What Scalable Programs Need from Transactional Memory

Donald Nguyen
Synthace Limited
d.nguyen@synthace.com

Keshav Pingali
The University of Texas at Austin
pingali@cs.utexas.edu

Abstract

Transactional memory (TM) has been the focus of numerous studies, and it is supported in processors such as the IBM Blue Gene/Q and Intel Haswell. Many studies have used the STAMP benchmark suite to evaluate their designs. However, the speedups obtained for the STAMP benchmarks on all TM systems we know of are quite limited; for example, with 64 threads on the IBM Blue Gene/Q, we observe a median speedup of 1.4X using the Blue Gene/Q hardware transactional memory (HTM), and a median speedup of 4.1X using a software transactional memory (STM).

What limits the performance of these benchmarks on TMs? In this paper, we argue that the problem lies with the programming model and data structures used to write them. To make this point, we articulate two principles that we believe must be embodied in any scalable program and argue that STAMP programs violate both of them. By modifying the STAMP programs to satisfy both principles, we produce a new set of programs that we call the Stampede suite. Its median speedup on the Blue Gene/Q is 8.0X when using an STM. The two principles also permit us to simplify the TM design. Using this new STM with the Stampede benchmarks, we obtain a median speedup of 17.7X with 64 threads on the Blue Gene/Q and 13.2X with 32 threads on an Intel Westmere system.

These results suggest that HTM and STM designs will benefit if more attention is paid to the division of labor between application programs, systems software, and hardware.

Keywords transactions; transactional memory; programming models; scalability; Stampede benchmarks

ASPLOS '17 April 8–12, 2017, Xi'an, China.

 ISBN 978-1-4503-4465-4/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3037697.3037750

1. Introduction

In the last decade, transactional memory (TM) has been the subject of numerous studies in both the architecture and parallel programming communities [1, 8, 9, 15, 16, 20, 21, 23, 24, 33, 45–48, 50]. More recently, several companies including Azul, Oracle, IBM and Intel have released hardware transactional memory (HTM) implementations [49, 51], and the C++ standards committee is considering adding language support for transactions [31].

One of the main arguments for the TM approach is that it simplifies the writing of parallel programs because it gives programmers the concurrency benefits of fine-grain locking without requiring them to write fine-grain locking code, which is usually difficult to debug, port and maintain. TM has been evaluated using microbenchmarks such as implementations of traditional data structures like stacks and trees, and benchmark suites such as the STAMP benchmarks [10], which use these data structures. In the original STAMP study, evaluation was done on a hardware simulator. However, evaluations on real hardware and software TMs show limited speedups. For example, when we ran the STAMP benchmarks on the IBM Blue Gene/Q using 64 threads and hardware transactional memory, we obtained a median speedup of only 1.4X. When we used TinySTM [21], a state-of-the-art STM, instead of HTM, we obtained a median speedup of 4.1X.

We believe it is important understand why TM benchmark suites like STAMP perform poorly on transactional memory implementations. Do the algorithms and data sets in these benchmarks have limited parallelism? If so, we need different benchmark suites. Is performance limited by the programming model and data structures used to write these benchmarks? If so, we need to re-implement these applications using better programming models and data structures since inventing elaborate mechanisms to speedup badly written programs is usually an exercise in futility. Is there some fundamental limitation in using TM to exploit parallelism even in well-written transactional programs? If so, we need to understand what these limitations are and whether they can be circumvented.

Understanding these issues is important because one of the main lessons learned from the RISC versus CISC debates of the 1980s and 1990s is that the design of high-

performance systems requires careful attention to the division of labor between application programs, systems software (such as compilers and runtime systems), and hardware. In particular, implementing a general programming construct directly in hardware or in systems software may be inefficient compared to specialized implementations for each occurrence of that construct, optimized for the context and structure of that occurrence.

In this paper, we argue that transactional programs can be scalable, and that lack of scalability in current TM benchmarks arises fundamentally from the programming models and data structures used in these benchmarks. To make this point, we articulate two design principles for writing parallel programs that we believe are essential for scalability. These principles arise from the fact that the most common factor limiting scalability is data movement. Communication adds costs that increase as the number of threads or packages increases, so the best way to improve the scalability of a program is to (1) reduce communication and (2) be tolerant of communication costs when they do arise. Concretely, this means the following.

Principle 1 (Disjoint accesses)**.** *Transactions that are disjoint at the logical level should be disjoint at the physical level.*

This principle is a guide for reducing or eliminating conflicts and data movement between transactions: it says that concurrent transactions should not conflict if they operate on disjoint data. For example, transactions that add different data items to a bag (multiset) should not conflict even though they operate on the same data structure. This requires careful attention to the implementation of the bag; an implementation that uses a single lock to control access to the entire bag violates this principle.

Principle 2 (Virtualized transactions)**.** *Transactions should be virtualized.*

Transactions are virtualized if their execution is not tied to a particular thread. This principle is a guide for ensuring that threads do useful work even in the presence of conflicts or communication delays; a thread can set aside a transaction and perform other work instead of having to retry an aborted transaction immediately or wait for a response to a remote memory request.

These principles may not be surprising,[1] but we show that the programming model used in the STAMP benchmark suite satisfies neither of them. One contribution of this paper is a new benchmark suite for transactional programs, called Stampede,[2] that is derived by systematically applying these scalability principles to the STAMP benchmark suite. These changes were not particularly difficult to implement, as we discuss in Section 6; roughly 90% of the application code is unchanged. We show that the performance of the Stampede benchmarks is substantially better than that of STAMP benchmarks with state-of-the-art HTMs and STMs.

Our second contribution is a demonstration that if application programs embody these principles, the design of STMs can be simplified. The simplification arises from the fact that sophisticated conflict detection schemes in TM are not necessary for scalable programs. The simplified STM, which we call VXTM, boosts performance of Stampede programs further.

We present an extensive experimental evaluation of both the STAMP and Stampede suites on TinySTM, VXTM, and the Blue Gene/Q HTM, as well as on a 32-thread Intel Westmere system. On the Westmere system for example, the Stampede version of the intruder benchmark with VXTM runs 168 times faster than the STAMP version on TinySTM; the yada benchmark improves by a factor of 62.

The rest of this paper is organized as follows. In Section 2, we describe the programming model used in current TM benchmarks like STAMP and argue that it violates the two scalability principles discussed above. In Section 3, we describe how to implement programs to conform to the disjoint access principle; to illustrate this, we show how this can be accomplished in the STAMP benchmarks by replacing the data structures, task schedulers, and the memory allocator. In Section 4, we describe how to implement programs to conform to the virtualized transaction principle; we illustrate this again by modifying the STAMP benchmarks. In Section 5, we explore how programs following these principles can benefit from simpler conflict detection and handling strategies and describe an implementation of these in VXTM. Experimental results are presented in Section 6, and related work is presented in Section 7.

2. STAMP Programming model

The phrases *transactions*, *transactional execution*, and *transactional memory* are often used interchangeably in the literature, but for the discussion in this paper, it is useful to distinguish between these concepts as follows.

Transactional execution is a property of the execution of a piece of code in a parallel program: a piece of code (call it T) is said to execute transactionally if its execution is *atomic* and *isolated* [25]. The precise definitions of these two concepts can be quite subtle, and there are also various flavors such as strong and weak atomicity, but roughly speaking, atomicity means that updates to shared state made by T become visible to other threads only at the end of the execution of T, and isolation means that state changes made by other concurrently executing threads after T begins execution are not visible to T while it executes.

