Performance Evaluation of Intel® Transactional Synchronization Extensions for High-Performance Computing

Richard M. Yoo[†]
richard.m.yoo@intel.com

Christopher J. Hughes[†]
christopher.j.hughes@intel.com

Konrad Lai[‡]
konrad.lai@intel.com

Ravi Rajwar[‡]
ravi.rajwar@intel.com

[†]Parallel Computing Laboratory
Intel Labs
Santa Clara, CA 95054

[‡]Intel Architecture Development Group
Intel Architecture Group
Hillsboro, OR 97124

ABSTRACT

Intel has recently introduced Intel® Transactional Synchronization Extensions (Intel® TSX) in the Intel 4th Generation Core™ Processors. With Intel TSX, a processor can dynamically determine whether threads need to serialize through lock-protected critical sections. In this paper, we evaluate the first hardware implementation of Intel TSX using a set of high-performance computing (HPC) workloads, and demonstrate that applying Intel TSX to these workloads can provide significant performance improvements. On a set of real-world HPC workloads, applying Intel TSX provides an average speedup of 1.41x. When applied to a parallel user-level TCP/IP stack, Intel TSX provides 1.31x average bandwidth improvement on network intensive applications. We also demonstrate the ease with which we were able to apply Intel TSX to the various workloads.

Categories and Subject Descriptors

B.8.2 [**Hardware**]: Performance and Reliability—*performance analysis and design aids*; C.1.4 [**Computer Systems Organization**]: Processor Architectures—*parallel architectures*; D.1.3 [**Software**]: Programming Techniques—*concurrent programming*

General Terms

Performance, Measurement

Keywords

Transactional Memory, High-Performance Computing

SC13, November 17–21, 2013, Denver, CO, USA.

ACM 978-1-4503-2378-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2503210.2503232

1. INTRODUCTION

Due to limits in technology scaling, software developers have come to rely on thread-level parallelism to obtain sustainable performance improvement. However, except for the case where the computation is massively parallel (e.g., data-parallel applications), performance of threaded applications is often limited by how inter-thread synchronization is performed. For example, using coarse-grained locks can limit scalability, since the execution of lock-guarded critical sections is inherently serialized. Using fine-grained locks, in contrast, may provide good scalability, but increases locking overheads, and can often lead to subtle bugs.

Various proposals have been made over the years to address the limitations of lock-based synchronization. Lock-free algorithms support concurrent updates to data structures and do not require mutual exclusion through a lock. However, such algorithms are very difficult to write and may not perform as well as their lock-based counterparts. Hardware transactional memory [11] and Oklahoma Update Protocol [26] propose hardware support to simplify the implementation of lock-free data structures. They rely on mechanisms other than locks to ensure forward progress. Speculative Lock Elision [22] proposes hardware support to expose concurrency in lock-based synchronization—the hardware would optimistically execute critical sections without serialization and serialize execution only when necessary. In spite of these proposals, writing correct, high-performance multi-threaded programs remains quite challenging.

Intel has introduced Intel® Transactional Synchronization Extensions (Intel® TSX) in the Intel 4th Generation Core™ Processors [12] to improve the performance of critical sections. With Intel TSX, the hardware can dynamically determine whether threads need to serialize through lock-protected critical sections. Threads perform serialization only if required for correct execution. Hardware can thus expose concurrency that would have been hidden due to unnecessary synchronization.

In this paper we apply Intel TSX to a set of workloads in the high-performance computing (HPC) domain and present the first evaluation of the performance benefits when running on a processor with Intel TSX support. The evaluation incorporates a broad spectrum of workloads, ranging from kernels and benchmark suites to a set of real-world

workloads and a parallel user-level TCP/IP stack. Some of the workloads were originally written to stress test a throughput-oriented processor [24], and have been optimized for the HPC domain. Nevertheless, applying Intel TSX to these workloads provides an average speedup of 1.41x. Applying Intel TSX to a user-level TCP/IP stack provides an average bandwidth improvement of 1.31x on a set of network intensive applications. These results are in contrast to prior work on other commercial implementations that show little to no performance benefits [23, 29], or are limited to small kernels and benchmarks [20, 6, 5].

