modified violin plots that normalize the width of the violin such that each violin has the same widest point. This can cause distortion — in comparison to a violin plot that normalizes area — because wide uniform distributions will have substantially more visual weight than deserved. However, it is appropriate for the plots presented here because (1) we are interested in *relative* distributions rather than absolute distributions; and (2) each serialization manager has a different number of samples relating to the size of the explored parameter space.

marks for at least one tuning. While we don't have an oracle that would indicate if this is the highest possible speedup, the fact that multiple serialization managers achieve the same speedup indicates that it may be the highest speedup achievable within the serialization manager paradigm.

The large range of speedups in the STAMP benchmarks, caused by changing serialization manager or by tuning for a particular serialization manager, provides substantial support for the hypothesis of this thesis: improved serialization management can improve performance much like improved Contention Management can improve STM program performance. Similarly, poor serialization management can destroy all possible performance gains, especially for applications similar to the STAMP benchmarks that spend the majority of application time in transactions.

For many benchmarks each individual serialization manager has a large range of speedups across the possible tunings indicating that it is far more important to choose the correct tuning for the deployed serialization manager than it is to choose the serialization manager. Even a simple serialization manager, like MAX-RETRY, is a high performer for most benchmarks with at least one particular value of allowed rollbacks. The exceptions are: `kmeans`, that sees a better result with either BE-ATS or LIMIT, and `intruder` executing in long-running mode, `genome` and `yada` executing in short-running mode, all three of which see a better speedup with LIMITMEANST.

Across all the presented configurations there is no consistent ordering of serialization managers in terms of the best-achieved speedup. The results are completely dependent on the application, the thread-count, and the platform (running mode). This extensive experimental evaluation appears to put to rest the hope that there could be a 'best' serialization manager for the BG/Q BE-HTM. Further experimentation is required to determine if this is also the case for other BE-HTMs.

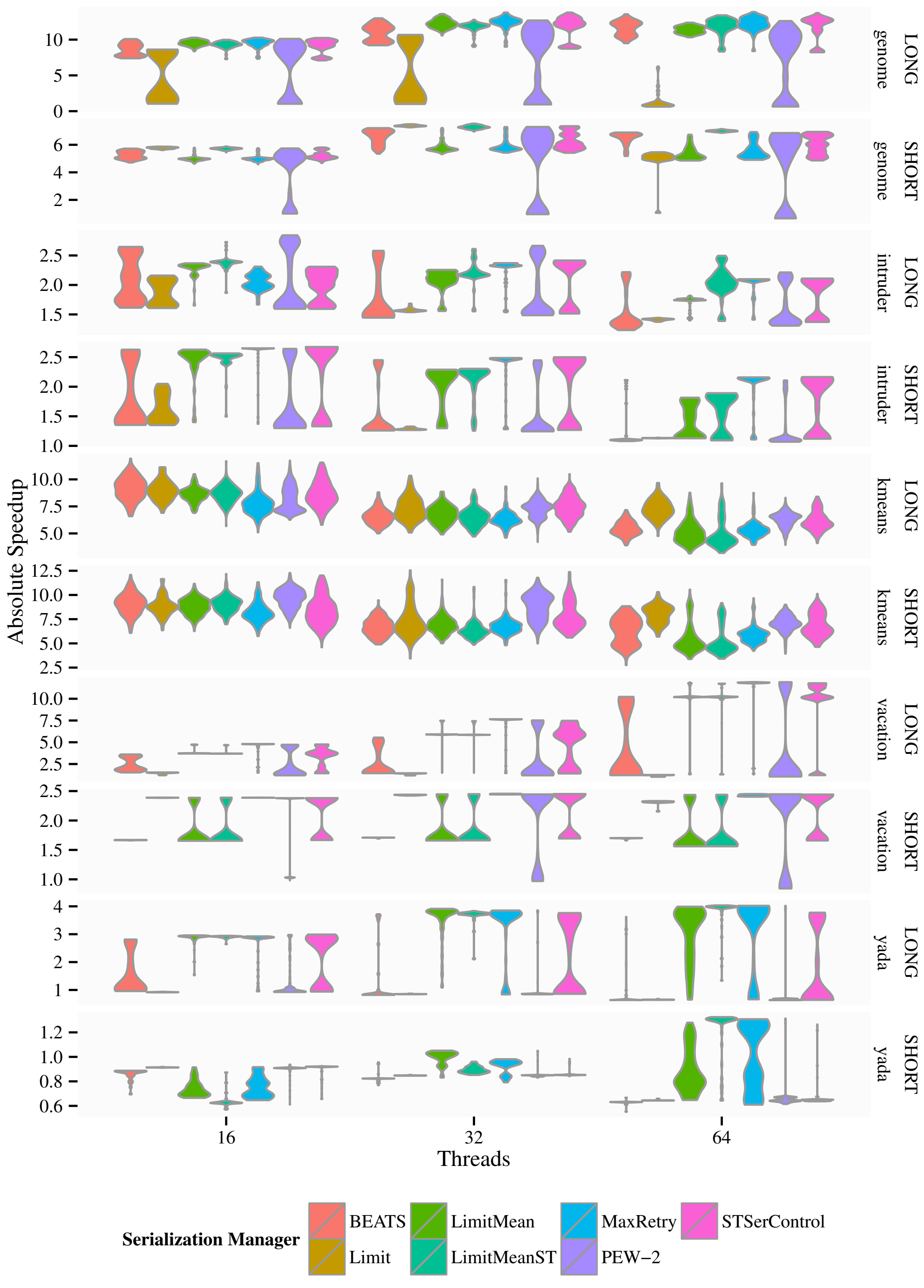

Figure 8.4: Absolute Speedup distribution for STAMP benchmarks over all serialization managers and all their tunings displayed as violin plot (exponential-delay and capacity-induced serialization enabled)