Transactions are programming constructs that permit the programmer to delineate a piece of code and require that it be executed transactionally by the implementation of the pro-

[1] We have used them in the Galois system for almost a decade now as we describe in Section 6.6.

[2] The Stampede benchmarks can be accessed at `http://iss.ices.utexas.edu/`.

```
Worklist wl

void func(int threadId):
  while true:
    Task t = atomic { wl.pop() }
    if !t:
      break
    Task* newTs = atomic { work(t, threadId) }
    for Task nt in newTs:
      atomic { wl.push(nt) }

thread_run(func, numThreads)
```

Figure 1: STAMP programming model

gram. The notation *atomic* {T} is used in many parallel programming systems to specify that code T should be executed transactionally.

Transactional memory (TM) is a mechanism in hardware (HTM) or software (STM) to implement transactional execution. Unlike locks, which protect access to shared data but are not associated directly with that data, TM is *data-centric* in the sense that it tracks the memory locations accessed by each transaction; if the memory accesses made by concurrently executing transactions may violate transactional semantics, these transactions are said to *conflict*, and one or more of them is aborted and re-tried. If the transaction reaches the end of its execution and its accesses have not violated transactional semantics, it is committed and its updates to global state become visible to other computations.

In the STAMP suite, transactional execution is achieved by adding transactional constructs to an explicitly parallel threaded program. Figure 1 gives the outline of a typical STAMP benchmark. To ground the discussion, we use the yada benchmark, which performs Delaunay mesh refinement of a triangular mesh, using Ruppert's algorithm to eliminate badly shaped triangles [26]. The badness of a triangle can be determined by applying a simple geometric test to its vertices. The algorithm is driven by a worklist, which keeps track of bad triangles in the mesh. At the start of the algorithm, this worklist is initialized with all the bad triangles in the mesh. During the execution of the algorithm, new bad triangles may be created, and if so, they are added to the worklist. The algorithm terminates when the worklist becomes empty.

To process and eliminate a bad triangle, a small neighborhood of triangles surrounding the bad triangle is identified and deleted from the mesh (these triangles are said to lie in the *cavity* of the bad triangle). The region previously occupied by these triangles is then retriangulated. This series of steps is implemented by the function `work` in Figure 1. Then, any new bad triangles created by processing the current triangle are added to the worklist.

Parallelism in this algorithm arises from the fact that each cavity is usually a small region of the overall mesh, so bad triangles whose cavities do not overlap can be processed in parallel. However, it is difficult to tell a priori whether or not the cavities of two bad triangles will overlap, so static parallelization is not possible. Instead, one can mark the processing of each bad triangle as a transaction and execute transactions speculatively, leaving it to the TM system to detect and recover from conflicts on the fly.

To exploit this pattern of parallelism, the yada benchmark spawns some number of threads, each of which executes the function `func` in Figure 1. The body of this function repeatedly gets a bad triangle from the worklist, processes it, and adds any newly created bad triangles to the worklist. These three operations manipulate a shared data structure—either the mesh or worklist—so each operation is performed in a transaction.

The STAMP implementation of yada violates both principles for scalable parallel performance introduced in Section 1.

Principle 1: Worklist push and pop operations from different threads conflict because a linked-list is used to represent the worklist. At the logical level however, two threads that pop work items from the work list touch different data items, so they should not conflict according to the disjoint access principle. Similar considerations apply to the representation of the graph data structure: as long as cavities do not overlap, transactions should not conflict. However, the graph representation used in STAMP introduces spurious conflicts, violating the principle of disjoint access.

Principle 2: The explicitly threaded programming model of Figure 1 violates the principle of virtualized transactions. Once a transaction is attempted by a thread, it is not possible for the thread to put aside that transaction if it aborts and do some other work. In most implementations, the same thread will keep trying a transaction until it commits. This limits scheduling freedom and reduces effective processor utilization.

While this discussion has focused on yada, similar issues arise in the other benchmarks. In the next two sections, we show how to incrementally modify STAMP benchmarks to make them conform to the disjoint access and virtualized transactions principles. In Section 6, we show that these changes, together with a simpler STM enabled by the two principles, reduce the execution time of yada by a factor of 62 on a 32 core Intel Westmere system.

3. Enforcing the Disjoint Access Principle

The disjoint access principle requires that transactions accessing disjoint *logical* data structure elements should access disjoint *physical* memory locations. This is a general principle for scalability even for programs without transactions. For the STAMP benchmark suite, we achieve this by examining and changing the data structures used in each program. Although our data structure analysis is manual, the code changes required are small because in many cases we simply swap one abstract data type (ADT) implementation

	Original	Scalable Alternative
bayes	linked-list	workstealing scheduler
genome	hashtable	reduction of hashtables
kmeans	shared-counter	workstealing scheduler
intruder	dynamic buffer	per-iteration buffer
labyrinth	growable array	workstealing scheduler
ssca2		
vacation	red-black tree	hashtable
yada	eager root update; linked-list	lazy update; workstealing scheduler

Figure 2: Original STAMP data structures and their scalable alternative.

for another. Figure 2 shows the original STAMP data structures and their scalable alternatives. We describe some of the important changes below. There are three general techniques that we follow.

First, for a given functionality, we choose a scalable abstract data type. Supporting certain combinations of operations may be more scalable than others. For instance, a counter that supports concurrent modification and provides access to intermediate values requires more communication than a counter for which access to intermediate values is not needed: the second data structure can be implemented using a reduction tree while the former requires read-modify-write updates.

Second, we implement each abstract data type in a way that satisfies the disjoint access principle. Although there may be many concrete implementations of a given ADT, some are typically more scalable than others. For instance, a map can be implemented using either a red-black tree or a hashtable, but the hashtable is more scalable because operations on different keys typically access different portions of the hashtable. In contrast, accesses to different keys in a red-black tree will have overlapping accesses on shared paths from the root.

Third, we use a scalable memory allocator to make data structure implementations more efficient. Memory allocation is a common scalability bottleneck. Allocation of new pages may be serialized in the operating system, and techniques for recycling and coalescing memory regions requires communication between threads. To avoid contention overheads, we pre-allocate pages. For temporary objects that do not escape a transaction, we use a thread-local memory allocator. For other allocations, we use our own memory allocator which maintains thread-local, size-segregated free-lists that draw from a pre-allocated but expandable page pool.

These general approaches were applied to the STAMP benchmarks as follows.

In four applications (bayes, kmeans, labyrinth, yada), a sequential data structure with transactional operations is used to implement a work-stealing scheduler, but it is more efficient to use a data structure designed from the ground up to be a scheduler than to use a shared counter or linked-list. For these applications, we use the workstealing scheduler from the Galois system [30, 37].

```
foreach Item i in I:
  int v
  atomic { v = fn0(i) }
  atomic { fn1(i, v) }
}
```

(a) Original fine-grain transaction program.

```
struct Task { Item i; int v; int s; }

foreach Task t in T:
  atomic:
    switch t.s:
      case 0:
        t.v = fn0(t.i); t.s = 1; T.push(t);
        break
      case 1:
        fn1(t.i, t.v)
        break
```

(b) New loop-based transaction program.

Figure 3: Example of converting fine-grain transactions into loop-based transactions.

In genome, a hashtable is used to find duplicate strings. STAMP uses a single hashtable, while a more scalable implementation is to have per-thread hashtables and to merge the counts at the end using a reduction tree.