We demonstrate multiple sources of performance gains. The dynamic avoidance of unnecessary serialization allows more concurrency and improves scalability. In other cases, we reduce the cost of uncontended synchronization operations, and achieve performance gains even in single thread executions. Much of the gain is achieved with changes just in the synchronization library: In some cases, localized changes in the application code results in additional gains.

Section 2 presents a brief overview of Intel TSX. We describe the experimental setup in Section 3 and outline how we apply Intel TSX to the various workloads. Section 4 characterizes Intel TSX using a suite of benchmarks. We evaluate Intel TSX for these benchmarks without any source code changes. In Section 5, we evaluate Intel TSX performance on a set of real-world workloads. We also demonstrate two key techniques to further improve performance: *lockset elision* and *transactional coarsening*. These techniques are useful if one can modify the source code to optimize for performance. In Section 6, we apply Intel TSX to a large-scale software system using a user-level TCP/IP stack and identify some of the challenges, such as condition variables. We discuss related work in Section 7 and conclude in Section 8.

2. INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS

Intel TSX provides developers an instruction set interface to specify critical sections for transactional execution[1]. The hardware executes these developer-specified critical sections transactionally, and without explicit synchronization and serialization. If the transactional execution completes successfully (*transactional commit*), then memory operations performed during the transactional execution appear to have occurred instantaneously, when viewed from other processors. However, if the processor cannot complete its transactional execution successfully (*transactional abort*), then the processor discards all transactional updates, restores architectural state, and resumes execution. The execution may then need to serialize through locking if necessary, to ensure forward progress. The mechanisms to track transactional states, detect data conflicts, and commit atomically or rollback transactional states are all implemented in hardware.

Intel TSX provides two software interfaces to specify critical sections. The Hardware Lock Elision (HLE) interface is a legacy compatible instruction set extension (XACQUIRE and XRELEASE prefixes) for programmers who would like to run HLE-enabled software on legacy hardware, but would also like to take advantage of the new transactional execution capabilities on hardware with Intel TSX support. Restricted Transactional Memory (RTM) is a new instruction set extension (comprising the XBEGIN and XEND instructions) for programmers who prefer a more flexible interface than HLE. When an RTM region aborts, architectural state is recovered, and execution restarts non-transactionally at the fallback address provided with the XBEGIN instruction.

Intel TSX does not guarantee that a transactional execution will eventually commit. Numerous architectural and microarchitectural conditions can cause aborts. Examples include data conflicts, exceeding buffering capacity for transactional states, and executing instructions that may always abort (e.g., system calls). Software using RTM instructions should not rely on the Intel TSX execution alone for forward progress. The fallback path not using Intel TSX support must ensure forward progress, and it must be able to run successfully without Intel TSX. Additionally, the transactional path and the fallback path must co-exist without incorrect interactions.

Software using the RTM instructions for lock elision must test the lock during the transactional execution to ensure correct interaction with another thread that may or already has explicitly acquired the lock non-transactionally, and should abort if not free. The software fallback handler should define a policy to retry transactional execution if the lock is not free, and to explicitly acquire the lock if necessary.

When using the Intel TSX instructions to implement lock elision, whether through the HLE or RTM interface, the changes required to enable the use of these instructions are limited to synchronization libraries, and do not require application software changes.

The first implementation of Intel TSX on the 4th Generation Core™ microarchitecture uses the first level (L1) data cache to track transactional states. All tracking and data conflict detection are done at the granularity of a cache line, using physical addresses and the cache coherence protocol. Eviction of a transactionally written line from the data cache will cause a transactional abort. However, evictions of lines that are only transactionally read do not cause an abort; they are moved into a secondary structure for tracking, and may result in an abort at some later time.

3. EXPERIMENTAL SETUP

We use an Intel 4th Generation Core™ processor with Intel TSX support. The processor has 4 cores with 2 Hyper-Threads per core, for a total of 8 threads. Each core has a 32 KB L1 data cache. We use Intel C/C++ compiler for most of our studies, but for those applications utilizing OpenMP, we also use GCC with `libgomp` to precisely control the number of threads. We use inline assembly to emit bytes for Intel TSX instructions, but intrinsics are also available through compiler header files (e.g., `<immintrin.h>`).