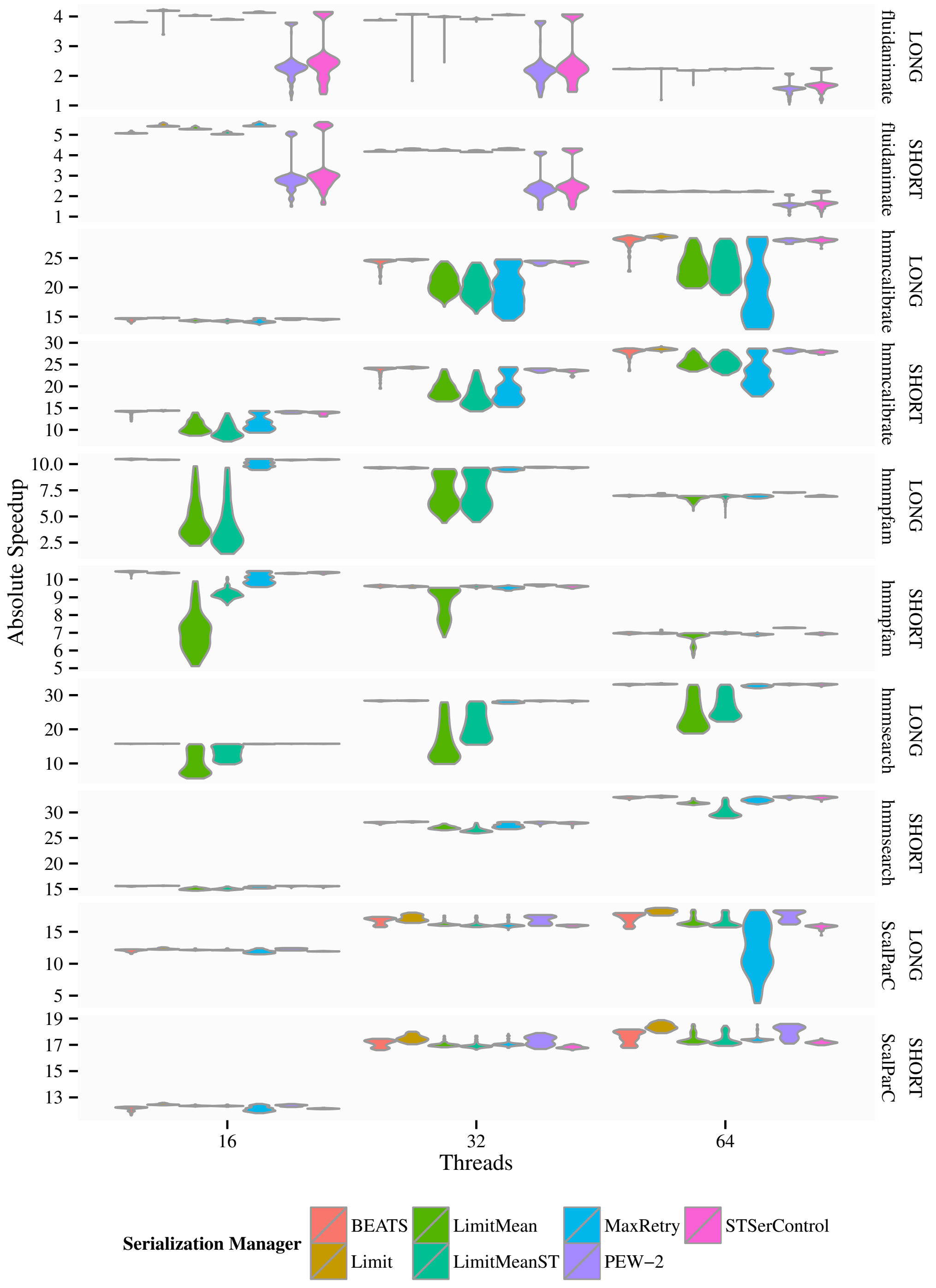

Figure 8.5: Absolute Speedup distribution for RMS-TM benchmarks over all serialization managers and all their tunings displayed as violin plot (exponential-delay and capacity-induced serialization enabled)

Chapter 9

Related Work

Related work for this thesis can be split into categories: HTM Performance analyses, non-speculative serialization and similar systems, enhancements to non-speculative serialization and transactional profiling.

9.1 HTM Performance Analyses

Christie *et al.* evaluate the AMD ASF proposal on a number of transactional memory benchmarks [19]. To provide forward progress they provide a non-speculative serialization mode, using a MAX-RETRY policy (though the allowed number of retries is not discussed) with capacity-induced serialization. Unlike the study in this thesis that focuses on serialization management, their evaluation fixes serialization management and instead varies the implementation parameters for ASF, changing the amount of supported speculative state.

Yoo *et al.* investigate Intel's RTM system with CLOMP-TM [65] and STAMP [78]. Their investigation uses a serialization manager similar to MAX-RETRY with a fixed number of allowed rollbacks (5) that they state that performes 'best'. Given the results of the study in this thesis, it is not clear how they determine what 'best' means under the competing constraints of different benchmarks, though by tuning their version of MAX-RETRY they acknowledge the problem described here.

Wang *et al.* investigate a limited form of tuning for a serialization manager very similar to MAX-RETRY running on Intel's RTM system [76]. They investigate an array-access microbenchmark and a molecular-dynamics simulation, and found a dramatic improvement in both speedup and abort rate by increasing the number of allowed retries within the range of 1-10. They note that there is likely no fixed optimum and on-line tuning may be desirable, as has also been indicated through the much more extensive study here.

Diegues and Romano describe a similar — though smaller scale — exploration of serialization management (named by them as *software-based fallback*) to the study presented in this thesis, as well as a serialization manager called Tuner that uses online-learning techniques from reinforcement learning to learn an appropriate tuning for each transaction in an application at runtime [27]. Their study involves two key elements: Varying the strategy for handling capacity aborts (which is substantially more important on Intel's Haswell because it can store much less speculative state) and adjusting the number of aborts allowed to each transaction. Their study draws many of the same conclusions as this study: 1) application performance can be dramatically affected by the choice of serialization management, 2) the best performing serialization manager changes from application to application. Driven by this conclusion, they present the serialization manager Tuner, which uses reinforcement-learning techniques to learn for each static transaction the highest performing strategy for dealing with capacity aborts, and the appropriate number of aborts for each static transaction. This path of self-tuning is expected to be very fruitful for serialization management, likely becoming the default approach in light of the challenges to static tuning exposed by this study.

9.2 Non-Speculative Serialization

Non-speculative execution as a method for handling poorly behaved speculation is a common theme in the literature of speculative parallelism. In addition, the notion has appeared in STM systems as a method for supporting actions that do not fit in the transactional model, such as I/O.

9.2.1 Hardware Systems with Non-speculative Execution

The proposed Transactional-Memory Coherence and Consistency (TCC) model executes the majority of code speculatively [38]. Similar to BG/Q, in order to support transactions that exhaust the speculative storage, or that execute irrevocable actions such as I/O, TCC supports a form of non-speculative execution that makes transactions inevitable and propagates their writes immediately. Transactions that are executed in non-speculative mode are overflowing or irrevocable transactions. In contrast to BG/Q's software implementation, TCC's non-speculative execution is a hardware mechanism built directly into its coherency protocol. TCC only uses non-speculative mode for overflowing transactions, and so has no policy to evaluate.

The STAMPede Thread-Level Speculation framework, which allows speculating loops and other control flow, provides a *home-free* token that indicates when a thread is executing the oldest work, at which point speculative state may overflow, and irrevocable actions may be taken safely [72]. STAMPede's home-free token serves much the same purpose as the global lock used for serialization in BG/Q, however it is used to provide irrevocability to the eldest piece of work. Compared to BG/Q, STAMPede has no tuning with regards to the home-free token, as there is only ever one eldest piece of work.