In intruder, dynamically allocated buffers are used to communicate values between processing pipeline stages. These allocations can be replaced with per-iteration allocation, which is highly scalable when downstream pipeline stages are nested in the current stage. The vacation application uses a red-black tree to maintain database relations, but a more scalable implementation of the same abstract data type is a hashtable.

Finally, the yada application maintains a root pointer to a mesh element to verify connectivity of the final mesh. Each transaction checks if it is removing the element referenced by the root pointer, and if so, it updates the pointer to another element. After parallel execution, the application verifies that all valid mesh elements are reachable from the root. Since this pointer is used only during verification, it is possible to simply search for a valid element to serve as root just before the verification step rather than eagerly updating it throughout parallel execution.

4. Applying the Virtualization Principle

The virtualized transaction principle requires that transactions be decoupled from threads and schedules. STAMP

```
void execute():
  for int i in range(begin, end):
    while true:
      task[i].execute()
      if !task[i].aborted:
        break
```

(a) Scheduling for unvirtualized transactions.

```
ThreadLocal<int> myId
TerminationDetection term
WorkstealingQueue q
Queue aborted[numThreads]

void execute():
  Task t
  while !term.allDone():
    if (t = q.pop())
        || (t = aborted[myId].pop()):
      term.notDone()
      t.execute()
      if t.aborted:
        aborted[myId].push(t)
      while (t = aborted[myId].pop()):
        t.execute()
        if t.aborted:
          aborted[myId/2].push(t)
    else:
      term.done()
```

(b) Improved scheduling for virtualized transactions using workstealing and serialization tree.

Figure 4: Scheduling virtualized transactions.

programs violate this principle in two ways. First, since STAMP programs use threads directly, transactions are tied to whichever thread issues them. Second, transactions assigned to a thread are processed in order, which unnecessarily restricts the space of possible schedules. In particular, a thread cannot execute a new transaction until its current transaction has committed.

As noted in Section 2, parallel programs can be written in a form that separates scheduling from the core computational operator. Given this form, the most natural granularity for a transaction is the complete execution of the operator or equivalently, an iteration of a parallel loop. We call these *loop-based transactions*. Loop-based transactions are naturally virtualized. The mapping of transactions to threads is implicit, and the iteration space of the loop specifies the entire set of transactions to execute.

In STAMP programs, transactions can appear anywhere, and in fact, one optimization is to reduce the granularity of transactions to be smaller than the computational operator. We call such transactions *fine-grain transactions*. Fine-grain transactions are akin to fine-grain locking. They can significantly improve performance by reducing transactional state, but they require sophisticated reasoning to ensure correct behavior. In contrast, loop-based transactions are easier to reason about. The behavior of a parallel loop with coarse-grain transactions is equivalent to executing iterations in some sequential order. They can also have performance benefits over fine-grain transactions. For instance, since transactions are now parallel tasks, many of the techniques used to schedule parallel tasks like *workstealing* [7] can be directly applied to schedule transactions.

Fine-grain transactions can be made into loop-based ones by applying a continuation-passing transformation to divide loop iterations into units matching the fine-grain transaction boundaries (see Figure 3). This transformation is a standard compiler technique [3, 32]; although for this work, we apply the transformation manually. These new coarse-grain transactions will have the same granularity as the original fine-grain ones, but unlike the originals, there is overhead from forming the continuation and a possible loss of data locality due the possible rescheduling of two previously sequentially composed transactions.

If expert parallel programmers still want to use fine-grain transactions, they can mimic most of their effect in loop-based code by annotating particular reads or writes in an iteration as protected by the transaction or not. In our programs, this is accomplished by a special API call before shared data accesses.

Virtualization provides an opportunity for rescheduling work to improve performance. Figure 4 shows two ways to execute transactions. The first scheduler (Figure 4a) uses static work assignment and immediately retries aborted transactions. This is roughly the behavior of all the transactions in STAMP programs. The second scheduler (Figure 4b) uses workstealing to initially distribute transactions and uses a *serialization tree* to guarantee forward progress by gradually serializing aborted transactions on fewer and fewer threads (this is the approach used in the Galois system).

The second scheduler is only possible when transactions are virtualized and is one of many schedulers that could be used. In the context of STMs, CAR-STM [19] previously investigated serializing aborted transactions on a different thread than the one that first issued the transaction; however, because TMs are layered on top of threaded programs, additional effort must be spent to preserve program semantics. Consider a thread that acquires a reentrant lock and then issues a transaction that acquires that lock again. Under TM semantics, the second acquire should always succeed but CAR-STM can cause it to fail.

5. Exclusive TM

TM systems ensure transactional execution by monitoring reads and writes to memory, and one challenge tackled by prior TM work is supporting a high amount of concurrency by sometimes allowing a read in one transaction to proceed in the presence of a concurrent read or write to the same address in another transaction.

In multicore architectures, writing to a shared cache line is a potential scalability bottleneck, and scalable programs are typically written (or rewritten) to not have such sharing [13]. Having a conflict detection scheme that permits transactions to write to a shared cache line enables a level of concurrency at the transaction level that is not scalable at the cache coherency level.

An alternative is to simply not permit transactions to proceed if they access the same memory locations. One such scheme is *exclusive locking*. Whenever a transaction reads or writes memory, it marks that location with its transaction id. If another transaction has already marked the location, the current transaction *aborts*, rolling back its state changes and clearing its marks. When a transaction finishes, it also clears its marks.

To clear its marks, a transaction walks a list of marked locations and sets the owner fields back to a special unacquired value. Rollback is similar to clearing marks except that the transaction additionally restores the original value. In many cases, a transaction reads all its locations before modifying any of them. In this case, there is no state to rollback when a conflict occurs because no writes have happened yet.

Exclusive locking *fails fast* on transactions that other conflict detection schemes would permit to continue. However, if a program follows the disjoint access and virtualization principles, the number of conflicts should be small because (1) mostly disjoint data is accessed and (2) aborted transactions can be rescheduled to reduce conflicts. Exclusive locking also reflects the realities of modern hardware. Writes to shared cache lines are scalability bottlenecks. Exclusive locking reflects the performance model of the underlying hardware directly in the programming model. This makes potential problems more obvious, which encourages programmers to address them early.

An objection to exclusive locking is that it produces conflicts with read-only workloads. If this is a serious issue for a program, compiler analysis to find read-only locations can be used [28]. For situations where a location is sometimes read and sometimes written, TMs that allow concurrent reads typically use timestamp-based validation [16, 41]. This has its own scalability cost because a global counter is atomically updated by each thread on commit. The cost of synchronization can be eliminated by using a hardware clock [42] or hardware performance counters [43], but in these cases, transactions sometimes must wait to commit. One could instead turn to thread-local clocks [4], but that comes at the price of false conflicts.

6. Evaluation

Section 6.1 describes the STMs and HTMs used in the evaluation as well as the details of how the Stampede benchmarks were produced from the STAMP benchmarks. Section 6.2 compares the sequential performance of the STAMP and Stampede programs. We show that, for all but one benchmark, the serial performance of the STAMP and Stampede versions is similar; therefore, any differences in the parallel performance can be attributed to differences in parallel behavior rather than, say, algorithmic differences. Sections 6.3 and 6.4 show the performance of the STAMP and Stampede benchmarks on STMs and on the Blue Gene/Q HTM respectively. Section 6.5 analyzes these performance results in more depth.

6.1 Experimental setup

STM and HTM: For our STM, we choose TinySTM [21] (v1.0.4), which has been used in a previous TM evaluation [49]. TinySTM supports several policies for conflict detection and resolution. We use its default configuration, which is encounter-time locking with write-back of transactional state on commit. For the HTM, we used the IBM Blue Gene/Q.