Unless otherwise noted, we use thread affinity to bind threads to cores so that as many cores are used as possible—e.g., a 4 thread run will use a single thread on each of the 4 cores, while an 8 thread run will also use 4 cores, but with 2 threads per core. A minimum of 10 executions are averaged to derive statistically meaningful results.

The workloads we use in this paper include transactional memory benchmark suites (CLOMP-TM [23], STAMP [19], and RMS-TM [16]), real-world applications from the HPC domain, and a large-scale software system with a TCP/IP stack running network intensive applications.

[1]Full specifications for Intel TSX can be found in [12]. Enabling and optimization guidelines can also be found in [13]. Additional resources for Intel TSX can be found at http://www.intel.com/software/tsx.

These workloads use *synchronization libraries* to coordinate accesses to shared data. An application may either directly call these libraries, or invoke them indirectly through macros or pragmas. These underlying libraries provide multiple mechanisms for synchronizing accesses to shared data. If the shared data being updated is a single memory location (an atomic operation), then the library can achieve this through the use of an atomic instruction (such as LOCK-prefixed instructions in the Intel 64 architecture). For more complex usages, lock-protected critical sections are used.

We apply Intel TSX to the underlying synchronization library, and do not *require* application source changes or annotations. Specifically, in this paper we use the RTM-based interface to elide the relevant critical section locks specified by the synchronization library, and execute the critical section transactionally. If the transactional execution is unsuccessful, then the lock may be explicitly acquired to ensure forward progress. The decision to acquire the lock explicitly is based on the number of times the transactional execution has been tried but failed; for our hardware and workloads, 5 gave the best overall performance. To ensure correct interaction of the transactional execution with other threads that may or already has explicitly acquired the lock, the state of the lock is tested during the transactional execution.

4. EVALUATION ON TRANSACTIONAL MEMORY BENCHMARKS

We start by using CLOMP-TM [23] microbenchmark to characterize Intel TSX performance, and then use the STAMP benchmark suite [19] to see how such performance translates to workload performance. We also apply Intel TSX to RMS-TM [16], and observe how it compares to fine-grained locking, and how it interacts with system calls during a transactional execution.

These transactional memory (TM) benchmark suites use macros and pragmas to invoke the underlying TM library. In addition to a TM implementation, the library also provides a lock-based critical section implementation, equivalent to a conventional lock-based execution model using a global lock. We apply Intel TSX to elide the global lock in the critical section implementation.

4.1 CLOMP-TM Results

In this section we characterize Intel TSX performance using the CLOMP-TM benchmark 1.6 [23]. CLOMP-TM is a synthetic memory access generator that emulates the synchronization characteristics of HPC applications; an unstructured mesh is divided into *partitions*, where each partition is subdivided into *zones*. Threads concurrently modify these zones to update the mesh.

Specifically, each zone is pre-wired to deposit a value to a set of other zones, *scatter zones*, which involves (1) reading the coordinate of a scatter zone, (2) doing some computation, and (3) depositing the new value back to the scatter zone. Since threads may be updating the same zone, value deposits need to be synchronized. Conflict probability can be adjusted by controlling how the zones are wired; and by changing the number of scatters per zone, the amount of work done in a critical section can be adjusted.

To compare Intel TSX performance against existing synchronization methods, we use the benchmark to reproduce the experiment conducted in [23]. Here, threads do not contend for memory locations, and to avoid artifacts from L1 data cache sharing among threads, we disable Hyper-Threading (i.e., we use 4 threads).