9.2.2 STM Analogues to Non-speculative Execution

Non-speculative serialization in Best-Effort HTM systems is similar to a concept from the STM literature described slightly differently by two different authors:

Irrevocable Transactions, described by Welc *et al.*, are software transactions that have been granted immunity to rollback by the TM system [77]. As in non-speculative serialization, these transactions can be allowed to perform I/O or system calls because their commit is guaranteed — and only one irrevocable transaction may be executing at any given time. Welc *et al.* describe irrevocable transactions as a programmer-accessible enhancement to the TM programming model both to allow actions that cannot be rolled back and to force completion of transactions that may be expected to induce large amounts of aborts if allowed to proceed speculatively – a hash-table resize operation for example. *Inevitable Transactions* is the name given to much the same concept by Spear *et al.* concurrently to the work of Welc *et al.* [69]. Spear *et al.* contributes multiple methods for supporting inevitable transactions, each allowing more concurrency than the base method — a Global Write Lock. Non-speculative serialization, as implemented in BG/Q, is effectively the Global Write Lock as described by Spear *et al.*. The remaining mechanisms described are applicable only to STM systems, unless hardware modifications are made to expose transactional read and write sets.

Welc *et al.* explores a naïve policy for automatic transition into irrevocable execution to improve performance by providing a quicker forward-progress guarantee similar to non-speculative serialization in BG/Q. Similar to the findings in this study, that serialization management is an important part of transactional performance. Welc *et al.* found that automatic transition into irrevocable execution could dramatically affect transactional performance, however their investigation was limited to a simple heuristic. The evaluation in Spear *et al.* found, as this study did, a dependence on workload characteristics for the selection of an inevitability mechanism, though their inevitability is only user-specified and tested on synthetic benchmarks.

9.3 HTM Serialization Enhancements

Calciu *et al.* presents a compiler-assisted improvement to non-speculative serialization that reduced the rate of lock acquisition, as well as a number of theoretical schemes to improve HTM systems [15]. Lazy lock checking was combined with a compiler transformation to ensure that any indirect jumps that occur within a transaction are instrumented with checks to the global lock in order to avoid the possibility of reading a value written by a non-speculative transaction, and using that value to jump to a transaction-ending instruction, prematurely ending the transaction and violating atomicity[1]. Despite providing an enhanced non-speculative serialization, unlike this thesis the authors did not discuss the policy under which they reverted to non-speculative serialization, nor the settings of whatever policy they did use. As seen in our evaluations, this can be critical when evaluating performance.

[1]Indirect jumps are disallowed as part of the programming specification for BG/Q TM, thus no instrumentation is required for BG/Q even in lazy-lock checking mode.

9.4 Transactional Profiling

Time-series analysis of transactional memory programs has been rare. Previous work has been done on Software Transactional Memory Systems, or has required specialized hardware to be added to the processors [2, 18, 80]. The existence of the TM runtime in BG/Q's HTM makes inserting the event profiling trivial as compared to other HTM systems where the TM code is tightly woven with application code.

Lev describes a prototype transactional profiler called *T-PASS* —Transactional Program Analysis System — that used stub calls read by the DTrace dynamic tracing framework to implement the profiling. Lev's profiler could profile at a deeper level than the TEP presented in this thesis, including being able to profile conflict sets and finding silent-writes. However, these features rely on it having been built specifically for the SkySTM runtime [48]. Lev's profiler included some aggregate analyses, including computing dynamic transaction lengths, as were computed for Figure 4.4.

Zyulkyarov *et al.* describe a transactional profiler for an STM augmented C# system, and describe a visualization technique that inspired the visualization in Figure 4.5, though they did not explore the possible insight of aggregate data visualization [80].

Gottschlich *et al.* presents *TMProf*, a transactional memory profiler intended for end-user usage [33]. They create application traces similar to ours. However, instead of offline analyzers, they provide an interactive graphical front-end in order to display a micro-level analysis of the transactional execution in order to visually indicate areas of possible improvement in the algorithms. The focus of *TMProf* is on micro-level interactions and thus aggregate analyses are not discussed.

Chapter 10

Study Limitations

Though the study in this thesis is very comprehensive, it does not entirely cover the possible experimental space for Blue Gene/Q BE-HTM. Had time and resources permitted, further study could have explored:

- **Eager vs. Lazy conflict resolution.** Section 2.3.1 stated that BG/Q can resolve memory conflicts either eagerly or lazily. Eager conflict resolution was used throughout the experimental evaluation because a previous study, comparing the absolute speedup of Max-Retry obtained with eager and lazy conflict resolution on the STAMP benchmarks, found that eager performed as well as, or better than, lazy conflict resolution. While I believe that the outcome of testing with lazy conflict resolution would only be a different tuning curve for each manager, it is possible that it could provide substantially better performance when paired with a new serialization manager.

- **Eager vs. Lazy Lock acquisition.** Section 2.3.2 explained that BG/Q can check the irrevocable token either at transaction start (eagerly) or transaction finish (lazily). The runtime defaults were used to check the lock eagerly in short-running mode and lazily in long-running mode. While my expectation would be that experimentation with eager vs lazy lock checking would prove to just be another dimension for tuning, similar to conflict resolution, it is also possible that an interaction between serialization managers and lock checking could have remained hidden in this study.

- **Software Stack Modifications.** More aggressive modification of the software stack could have provided some interesting possibilities for optimization. In particular, exposing more information available at the kernel level to the serialization managers could have allowed for the design of more intelligent managers.

  Another option would be to raise the contention management decision to the runtime level to allow the run-time system to collect statistics, and possible be more intelligent than the kernel.

- **Exploring Sandboxing for Protection.** Similar to capacity-induced serialization, all serialization managers presented here serialized on the detection of a sandboxing violation. This is a safe assumption when TM programs expect to operate entirely on consistent state when executing a transaction. However, it is possible to write TM programs that use the hardware sandboxing as a safety guarantee for speculation— for example, allowing some non-transactional writes to a data structure, and relying on sandboxing and memory-protection to ensure that transactions do not commit when accessing invalid or unmapped pointers. Some STAMP benchmarks do precisely this— which may be a high-performance choice. If this form of TM programming were to become common, immediate serialization on a sandboxing violation may be too aggressive and may dramatically reduce performance.

Though exploration on Intel's Haswell [44], IBM's zEC12 [45] or POWER8 [14] would be possible and desirable in order to understand more points in the BE-HTM design space, it would require porting the transactional memory runtime built for BG/Q, and, in the case of Haswell, the XLC compiler to the new platform, both of which are beyond the scope of this thesis.

Chapter 11

Future Work

There are a number of possible directions that could be pursued in future work.

11.1 Formalization and Abstraction

Serialization management is very tightly intertwined with the TM runtime in BG/Q. This is an artifact of history, however in the course of this study I elected not to disentangle the dependencies. Future investigation of serialization management should specify an interface in order to provide pluggable serialization management.

11.2 Invasive Serialization Managers

Some contention managers or transactional schedulers created for STM cannot be implemented in BG/Q because of limitations imposed by the hardware and kernel implementations. For instance, a policy called Collision Avoidance and Resolution for Software Transactional Memory (CAR-STM) reschedules transactions that are predicted to cause failure with the goal of reducing the chances of repeated conflicts [28]. In its simplest version, CAR-STM only reschedules transactions that have already conflicted. The policy enters an aborted transaction into an execution queue owned by the thread that aborted the transaction. An implementation of CAR-STM on BG/Q would require changing how BG/Q saves transaction context in order to support rescheduling transactions on different threads, which would involve changes to the compiler and to the TM run-time system.