Generating the Stampede banchmarks: Starting with the STAMP benchmark suite (v0.9.10), we modify each program according to the disjoint access and virtualization principles to produce the Stampede benchmark suite. To give a sense of the change from STAMP to Stampede, STAMP has 41586 lines of code, whereas Stampede has 43563 lines of code of which 38677 lines (88%) are exactly the same as STAMP. The bulk of the changes are due to changing the way STAMP data structures iterate through elements to support the Stampede memory allocator and moving code to conform to loop-based transactions. Of the changes, only 326 lines are due to new code; in this case, these are new data structures that do not have any analogue in the original STAMP code. We believe that this is a reasonable amount of effort as even applying TM to an existing threaded program with the aid of a compiler requires some level human intervention and significant non-local changes [44].

To run Stampede with HTM, we mark loop bodies with compiler pragmas to indicate a transaction. The transaction only includes the loop body itself and does not include scheduling code. To run with STM (Stampede+STM), we mark transactions boundaries as in the HTM case. We also annotate shared values as transactional variables and write the currently executing thread id to an owner field whenever we need to write a value, relying on the underlying STM system to detect conflicts.

Exploiting virtualized transactions: Although Stampede programs have virtualized transactions, Stampede with HTM or STM will behave as if transactions are not virtualized because the underlying TM is not aware of the virtualization, and it will execute transactions as in Figure 4a. To measure the impact of virtualization, we modify Stampede transactions to abort if they are re-executed. The aborted transactions are placed on a serialization tree and processed according Figure 4b. We call this variant Stampede+VTM. However, a similar modification could not be applied to Stam-

Application	Xeon Westmere Time	Xeon Westmere Speedup	Blue Gene/Q Time	Blue Gene/Q Speedup
bayes	7.07	1.94	67.79	2.58
genome	6.84	1.23	60.10	0.98
intruder	1.81	15.53	43.46	2.31
kmeans-high	2.78	0.96	14.78	0.84
kmeans-low	15.42	0.98	78.26	0.93
labyrinth	63.61	1.21	524.00	1.09
ssca2	7.82	0.89	29.99	0.91
vacation-high	25.19	1.03	70.67	1.26
vacation-low	19.39	1.02	51.77	1.23
yada	7.05	1.74	54.77	1.87

Figure 5: Sequential performance of Stampede+VXTM programs: execution time (seconds) and speedup over sequential STAMP version

pede+HTM because the HTM provides strong atomicity and does not indicate if a transaction aborts.

Finally, Stampede+VXTM schedules virtualized transactions using workstealing and uses a serialization tree like Stampede+VTM, but instead of using an off-the-shelf STM, it uses the exclusive TM implementation described in Section 5.

Machines: We run each application with its largest input on two machines: an Intel Xeon E7-4860 (Westmere) machine, which has four 10-core processors[3], and a single Blue Gene/Q compute node. The Blue Gene/Q compute node has a single processor with sixteen cores. Each core has four hardware execution contexts, which gives a total of 64 execution contexts per node. The Westmere does not support hardware transactional memory, but the Blue Gene/Q does.

On the Blue Gene/Q, programs are compiled with the IBM XLC compiler (v12.1) with the following compiler options: `-O -qsmp=noopt`[4] `-qalias=noansi`. On the Westmere, programs are compiled with GCC (v4.8.1) with `-O3`.

6.2 Sequential performance

Figure 5 gives the running times of Stampede+VXTM programs on one thread and the speedup of these programs relative to single-threaded execution of the corresponding STAMP programs with TinySTM. For most applications, the performance of the Stampede+VXTM version is close to that of the corresponding STAMP program. The only exception is intruder; the Stampede version has significantly better locality than the pipeline-based STAMP version.

[3] Although this machine has 40 cores, our experiments only run with threads in powers of two due to a restriction in the STAMP benchmark suite

[4] This option disables auto-parallelization in the XLC compiler; the other standard compiler optimizations are not affected.

These experiments establish that significant differences, if any, in the *parallel* performance of the STAMP and Stampede benchmarks arise primarily from the parallel behavior of these programs rather than from algorithmic differences.

6.3 Parallel performance on STMs

Figure 6a and Figure 6b shows performance results for STAMP and Stampede programs with the STM variants on the Westmere and the IBM Blue Gene/Q respectively. All speedups are relative to the STAMP sequential baseline for the application, shown in Figure 5. Each graph has four lines: (1) STAMP+(Tiny)STM, (2) Stampede+(Tiny)STM, (3) Stampede+VTM, and (4) Stampede+VXTM.

The first high-level observation is that Stampede programs outperform STAMP+STM programs substantially, and Stampede+VXTM has the fastest overall performance. On the Westmere, the median speedup with 32 threads is 2.3X for STAMP+STM, 6.0X for Stampede+STM and 13.2X for Stampede+VXTM. On the Blue Gene/Q, the median speedup with 32 threads (64 threads) is 3.6X (4.1X) for STAMP+STM, 6.8X (8.0X) for Stampede+STM and 12.8X (17.7X) for Stampede+VXTM.

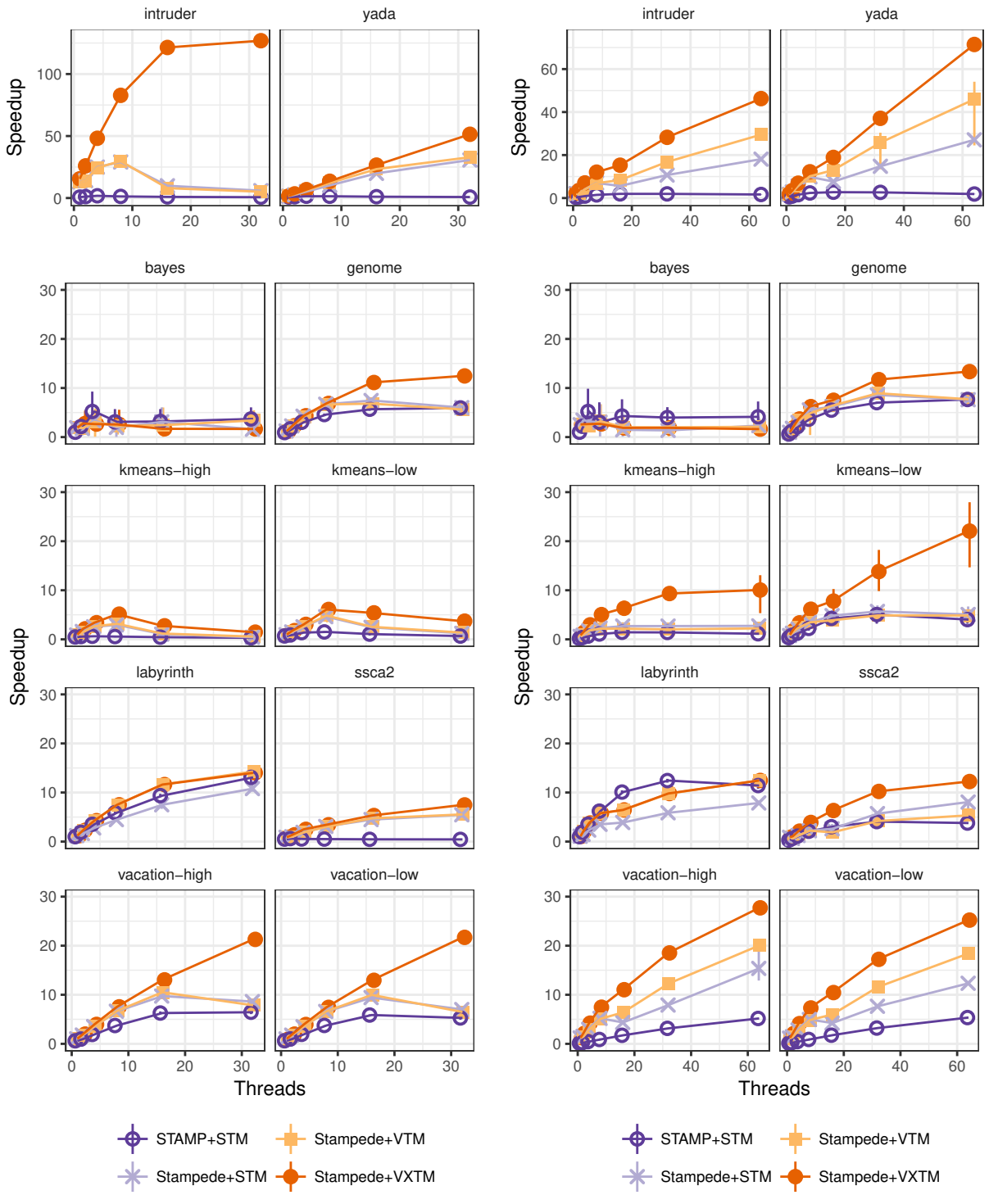
Two exceptions to these trends are bayes and labyrinth. The behavior of bayes is highly variable and none of its versions scales well, but the mean STAMP+STM time is slightly better than the Stampede time across threads. Section 6.5 examines this effect further, but we note that bayes has fallen out of favor in the TM community because of the high variability in its execution.