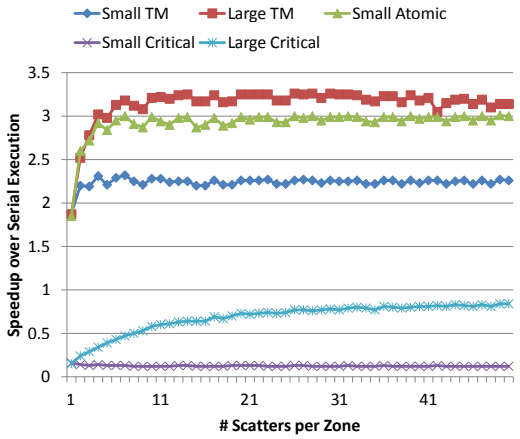


Figure 1: **CLOMP-TM benchmark results for 4 threads. Intel TSX version (*Large TM*) outperforms atomic instruction-based version (*Small Atomic*) when at least 3 or 4 scatter zone updates are batched.**

Figure 1 shows the results. In the figure, **Small Atomic** denotes the case where a LOCK-prefixed instruction is used to enforce atomicity on a single scatter zone value update; this is equivalent to using `#pragma omp atomic`. Likewise, **Small Critical** denotes the use of a lock, equivalent to `#pragma omp critical`, for each scatter zone update. **Large Critical** denotes the case where for each zone, we batch the scatter zone updates (and the accompanying index and value computation code) under a critical section guarded by a single lock. **Small TM** and **Large TM** map the lock-guarded critical sections in **Small Critical** and **Large Critical** into calls into the Intel TSX-enabled synchronization library.

The X-axis denotes the number of scatters for each zone, and at each scatter count, the speedup is against the execution time of the corresponding serial version.

When we synchronize on each scatter zone update, while the LOCK prefix-based version (**Small Atomic**) is the fastest, Intel TSX version (**Small TM**) is not too much worse. The version that uses lock (**Small Critical**), however, performs a lot worse. In contrast, batching a set of scatter zone updates into a single critical section allows better amortization of the synchronization costs. Especially, Intel TSX with batching (**Large TM**) outperforms even **Small Atomic** once we batch at least 3∼4 updates. Batching with lock (**Large Critical**), however, suffers from lock contentions, and remains slow.

Compared to the results presented in [23], which requires 5 to 10 updates to be batched before its transactional execution outperforms atomic updates, Intel TSX exhibits lower overhead. However, the scale at which the transactional execution is implemented on [23] is different (16 cores per chip, 4 threads per core). Therefore, a direct comparison cannot be made.

4.2 STAMP Results

STAMP [19] is a benchmark suite extensively used by the transactional memory community. Compared to CLOMP-TM, its workloads are much closer to a realistic application. We use the benchmark suite (0.9.10) to see how Intel TSX performance translates into application performance.

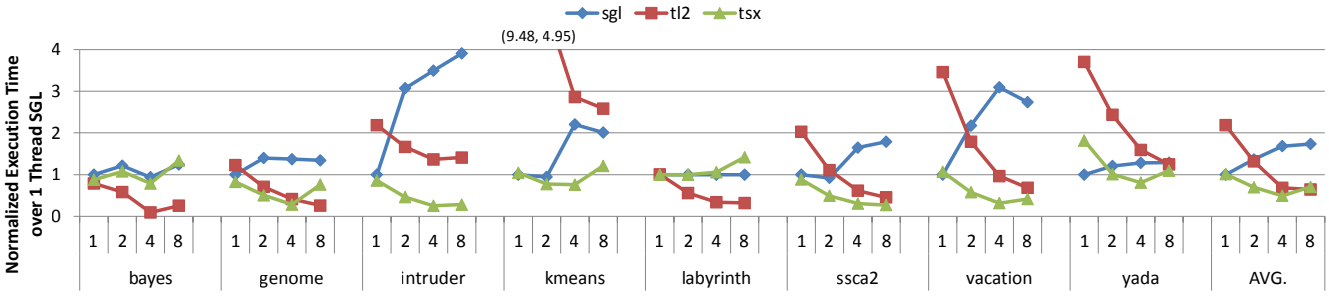


Figure 2: **STAMP benchmark results. Intel TSX provides low single thread overhead, while outperforming a software transactional memory (TL2 [7]) in many cases.**