Another policy that would require changes to the BG/Q run-time system is `adaptSTM` that uses a variety of adaptation strategies to change, at runtime, thread-local parameters in an STM in order to improve performance [58]. Opportunities for performance benefits may exist by changing running modes online in a similar fashion at quiescent points in transactional execution, such as serialization. However, the BG/Q TM runtime was not designed with dynamic mode switching in mind, and would require substantial modification to do so safely.

Investigation of an adaptive policy, like that described by Diegues and Romano for Haswell, would be desirable [27]. Their adaptive policy learns, per transaction, an appropriate serialization response over the course of a program's execution, leading to a single policy that can adapt to multiple dynamic conditions. As described in this thesis, policy tuning is important on BG/Q, so it would be interesting to reproduce their policy for BG/Q.

11.3 Hardware Non-speculative Serialization

Some of the tradeoffs faced through non-speculative serialization could be ameliorated with hardware-supported non-speculative serialization. By having the hardware be aware of the semantics and requirements for non-speculative serialization there is a possibility that more overlap could be allowed during serialization.

Chapter 12

Conclusion

This thesis introduces a new area of research: serialization management, the mechanism for guaranteeing forward progress and high performance, decoupled from correctness, in Best-Effort HTM systems.

Seven serialization managers were described, some inspired by STM Contention Managers (the inspiration for serialization management). These contention managers were subjected to a very rigorous evaluation, the first of its kind, involving hundreds of thousands of experiments across two benchmark suites, two experimental platforms — represented by the two running modes of BG/Q — and seven different thread counts.

The conclusion of the experimental evaluation is that serialization management is a critical part of BE-HTM performance. The improvements provided by judicious serialization indicate that, for many applications, it may be worthwhile to provide non-speculative serialization and a serialization manager even on HTM platforms that guarantee forward progress through other mechanisms, such as LogTM [55].

Across the serialization managers it was shown that most serialization managers can achieve high performance on many benchmarks with the correct tuning. The take away from this is that tuning may be more important than serialization managers, though, there were exceptions for which the choice of serialization manager did affect maximum performance.

Each serialization manager was evaluated across a swath of its tuning parameters, in each of BG/Q's two modes. These two modes are neighbours in the design space of TM systems, and yet we found that the parameter tunings were often fragile, in that the correct tuning for one BG/Q mode was often not the correct tuning for the other.

This finding further emphasizes that tuning is a must, especially when deploying an application on a new platform.

Results from at least one combination of benchmark and serialization manager (`intruder` and LimitMeanST) combined with insight provided by the Transactional Event Profiler, a tool designed for this thesis, indicate that more intelligent policies than those designed here may yet be able to improve performance further.

Bibliography

[1] C. Scott Ananian, Krste Asanović, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *High-Performance Computer Architecture (HPCA)*, pages 316–327, San Francisco, CA, USA, February 2005.

[2] Mohammad Ansari, Kim Jarvis, Christos Kotselidis, Mikel Luján, Chris Kirkham, and Ian Watson. Profiling Transactional Memory Applications. In *Parallel, Distributed, and Network-based Processing (PDP)*, pages 11–20, Feb 2009.

[3] Mohammad Ansari, Christos Kotselidis, Mikel Luján, Chris Kirkham, and Ian Watson. On the Performance of Contention Managers for Complex Transactional Memory Benchmarks. In *International Symp. on Parallel and Distributed Computing (ISPDC)*, pages 83–90, Lisbon, Portugal, July 2009.

[4] Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson. Steal-on-Abort: Improving Transactional Memory Performance through Dynamic Transaction Reordering. In *High Performance Embedded Architectures and Compilers (HiPEAC)*, pages 5–18, Paphos, Cyprus, January 2009.

[5] D. A. Bader and K. Madduri. Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors. In *Intern. Conf. on High Performance Computing (HiPC)*, pages 465–476, Goa, India, December 2005.

[6] David A. Bader and Kamesh Madduri. Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors. In *HiPC*, pages 465–476, 2005.

[7] Alexandro Baldassin, Edson Borin, and Guido Araujo. On the Impact of Dynamic Memory Management on Software Transactional Memory Performance. In *ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2014.

[8] Lee Baugh, Naveen Neelakantam, and Craig B. Zilles. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. In *ISCA*, pages 115–126, 2008.

[9] Philip A. Bernstein and Eric Newcomer. *Principles of Transaction Processing*. Morgan Kaufman, 2009.

[10] Colin Blundell, E Christopher Lewis, and Milo MK Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2):17–17, 2006.

[11] 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.

[12] S. Borkar. Design challenges of technology scaling. *Intern. Symposium on Microarchitecture (MICRO)*, 19(4):23–29, Jul 1999.

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

[14] 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 *International Conference on Computer Architecture (ISCA)*, Tel-Aviv, Israel, 2013.

[15] Irina Calciu, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Improved Single Global Lock Fallback for Best-effort Hardware Transactional Memory. In *ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2014.

[16] 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.

[17] Calin Cascaval, Colin Blundell, Maged M. Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: Why is it only a Research Toy? *Communications of the Association for Computing Machinery*, 51(11):40–46, 2008.

[18] Hassan Chafi, Chi Cao Minh, Austen McDonald, Brian D. Carlstrom, JaeWoong Chung, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun. TAPE: a Transactional Application Profiling Environment. In *International Conference on Supercomputing (ICS)*, ICS '05, pages 199–208, New York, NY, USA, 2005. ACM.

[19] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Rivière. Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 27–40, New York, NY, USA, 2010. ACM.

[20] 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.

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

[22] 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 *International Conference on Computer Architecture (ISCA)*, pages 484–495, Austin, TX, USA, 2009.

[23] 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.

[24] Luke Dalessandro and Michael L. Scott. Sandboxing Transactional Memory. In *PACT*, pages 171–180, 2012.

[25] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 336–346, San Jose, California, USA, 2006.

[26] 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.

[27] Nuno Diegues and Paolo Romano. Self-Tuning Intel Transactional Synchronization Extensions. *11th International Conference on Autonomic Computing (ICAC 14)*, 2014.

[28] Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: Scheduling-Based Collision Avoidance and Resolution for Software Transactional Memory. In *Symposium on Principles of Distributed Computing (PODC)*, pages 125–135, Toronto, ON, Canada, August 2008.

[29] Aleksandar Dragojevic, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a research toy. *Communications of the Association for Computing Machinery*, pages 70–77, 2011.

[30] 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.

[31] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-Based Software Transactional Memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, December 2010.

[32] Mark Giampapa, Thomas Gooding, Todd Inglett, and Robert W. Wisniewski. Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene's CNK. In *Supercomputing Conference*, pages 1–10, 2010.

[33] Justin E. Gottschlich, Maurice P. Herlihy, Gilles A. Pokam, and Jeremy G. Siek. Visualizing Transactional Memory. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 159–170, New York, NY, USA, 2012. ACM.