When comparing across machines, programs on the Blue Gene/Q tend to scale better than on the Westmere. Vacation-high and vacation-low are examples of this behavior. The Blue Gene/Q machine is a single-chip processor, while the Westmere is a multi-socket system. Modest communication in the STM or program itself can be tolerated at scale on the Blue Gene/Q, while similar communication patterns can be a bottleneck on the Westmere. Only when all the bottlenecks are removed, as is the case with vacation with Stampede+VXTM, does the program become scalable on the Westmere.

For labyrinth on the Westmere, STAMP+STM is faster than Stampede+STM but slower than Stampede+VXTM; on the Blue Gene/Q, it is faster than Stampede+VXTM for low number of threads. The main reason is that the STAMP program implements a lazy transaction validation scheme that is not available in the Stampede program.

Performance of STAMP on XTM The next set of results show that using exclusive TM (XTM) for programs written in a programming model that does not enforce the principles of disjoint access and virtualized transactions can lead to substantially worse performance.

Figure 7 shows the parallel performance of STAMP programs on XTM, compared to their performance on TinySTM. In the XTM runs, a transaction that aborts is im-



(a) Xeon Westmere

(b) Blue Gene/Q

Figure 6: Comparison of programs on two architectures using improved data structures and scheduling (Stampede) and original STAMP programs. Speedup relative to STAMP sequential baseline (see Figure 5). Points at mean value of at least 5 runs. Vertical bars indicate minimum and maximum observed values.

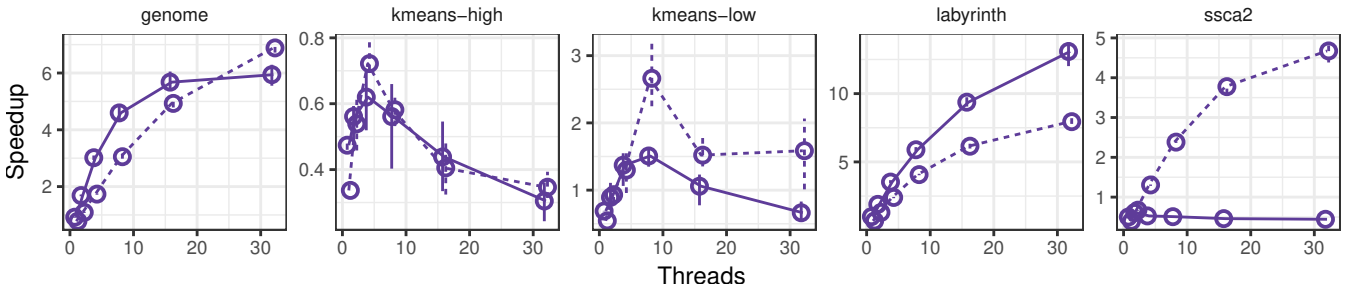


Figure 7: Performance of STAMP programs with STM (solid line) and with XTM (dotted line). Speedup relative to STAMP sequential baseline (see Figure 5).

mediately retried by the thread executing it, as is the norm in TM systems. On the XTM system, conflicts between concurrently executing transactions are more likely, but the overhead of conflict detection is lower. Therefore, programs with low conflict ratios might be expected to do well on the XTM; conversely, programs with high conflict ratios on the STM will perform poorly on XTM.

On two of the ten benchmarks (kmeans-low and ssca2), STAMP+XTM performs as well as or better than STM. These are instances where the benefit of lower overhead conflict detection outweighs the need for higher concurrency. On three benchmarks (i.e., genome, kmeans-high, labyrinth), STAMP+XTM is worse than STAMP+STM.

On the other five benchmarks not shown in the figure (bayes, intruder, vacation-high/low, yada), the performance of STAMP+XTM is much worse. In these benchmarks, transactions tend to read and write the same data structure, and the number of aborted transactions grows very large, slowing the overall execution and introducing livelock.

In contrast to these results, Figure 6 shows that XTM can provide a substantial boost in performance for Stampede programs. Virtualization alone has some benefits, and Stampede+VTM typically performs better than Stampede+STM. On the Westmere, the difference is minor, but on the Blue Gene/Q, intruder, yada, vacation-high and vacation-low see significant performance gains. However, virtualization should be seen as an *enabling transformation* that facilitates other optimizations like rescheduling and exclusive conflict detection. Neither virtualization nor XTM alone is adequate to address the scalability problem of transactional programs: co-design of the programming model and software/hardware TM is needed. We elaborate on this in Section 6.5.

6.4 Parallel performance on HTMs

Figure 8 shows performance of STAMP and Stampede programs on the IBM Blue Gene/Q, which has an HTM. With 32 threads (64 threads), the median speedup of STAMP with HTM is 1.4X (1.4X) and of Stampede with HTM is 6.1X (4.7X).

A comparison of STAMP programs with STM and HTM on the Blue Gene/Q was previously done by Wang et al. [49].