[34] Jim Gray. A Transaction Model. In Jaco Bakker and Jan Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 282–298. Springer Berlin Heidelberg, 1980.

[35] Jim Gray. The Transaction Concept: Virtues and Limitations (Invited Paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 144–154. VLDB Endowment, 1981.

[36] Transactional Memory Specification Drafting Group. Draft Specification of Transactional Language Constructs for C++ (Version 1.1), 2012.

[37] Theo Haerder and Andreas Reuter. Principles of Transaction-oriented Database Recovery. *ACM Computer Surveys*, 15(4):287–317, December 1983.

[38] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Hen Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Knule Olukotun. Transactional Memory Coherence and Consistency. In *International Conference on Computer Architecture (ISCA)*, pages 102–, Munich, Germany, March 2004.

[39] R.A. Haring, M. Ohmacht, T.W. Fox, M.K. Gschwind, D.L. Satterfield, K. Sugavanam, P.W. Coteus, P. Heidelberger, M.A. Blumrich, R.W. Wisniewski, A. Gara, G.L.-T. Chiu, P.A. Boyle, N.H. Chist, and Changhoan Kim. The IBM Blue Gene/Q Compute Chip. *IEEE Micro*, 32(2):48–60, March-April 2012.

[40] 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.

[41] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier Science, 2012.

[42] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software Transactional Memory for Dynamic-Sized Data Structures. In *Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, Boston, MA, USA, July 2003.

[43] William N. Scherer III and Michael L. Scott. Contention Management in Dynamic Software Transactional Memory. In *Workshop on Concurrency and Synchronization in Java Programs (CSJP)*, St. John's, NL, Canada, July 2004.

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

[45] C. Jacobi, T. Slegel, and D. Greiner. Transactional Memory Architecture and Implementation for IBM System z. In *Intern. Symposium on Microarchitecture (MICRO)*, pages 25–36, Vancouver, BC, Canada, December 2012.

[46] D. Kanter. Analysis of Haswell's Transactional Memory. http://www.realworldtech.com/page.cfm?ArticleID=RWT02151 2050738, February 2012. Real World Technologies.

[47] G. Kestor, V. Karakostas, O. Unsal, A. Cristal, I. Hur, and M. Valero. RMS-TM: A Comprehensive Benchmark Suite for Transactional Memory Systems. In *Intern. Conf. on Performance Engineering (ICPE)*, pages 335–346, Karlsruhe, Germany, March 2011.

[48] Yossi Lev. *Debugging and Profiling of Transactional Programs*. PhD thesis, Brown University, 2010.

[49] Heike McCraw, Daniel Terpstra, Jack Dongarra, Kris Davis, and Roy G. Musselman. Beyond the CPU: Hardware Performance Counter Monitoring on Blue Gene/Q. In *ISC*, pages 213–225, 2013.

[50] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural Semantics for Practical Transactional Memory. In *International Conference on Computer Architecture (ISCA)*, pages 53–65, Boston, MA, USA, June 2006.

[51] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* 2014.

[52] 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.

[53] 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.

[54] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965.

[55] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-Based Transactional Memory. In *High-Performance Computer Architecture (HPCA)*, pages 258–269, Austin, TX, USA, February 2006.

[56] Daniel Nicácio, Alexandro Baldassin, and Guido Araújo. Transaction Scheduling Using Dynamic Conflict Avoidance. *International Journal of Parallel Programming (IJPP)*, 41(1):89–110, 2012.

[57] Martin Ohmacht, Amy Wang, Thomas Gooding, Ben J. Nathanson, Indira Nair, Geert Janssen, Marcel Schaal, and Burkhard D. Steinmacher-Burow. IBM Blue Gene/Q memory subsystem with speculative execution and transactional memory. *IBM Journal of Research and Development*, 57(1/2):7, 2013.

[58] Matias Payer and Thomas R. Gross. Performance Evaluation of Adaptivity in Software Transactional Memory. In *Intern. Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 165–174, Austin, TX, USA, April 2011.

[59] Marcio Machado Pereira, J. Nelson Amaral, and Guido Araújo. Measuring Effective Work to Reward Success in Dynamic Transaction Scheduling. In *Intern. Conf. on Parallel Processing (ICPP)*, icpp2014loc, 2014.

[60] P.E. Ross. Why CPU Frequency Stalled. *Spectrum, IEEE*, 45(4):72–72, April 2008.

[61] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *PPOPP*, pages 47–56, 2010.

[62] Wenjia Ruan, Yujie Liu, and Michael Spear. STAMP need not be considered harmful. In *ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2014.

[63] 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.

[64] William N. Scherer III and Michael L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, Jul 2005.

[65] Martin Schindewolf, Barna Bihari, John Gyllenhaal, Martin Schulz, Amy Wang, and Wolfgang Karl. What Scientific Applications Can Benefit from Hardware Transactional Memory? In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 90:1–90:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[66] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[67] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible Decoupled Transactional Memory Support. In *International Conference on Computer Architecture (ISCA)*, pages 139–150, Beijing, China, June 2008.

[68] 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.

[69] Michael Spear, Maged Michael, and Michael Scott. Inevitability Mechanisms for Software Transactional Memory. In *ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, Feb 2008.

[70] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A Comprehensive Strategy for Contention Management in Software Transactional Memory. In *Principles and practice of parallel programming*, pages 141–150, Raleigh, NC, USA, February 2009.

[71] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A Scalable Approach to Thread-level Speculation. In *International Conference on Computer Architecture (ISCA)*, pages 1–12, Vancouver, BC, Canada, June 2000.

[72] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The STAMPede Approach to Thread-Level Speculation. *ACM Transactions on Computer Systems (TOCS)*, 23(3):253–300, 2005.

[73] Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. Multiple Reservations and the Oklahoma Update. *IEEE Parallel & Distributed Technology (PDT)*, pages 58–71, 1993.

[74] Amy Wang, Matthew Gaudet, Peng Wu, Martin Ohmacht, José Nelson Amaral, Christopher Barton, Raul Silvera, and Maged M. Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Parallel Architectures and Compilation Techniques (PACT)*, Sept 2012.

[75] Amy Wang, Matthew Gaudet, Peng Wu, Martin Ohmacht, José Nelson Amaral, Christopher Barton, Raul Silvera, and Maged M. Michael. Software Support and Evaluation of Hardware Transaction Memory on Blue Gene/Q. *IEEE Transactions on Computers*, 99(PrePrints):1, 2013.

[76] Mike Dai Wang, Mihai Burcea, Linghan Li, Sahel Sharifymoghaddam, Greg Steffan, and Cristiana Amza. Exploring the Performance and Programmability Design Space of Hardware Transactional Memory. In *ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2014.

[77] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. In *ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 285–296, 2008.

[78] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of intel Transactional Synchronization eXtensions for high-performance computing. In *SC*, page 19, 2013.

[79] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, Munich, Germany, June 2008.

[80] Ferad Zyulkyarov, Srdjan Stipic, Osman S. Unsal, Adrián Cristal, Tim Harris, Ibrahim Hur, and Mateo Valero. Profiling and Optimizing Transactional Memory Applications. *International Journal of Parallel Programming*, 40:25–56, February 2012.

Appendix A

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

This appendix contains the contents of the paper "Software Support and Evaluation of Hardware Transaction Memory on Blue Gene/Q", which is to be published in the IEEE Journal *Transactions On Computers* [75].

The contents of this appendix is the result of a collaborative work, and authorship is shared among Amy Wang, myself, Peng Wu, Martin Ohmacht, José Nelson Amaral, Christopher Barton, Raul Silvera, and Maged M. Michael.

A.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 [40, 73].

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) [23, 31, 63, 68] and conducted simulation-based studies of hardware transactional memory (HTM) [11, 13, 53]. More recently real HTM implementations start to emerge. An early implementation of HTM was reported but never distributed commercially [22]. For the HTM by Azul, there is little public disclosure on the implementation and no performance study of the TM support [21]. The specification of a hardware extension for TM in the AMD64 architecture has yet to be released in hardware [20]. Recently IBM [14, 39, 45] and Intel [44] 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 A.3) and an in-depth analysis of its major sources of overheads (Section A.6). One large pain point of STM is the high overhead associated with monitoring memory accesses and maintaining speculative state inside a transaction [16]. 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 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 [52] and RMS-TM [47] — running on BG/Q TM (Sections A.5 and A.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 [30]. 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 A.2 and A.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 A.2, A.3, and A.4 describe the TM programming model, BG/Q HTM implementation, and the software stack that supports the TM programming model, respectively. Section A.5 describes the evaluation methodology and the benchmarks. The performance study of BG/Q TM is presented in Section A.6 (comparison between the short- and long-running modes), Section A.7 (single-thread performance), and Section A.8 (scalability). A discussion of related work appears in Section A.9, and we conclude in Section A.10.

A.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 [36]. In a concurrent execution, transactions appear to execute sequentially in some total order with respect to each other. Specifically,

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

```
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];
    }
   }
}
```

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

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 A.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 run-time error. Similarly, exceptions thrown by a transaction are unsupported.

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

Figure A.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.

A.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 (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) [71]. 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.

A.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 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 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.

- **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

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

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.

A.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 (32M*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

[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.

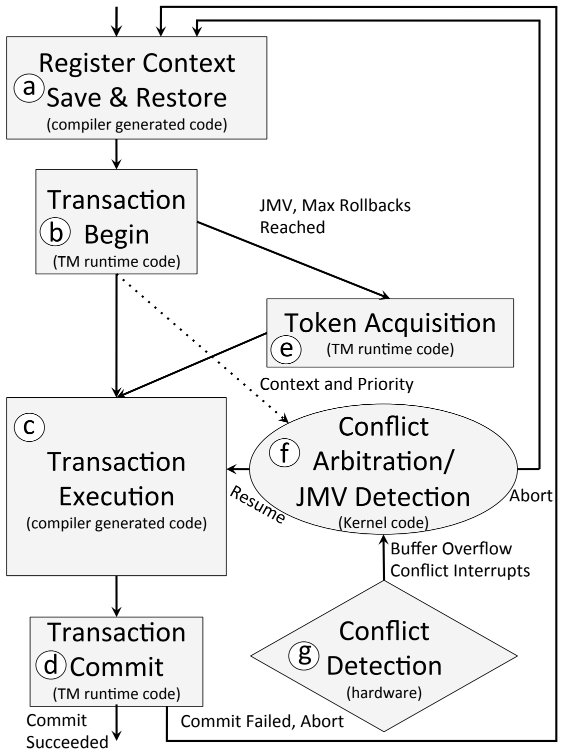

Figure A.2: Transactional Memory Execution Overview.

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.

A.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.

Figure A.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 Figure A.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.

A.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 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

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

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.

A.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 Figure A.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.

A.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.

A.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.

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` [5]	-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%

Table A.1: Benchmark Descriptions

A.5 Experimental Setup and Benchmarks

This evaluation of the BG/Q TM performance uses two benchmark suites. The STAMP benchmark suite [52] 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 [47]. 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 [31] and is manually instrumented to minimize the amount of tracked state. Table A.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

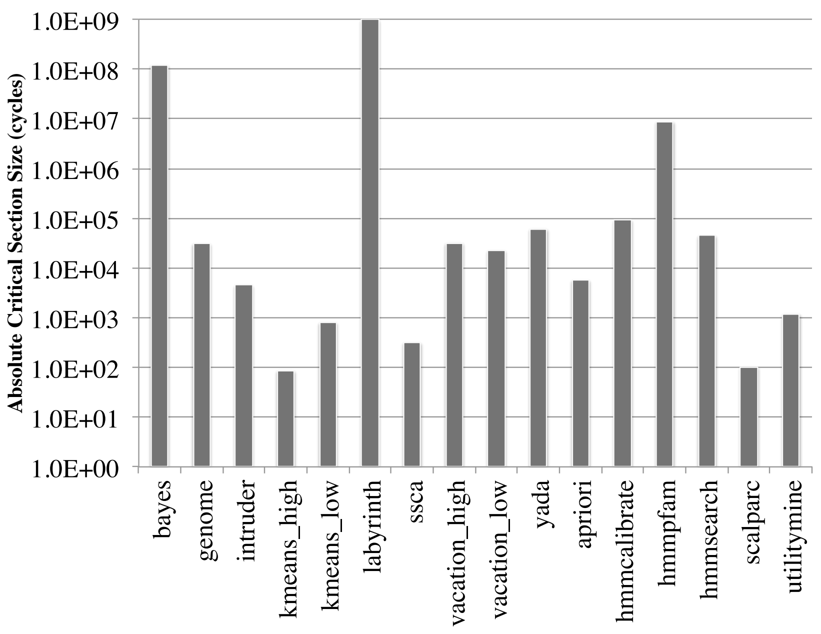

Figure A.3: Average time spent (in cycles) per dynamic instance of critical sections in the STAMP and RMS-TM benchmark suites.

(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.

Figure A.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.

As shown in Table A.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.

A.6 Long- vs. 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.