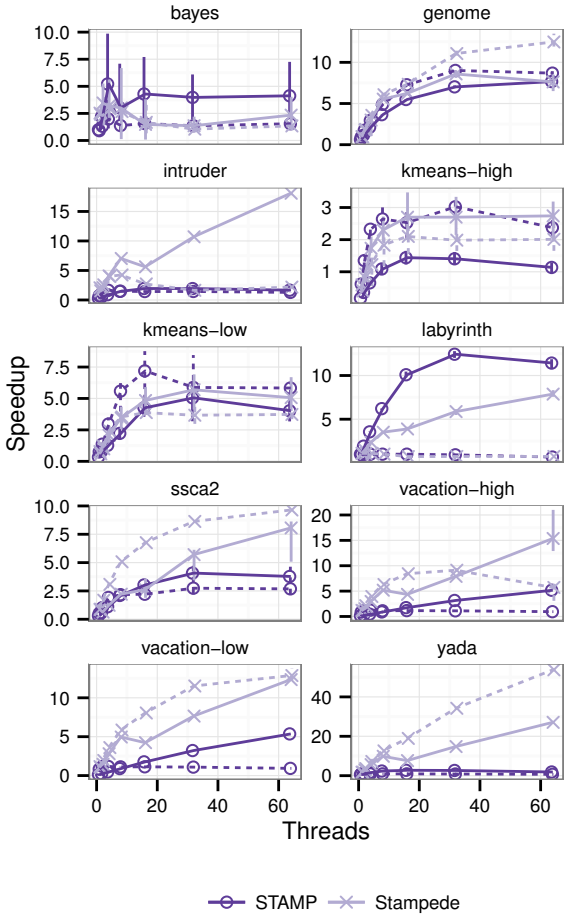


Figure 8: Speedup of STAMP and Stampede programs with STM (solid lines) and with HTM (dotted lines) on the Blue Gene/Q. Speedup relative to STAMP sequential baseline (see Figure 5).

Our results for the HTM and STM programs broadly match their reported results with one exception. Wang et al. report a maximum speedup of 12X for vacation-low and 16X for vacation-high.

The performance of HTM on vacation-high (and on vacation-low) strongly depends on the memory allocator used. The results in Figure 8 use standard malloc while the results of Wang et al. use a different memory allocator that preallocates thread-local pools but does not track freed memory, causing a memory leak. The similarity of results between the two different evaluations suggests that improving memory allocation by itself does not substantially impact performance. When we switched the ad-hoc pooled allocator for TCMalloc [22], a scalable malloc replacement, only the yada benchmark was significantly improved (maximum speedup of 12X on the Blue Gene/Q) compared to using standard malloc.

For STAMP programs, STAMP+HTM is faster than STAMP+STM for three out of ten benchmarks (genome, kmeans-high and kmeans-low). For most STAMP benchmarks, the performance difference between STAMP+HTM and STAMP+STM is not more than a factor of two. The exception is labyrinth where the large working set exhausts the transactional state of the hardware, which then serializes execution. For Stampede programs, Stampede+HTM is faster than Stampede+STM for four benchmarks (genome, ssca2, vacation-low and yada).

The existence of benchmarks with a large difference between STAMP+HTM and Stampede+HTM performance like ssca2, vacation-high, vacation-low and yada show that some of the poor performance with HTM can be alleviated by starting with more scalable programs.

6.5 The need for co-design

The experimental results demonstrate the advantage of co-design of transactional programming models and transactional memory mechanisms. In general, there are three reasons why one conflict detection implementation performs differently than another on the same application.

1. The schedule produced by one implementation creates more algorithmic work than another implementation. This can happen in the bayes application because depending on the scheduling, different operations will be attempted, which changes the total amount of work done.
2. Assuming that the total amount of work is the same between program executions, the number of conflicts can vary due to different definitions of a conflict (e.g., word or cache line-based detection, or the exclusive-access policy implemented by XTM).
3. Finally, even if one implementation produces less conflicts than another, the cost of implementing more precise conflict detection may outweigh the benefit of the lower number of conflicts.

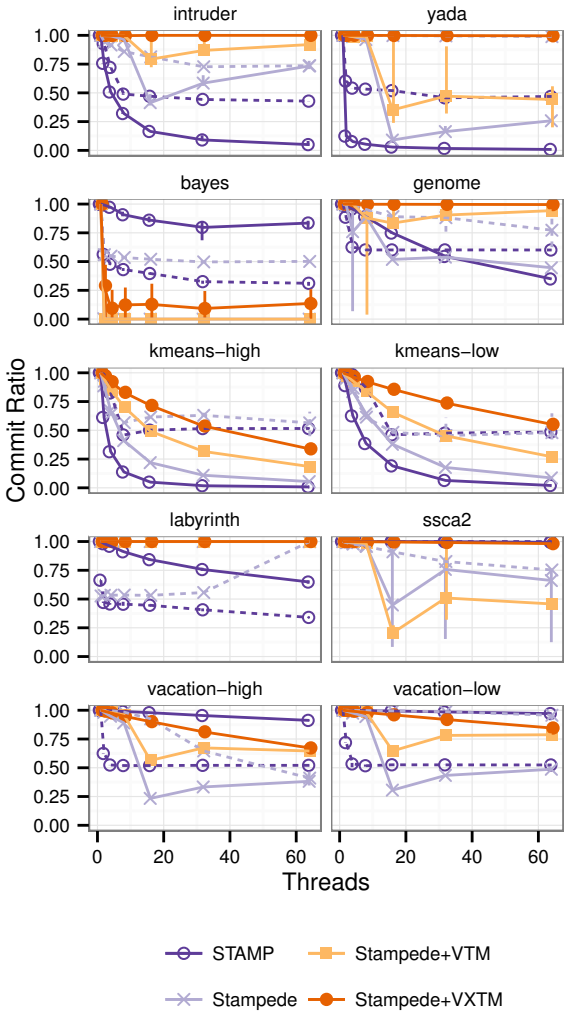


Figure 9: Commit ratios of STAMP+STM (solid line), STAMP+HTM (dotted line), Stampede+STM (solid line), Stampede+HTM (dotted line), Stampede+VTM and Stampede+VXTM programs on Blue Gene/Q.

The first reason is strongly dependent on the application, but the latter two reasons are due to general properties of conflict detection systems and represent trade-offs that must be made by all TM systems.

Stampede+VXTM allows only exclusive access to locations, so even shared reads in concurrent transactions will trigger conflicts. One might expect therefore that Stampede+VXTM programs will have a much lower commit ratio than Stamp+STM programs. Figure 9 shows that this is not the case: Stampede+VXTM programs have a much higher commit ratio for all but three benchmarks (bayes, vacation-high, vacation-low). Intuitively, this is because Stampede programs embody Principle 1 (disjoint accesses) so transactions in Stampede programs are less likely to conflict, regardless of the conflict detection policy in the TM.

Even with the (less precise) exclusive conflict detection in VXTM, the commit ratio of Stampede+VXTM programs exceeds that of Stampede+HTM for seven benchmarks (intruder, yada, genome, kmeans-low, labyrinth, ssca2, vacation-high). This suggests that capacity or false conflicts are inhibiting the performance of HTM programs. The Blue Gene/Q HTM detects conflicts at cache line granularity, and this may result in *false conflicts* when state from two transactions shares the same cache line. HTM systems also have a fixed amount of storage for transactional state, and exceeding that limit is treated as a conflict. This is exacerbated by the fact that in the Blue Gene/Q HTM, all memory accesses within a transaction are considered part of the transactional state. This explains the low commit ratio for the Stampede+HTM version of labyrinth: labyrinth has a large working set that exhausts the capacity of the HTM (the flattening of the commit ratio around 50% for Stampede+HTM is because during high conflict situations, the HTM runs a transaction once speculatively and on abort, runs it in "irrevocable mode", which gives an expected 50% commit ratio).

With respect to STMs, note that Stampede+VXTM, which has a lower overhead conflict detection scheme than Stampede+VTM, also has a better commit ratio. This is because Stampede+VXTM detects conflicts at the memory word granularity rather than at the cache-line level as TinySTM does.