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

A.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

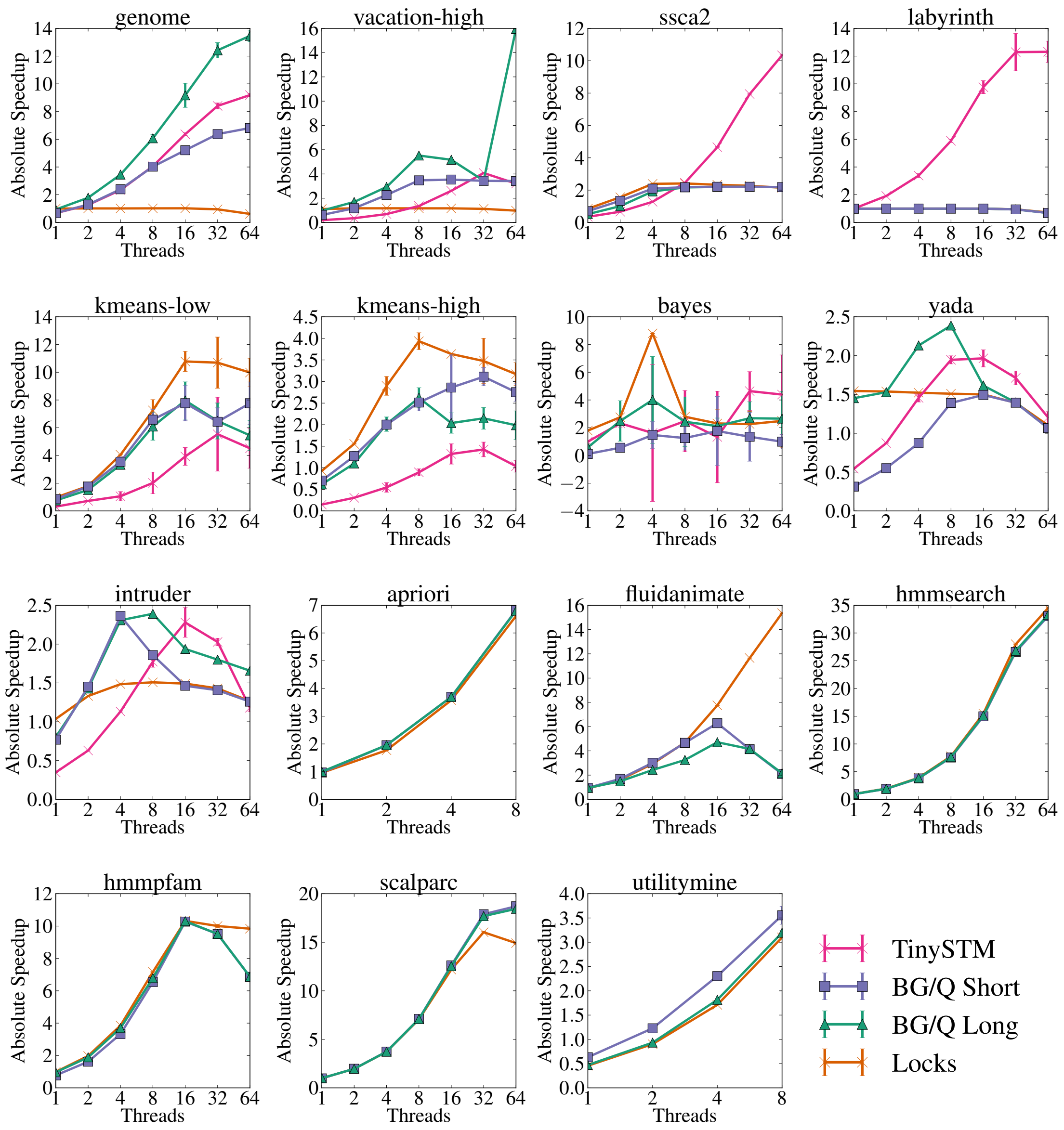

Figure A.4: Speedup of different critical section implementations of the STAMP and RMS-TM benchmark suites over the original sequential version of the benchmarks (`vacation-low` results were similar to `vacation-high` and `hmmcalibrate` were similar to `hmmsearch` — both are omitted).

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

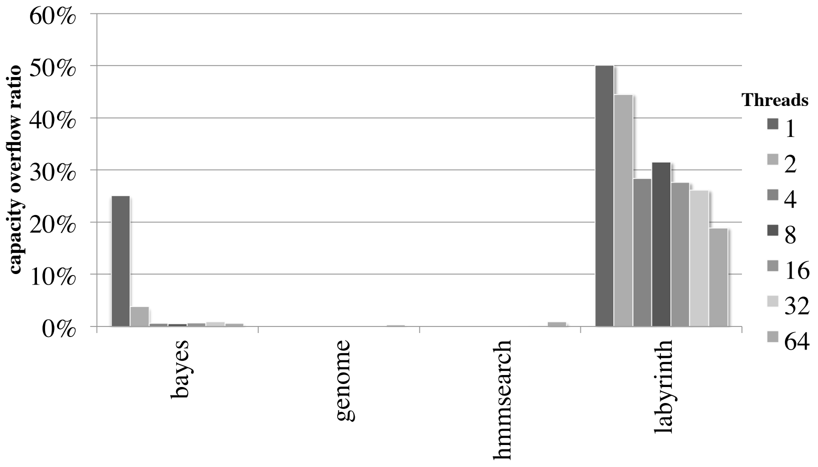

Figure A.5: LR mode: ratio of total transactions aborted due to capacity overflow.

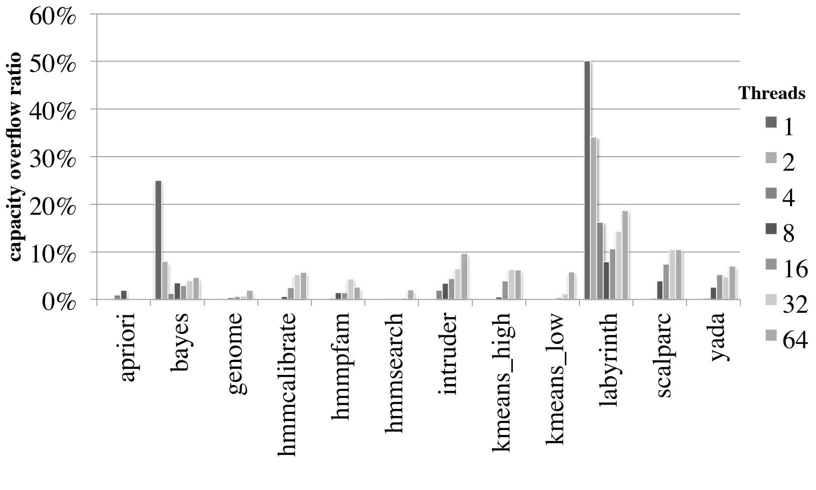

Figure A.6: SR mode: ratio of total transactions aborted due to capacity overflow.

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 %

Table A.2: Hardware performance monitor stats.

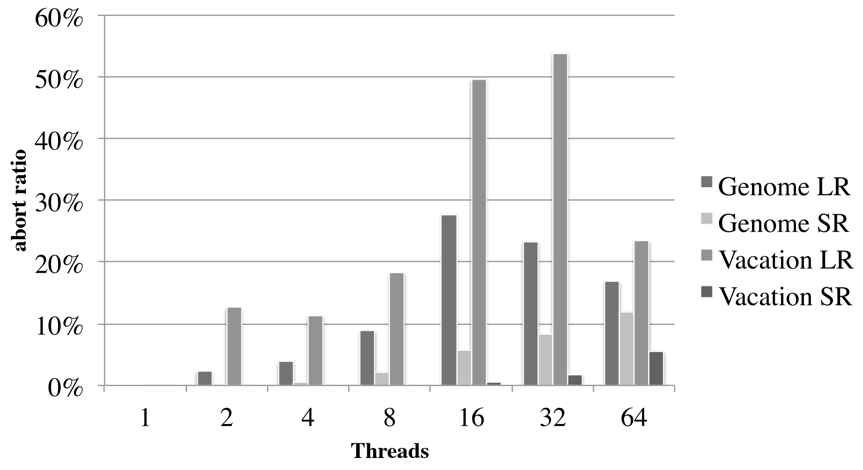

Figure A.7: The abort ratio of `genome` and `vacation`.