However, commit ratios tell only one part of the story; the second part of the performance story is the overhead of conflict detection. The cost of implementing conflict detection in Stampede+VXTM is quite small since it uses exclusive locking. General TM implementations however are more complicated because they support state rollback and more precise read-write conflicts. For instance, to support state rollback, the default configuration of TinySTM uses a write-back cache, which means that accesses of modified state must go through a level of indirection. To support lightweight read transactions, TinySTM uses a time-based mechanism that requires shared access to a global clock. The Blue Gene/Q HTM implementation stores transaction metadata in the L2 cache, and transactions must either bypass or flush the L1 cache to ensure consistent updates of metadata [49]. Even when the commit ratio of (say) Stampede+HTM is equal to or greater than Stampede+VXTM, like for bayes, kmeans-high and vacation-low, we see that the end-to-end performance is worse than Stampede+VXTM, suggesting that the cost of implementing more precise conflicts outweighs the benefit of having less conflicts.

6.6 Other benchmarks and systems

We have applied these scalability principles to other benchmarks as well.

The PARSEC suite [6] (v2.1) is a collection of parallel programs that cover a number of parallelization techniques However, none of the programs use TM, and as reported by others, most of the programs are data parallel [5]. One that is not is canneal, which simulates cache-aware annealing to optimize routing costs. We evaluated three variants of this program. The first is the original PARSEC program that uses fine-grain locking, PARSEC+FGL. The second is a manual transformation to use TinySTM instead of locks, PARSEC+STM. The third is a new program that follows the disjoint access and virtualization principles and uses exclusive locking, NewPARSEC+VXTM. To satisfy disjoint access, the only change made was to allocate the main graph data structure in a NUMA-interleaved fashion to avoid congestion on physical memory controllers. Satisfying virtualization was simply a matter of converting explicit threading to parallel loops. On the Westmere with 32 threads and using the native input size, PARSEC+FGL is 22.3X faster than with 1 thread. On 32 threads, PARSEC+STM is 7% faster than PARSEC+FGL, and NewPARSEC+VXTM is 16% faster. These results support the applicability of our scalability principles beyond STAMP and programs initially written to use transactions.

We have also used the two scalability principles discussed in Section 1 in the Galois system for almost a decade now [27, 29, 36, 37]. Galois supports a high-level parallel programming model called the operator formulation of algorithms [40], implemented as a pattern language in sequential C++. The Galois system produces code for distributed, heterogeneous architectures including clusters of CPUs, GPUs, and FPGAs, and it supports both optimistic parallelization as well as round-based execution based on interference graphs. Optimistic parallelization is supported by a conflict detection mechanism very similar to that of VXTM. The data structures and schedulers in the Galois library are designed carefully from the ground up to respect the disjoint access principle; a case study for the priority scheduler in Galois is described in Lenharth et al. [30]. The execution model supports virtualized transactional execution. Experimental studies show that a variety of applications including Delaunay mesh generation, Delaunay mesh refinement (yada is derived from this benchmark), the Barnes-Hut n-body method, and motif detection in graphs scale to hundreds of cores on large-scale NUMA shared-memory machines like the SGI Ultraviolet [29], demonstrating the power of the two scalability principles discussed in this paper. The same principles have been used in GPU implementations of the Galois model [34, 35], such as the one supported by the IrGL compiler [39].

7. Related Work

Transactional memory is a broad topic, so we confine the discussion of related work to other studies of TM benchmarks like STAMP.

The original STAMP paper [10] included simulation results but not results on real hardware.

There have been several performance evaluations of hardware transactional memory [14, 15, 17, 18, 49, 51], and as

noted in Section 6.4, part of our evaluation reproduces the results of Wang et al. Intel has released HTM support in the Intel Core i7 (Haswell) processors. There are several evaluations of the Haswell HTM [18, 51]. The hardware is targeted towards short-running, fine-grain transaction programs, which are not the kinds of transactions that arise in STAMP programs. The results of Yoo et al. on STAMP programs using four threads show abort rates significantly higher than using an STM, which suggests a large number of capacity or false conflicts even at low levels of parallelism. The highest speedup over sequential reported by Diegues et al. for HTM is 3.5X on 8 threads for kmeans. The goal of this paper is not to evaluate HTM implementations but to show how simple mechanisms are sufficient to efficiently execute many transactional programs.

SwissTM [20] has been shown to perform slightly better than TinySTM, which was used as a representative STM in Section 6, but the STAMP benchmarks where SwissTM outperforms TinySTM the most (by 1.2X–1.5X)—intruder, kmeans-high, yada—also have poor scalability, so the actual performance difference is small.

There are several proposals for parallel programming models that support transactional execution [2, 11, 12, 38], but in these, transactions are fine-grain, and no separation is made between schedulers and user data structures, which is crucial to simplifying transaction implementation. Automatic Mutual Exclusion (AME) [1] introduced a higher level abstraction for transactional execution where code blocks are transactional by default. This matches the spirit of loop-based transactions described here, but unlike this paper, the work on AME is mainly concerned with semantics rather than performance.

8. Conclusion

Transactional memory provides a solution to a diverse set of problems on multicore machines such as providing easy-to-understand semantics for concurrent accesses to shared data, achieving fine-grain synchronization without using locks, extracting concurrency from legacy code, and facilitating the parallelization of programs with unknown data dependencies, but providing efficient implementations of TM remains a challenging research problem.

In this paper, we showed that some of the performance problems of transactional workloads like the STAMP benchmarks arise from the programming model and data structures used in the STAMP benchmarks, and that these problems are not intrinsic to the algorithms and data sets underlying them. To make this point, we articulated two principles that we believe must be embodied by scalable programs and argued that STAMP programs violate both of them.

We then incrementally modified STAMP programs to satisfy both principles, producing a new set of programs that we call the Stampede suite. The two principles also permit us to simplify the design of the STM through simpler conflict detection policies. Using this new STM with the Stampede suite, we obtain a median speedup of 17.7X with 64 threads on the Blue Gene/Q and 13.2X with 32 threads on an Intel Westmere system. On 32 thread Westmere runs, these changes reduce the execution time of the intruder benchmark by a factor of 168 and of yada by a factor of 62 compared to the original STAMP programs with a state-of-the-art STM.

Existing TM benchmarks represent code patterns in legacy code and have their place in the sun for evaluating TM systems, but we believe the Stampede benchmark suite, which embodies software principles for scalable parallelism, is a useful and necessary addition to the collection of TM benchmarks. More generally, we believe HTM and STM designs will benefit if careful attention is paid to the division of labor between application programs, systems software and hardware, a lesson similar to the one learned about processor design from the RISC/CISC debates of the 1980s and 1990s.

Acknowledgments

This research was supported by DARPA BRASS contract 750-16-2-0004 and NSF grants 1337281, 1406355, and 1618425. Sandia Labs supported Donald Nguyen with a fellowship and gave access to the IBM Blue Gene/Q used in this study. We are grateful to members of the ISS group at the University of Texas at Austin for discussions and for their contributions to the systems used in this study and to Amber Hassaan for his detailed feedback on this paper.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. *ACM Trans. Programming Language and Systems*, 33(1):2:1–2:50, Jan. 2011. doi: 10.1145/1889997.1889999.

[2] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, PLDI, pages 26–37, 2006. doi: 10.1145/1133981.1133985.

[3] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 2007.

[4] H. Avni and N. Shavit. Maintaining consistent transactional states without a global clock. In *Proc. Intl Colloq. Structural Information and Communication Complexity*, pages 131–140, 2008. doi: 10.1007/978-3-540-69355-0_12.

[5] M. J. Best, S. Mottishaw, C. Mustard, M. Roth, A. Fedorova, and A. Brownsword. Synchronization via scheduling: Techniques for efficiently managing shared state. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, PLDI, pages 640–652, 2011. doi: 10.1145/1993498.1993573.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proc. Intl Conf. Parallel Architectures and*

Compilation Techniques, PACT, pages 72–81, 2008. doi: 10.1145/1454115.1454128.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Notices*, 30(8):207–216, 1995. doi: 10.1145/209937.209958.

[8] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proc. Intl Symp. Computer Architecture*, ISCA, pages 24–34, 2007. doi: 10.1145/1250662.1250667.

[9] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood. TokenTM: Efficient execution of large transactions with hardware transactional memory. In *Proc. Intl Symp. Computer Architecture*, ISCA, pages 127–138, 2008. doi: 10.1109/ISCA.2008.24.

[10] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proc. IEEE Intl Symp. Workload Characterization*, IISWC, Sept. 2008.

[11] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, PLDI, pages 1–13, 2006. doi: 10.1145/1133981.1133983.

[12] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, Aug. 2007. doi: 10.1177/1094342007078442.

[13] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proc. ACM Symp. Operating Systems Principles*, SOSP, pages 1–17, 2013. doi: 10.1145/2517349.2522712.

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

[15] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: a case study in the effectiveness of best effort hardware transactional memory. In *Proc. Intl Conf. Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 39–52, 2011. doi: 10.1145/1950365.1950373.

[16] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. Intl Conf. Distributed Computing*, pages 194–208, 2006. doi: 10.1007/11864219_14.

[17] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proc. Intl Conf. Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 157–168, 2009. doi: 10.1145/1508244.1508263.

[18] N. Diegues, P. Romano, and L. Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proc. Intl Conf. Parallel Architectures and Compilation*, PACT, pages 3–14, 2014. doi: 10.1145/2628071.2628080.

[19] S. Dolev, D. Hendler, and A. Suissa. CAR-STM: Scheduling-based collision avoidance and resolution for software transactional memory. In *Proc. ACM Symp. Principles of Distributed Computing*, PODC, pages 125–134, 2008. doi: 10.1145/1400751.1400769.

[20] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, PLDI, pages 155–165, 2009. doi: 10.1145/1542476.1542494.

[21] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, PPoPP, pages 237–246, 2008. doi: 10.1145/1345206.1345241.

[22] S. Ghemawat and P. Menage. TCMalloc: Thread-caching malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`, 2014.

[23] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proc. ACM SIGPLAN Conf. Object-oriented Programing, Systems, Languages and Applications*, OOPSLA, pages 388–402, New York, NY, USA, 2003. doi: 10.1145/949305.949340.

[24] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proc. Intl Symp. Computer Architecture*, ISCA, 1993. doi: 10.1145/165123.165164.

[25] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008. ISBN 0123705916.

[26] M. Kulkarni, L. P. Chew, and K. Pingali. Using transactions in Delaunay mesh generation. In *Proc. Workshop on Transactional Memory Workloads*, WTW, 2006.

[27] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, PLDI, pages 211–222, 2007. doi: 10.1145/1250734.1250759.

[28] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, PLDI, pages 278–289, 2007. doi: 10.1145/1250734.1250766.

[29] A. Lenharth and K. Pingali. Scaling runtimes for irregular algorithms to large-scale NUMA systems. *Computer*, 48(8):35–44, 2015. doi: 10.1109/MC.2015.229.

[30] A. Lenharth, D. Nguyen, and K. Pingali. Priority queues are not good concurrent priority schedulers. In *Proc. European Conf. Parallel Processing*, pages 209–221, 2015.

[31] V. Luchangco, M. Wong, H. Boehm, J. Gottschlich, J. Maurer, P. McKenney, M. Michael, M. Moir, T. Riegel, M. Scott, T. Shpeisman, and M. Spear. Transactional memory support for C++. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3919.pdf`, Feb. 2014.

[32] V. Menon and K. Pingali. High-level semantic optimization of numerical codes. In *Proc. Intl Conf. Supercomputing*, ICS, pages 434–443, 1999. doi: 10.1145/305138.305230.

[33] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proc. Intl Symp. Computer Architecture*, ISCA, pages 69–80, 2007. doi: 10.1145/1250662.1250673.

[34] R. Nasre, M. Burtscher, and K. Pingali. Data-driven versus topology-driven irregular computations on GPUs. In *Proc. IEEE Intl Symp. Parallel and Distributed Processing*, pages 463–474, 2013.

[35] R. Nasre, M. Burtscher, and K. Pingali. Morph algorithms on GPUs. In *ACM SIGPLAN Notices*, volume 48, pages 147–156, 2013.

[36] D. Nguyen and K. Pingali. Synthesizing concurrent schedulers for irregular algorithms. In *Proc. Intl Conf. Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 333–344, 2011. doi: 10.1145/1950365.1950404.

[37] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proc. ACM Symp. Operating Systems Principles*, SOSP, pages 456–471, New York, NY, USA, 2013. doi: 10.1145/2517349.2522739.

[38] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for C/C++. In *Proc. ACM SIGPLAN Intl. Conf. Object-oriented Programming Systems Languages and Applications*, OOPSLA, pages 195–212, 2008. doi: 10.1145/1449764.1449780.

[39] S. Pai and K. Pingali. A compiler for throughput optimization of graph algorithms on GPUs. In *Proc. ACM SIGPLAN Intl Conf. Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, pages 1–19, 2016. doi: 10.1145/2983990.2984015.

[40] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, PLDI, pages 12–25, 2011. doi: 10.1145/1993498.1993501.

[41] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proc. Intl Conf. on Distributed Computing*, DISC, pages 284–298, 2006. doi: 10.1007/11864219_20.

[42] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, SPAA, pages 221–228, 2007. doi: 10.1145/1248377.1248415.

[43] W. Ruan, Y. Liu, and M. Spear. Boosting timestamp-based transactional memory by exploiting hardware cycle counters. *ACM Trans. Archit. Code Optim.*, 10(4):40:1–40:21, Dec. 2013. doi: 10.1145/2541228.2555297.

[44] W. Ruan, T. Vyas, Y. Liu, and M. Spear. Transactionalizing legacy code: An experience report using gcc and memcached. In *Proc. Intl Conf. Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 399–412, 2014. doi: 10.1145/2541940.2541960.

[45] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, PPoPP, pages 187–197, 2006. doi: 10.1145/1122971.1123001.

[46] A. Shriraman, M. F. Spear, H. Hossain, V. J. Marathe, S. Dwarkadas, and M. L. Scott. An integrated hardware-software approach to flexible transactional memory. In *Proc. Intl Symp. Computer Architecture*, ISCA, pages 104–115, 2007. doi: 10.1145/1250662.1250676.

[47] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. In *Proc. Symp. Parallelism in Algorithms and Architectures*, SPAA, pages 275–284, 2008. doi: 10.1145/1378533.1378583.

[48] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: eager-lazy hardware transactional memory. In *Proc. IEEE/ACM Intl Symp. Microarchitecture*, MICRO, pages 145–155, 2009. doi: 10.1145/1669112.1669132.

[49] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proc. Intl Conf. Parallel Architectures and Compilation Techniques*, PACT, pages 127–136, 2012. doi: 10.1145/2370816.2370836.

[50] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proc. IEEE Intl Symp. High Performance Computer Architecture*, HPCA, pages 261–272, 2007. doi: 10.1109/HPCA.2007.346204.

[51] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *Proc. Intl Conf. for High Performance Computing, Networking, Storage and Analysis*, SC, pages 19:1–19:11, 2013. doi: 10.1145/2503210.2503232.