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 A.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 A.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.

A.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. Figure A.6 and Figure A.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.

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

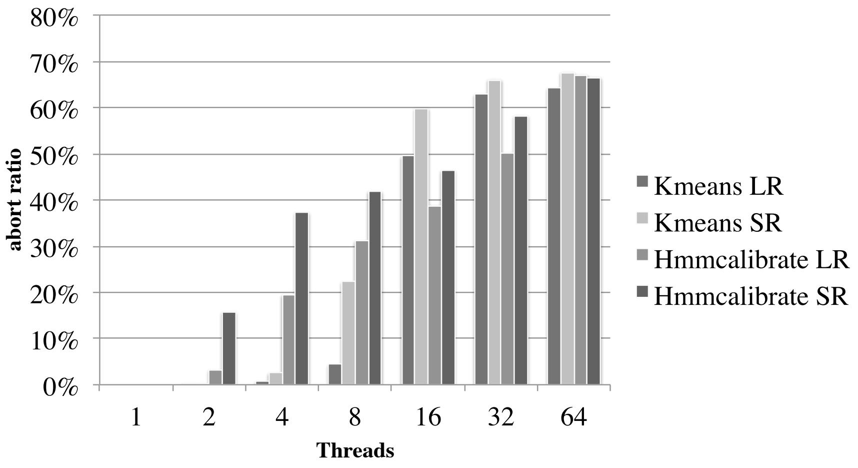

Figure A.8: The abort ratio of `kmeans` and `calibrate`.

As shown in Figure A.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 Figure A.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 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.

A.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. Figure A.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 Figure A.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 A.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.

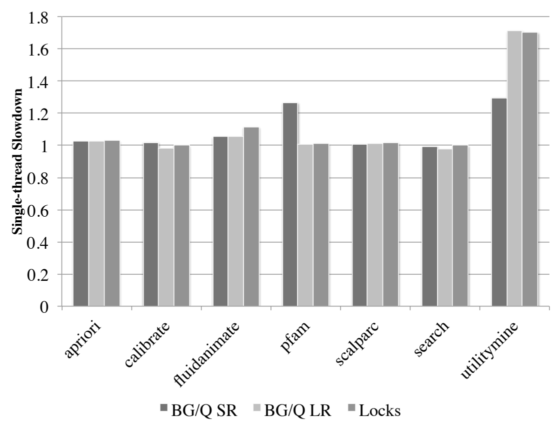

Figure A.9: Single-thread slowdown of the RMS-TM benchmarks.

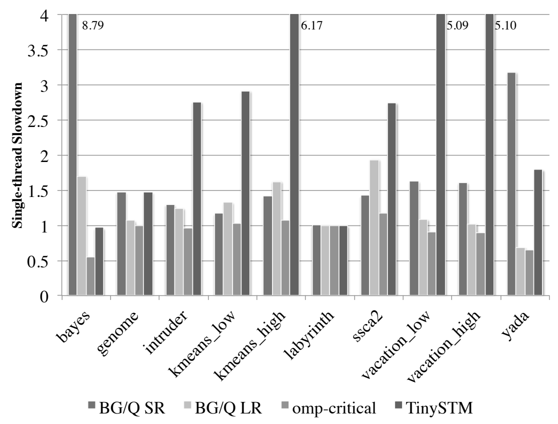

Figure A.10: Single-thread slowdown of the STAMP benchmarks.

A.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.

Figure A.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 Figure A.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 A.2).

Figure A.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 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.

A.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 A.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 A.2), which in turn causes a three-fold single-thread slowdown (Figure A.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.

A.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 (Figure A.6 and Figure A.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 A.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.

A.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

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 %

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

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 A.2, the instruction path length increase in `utilitymine`, `ssca2`, and `kmeans` is the result of this overhead.

A.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 Figure A.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`.

A.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 to scale well using locks. As shown in Figure A.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%.

A.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 A.3 shows the serialization ratio and the abort ratio (defined in Section A.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 A.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 only 128 spec-IDs and they are recycled periodically based on a pre-determined interval called the scrub interval. Figure A.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 Figure A.11, the scalability of both benchmarks improves significantly with a much lower scrub interval.

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

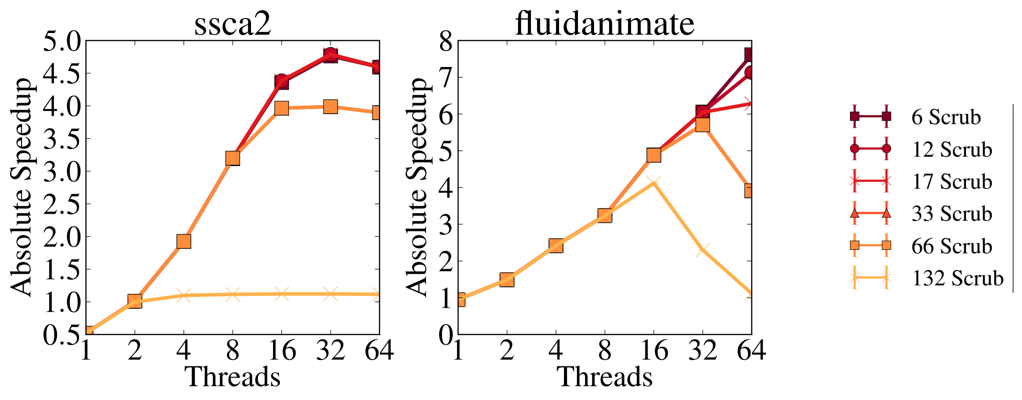

Figure A.11: Effect of varying scrub intervals for `ssca2`and `fluidanimate`

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

Table A.4: Basic features of real HTM implementations.

- **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 A.7.2). As a result, the performance of BG/Q TM is similar to that of locks and exhibits no scalability.

A.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`,

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

Table A.5: Average read- and write-set size (in words) of STAMP using TinySTM (1 thread).

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 A.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.

A.9 Related Work

Despite many HTM proposals in the literature for hardware support for transactional memory [13, 40, 50, 55, 67], only recently real HTM implementations became available. Besides the earlier Rock processor [26] and Vega Azul system [21], now we have Intel Haswell [46], the IBM zEC12 enterprise server [45], and IBM TM support for the POWER architecture [14]. While all are best-effort HTMs, their design points differ drastically. Table A.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 [26]. 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) [44], and could thus unburden the TM programmer from the need to reason about hardware limitations [46].

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 [65].

The implementation of HTM in the IBM zEC12 enterprise server uses the L1 and a modified MESI cache protocol to store speculative state [45]. In this machine both L1 and L2 use a write-through policy, thus the complexity of tracking dirty 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.

A.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.