Workload	1 thread		2 threads		4 threads		8 threads	
	tl2	tsx	tl2	tsx	tl2	tsx	tl2	tsx
bayes	0	**64**	1	**91**	2	**89**	6	**94**
genome	0	6	0	11	1	19	1	**88**
intruder	0	6	32	11	50	31	57	**74**
kmeans	0	0	15	26	35	71	55	**96**
labyrinth	0	**87**	4	**95**	8	**100**	16	**97**
ssca2	0	0	0	1	0	1	0	1
vacation	0	**38**	2	51	6	52	9	**99**
yada	0	**46**	46	68	58	84	65	**92**

Table 1: **Transactional abort rates (%) for the STAMP benchmark suite. Figures of particular interest are highlighted.**

Specifically, some STAMP workloads use critical sections with medium/large memory footprint. Memory accesses within a critical section that are required for synchronization correctness have been manually annotated for use by software transactional memory (STM) implementations. STMs rely on instrumenting memory accesses within a transactional region to track transactional reads and writes, and such annotation allows STMs to only track necessary accesses. When using a lock-based execution, these annotated accesses get mapped to regular loads and stores, and are synchronized using the underlying locking mechanism.

Figure 2 shows the execution time of different synchronization schemes implemented by the underlying TM library. We use the *native* input with high contention configuration.

The execution time in the figure is normalized to the single thread execution time of the **sgl** version. **sgl** represents the case where the TM library implements transactional regions as critical sections protected through a single global lock. This scheme forces all transactional regions to serialize, and thus prevents scaling if critical sections comprise a significant fraction of an application's execution. As expected, with increasing thread count, workloads do not scale.

tl2 represents the performance where the TM library implements transactional regions using the STM included in the benchmark distribution, called TL2 [7]. Overall, by leveraging the annotations to only track crucial memory accesses, STM provides good scalability. However, except for **labyrinth**, it suffers significant single thread overhead. This is because STM has to instrument the annotated memory accesses within a transactional region. On a single-threaded execution, it still pays this overhead, but cannot exploit concurrency to make up for the performance loss.

Intel TSX, however, does not require any instrumentation. In the figure, **tsx** represents the performance where we apply Intel TSX to transactionally elide the single global lock in **sgl**. As can be seen, the Intel TSX-enhanced library shows radically improved single thread performance. Specifically, the performance is comparable to single global lock. With more threads, however, Intel TSX scales significantly better than single global lock, and in many cases, outperforms STM. With both good single-thread performance and good scalability, a programmer may elect to apply Intel TSX over coarse-grained locks, instead of the conversion effort to fine-grained locks or suffering the high overheads of STM.

Although we provide results on all the workloads for completeness, results on **bayes** and **kmeans** should be discounted, because their execution is strongly dependent on the *order* of various parallel computations—thus, a slower synchronization scheme may result in faster benchmark execution, and vice versa. Specifically, **bayes** utilizes a hill-climbing strategy that combines local and global search [19]. We notice that executions with STM consistently get stuck in local minima, terminating the search earlier but returning inferior results. Similarly, **kmeans** iterates its algorithm until the cluster search converges; we notice that an implementation using Intel TSX always converges faster than STM. We suspect both cases are related to how this specific STM implementation handles floating point variables, and are currently investigating the issue.

Table 1 shows transactional abort percentage that gives more insight into TL2 and Intel TSX behavior. We collect Intel TSX statistics through Linux `perf`. First to note is the non-trivial abort rate of Intel TSX with only one thread. These aborts are mostly due to the effective capacity limit of the set-associative L1 data cache for medium/large critical sections. Hyper-Threading, on the other hand, increases the pressure on the L1, compounding the capacity issue. Thus, in the table, Intel TSX sees significantly higher transactional abort rates with 8 threads than with 4 threads.

Overall, while STAMP tries to cover diverse transactional characteristics, we see that some workloads stopped critical section refinement at medium/large footprint; this would not have been a problem for STMs with virtually unlimited buffer size. STMs also manage to avoid capacity issues through their heavy use of selective annotation (e.g., for **labyrinth**, a 14 MB copy of a global structure to thread-local memory is not annotated). Such manual annotation requires significant effort, especially in a large-scale software system [10], and is not possible with high-level transactional programming constructs [27].

However, due to its low overhead, Intel TSX provides speedup over STM in many cases where its capacity-induced abort rate is reasonable.

4.3 RMS-TM Results

The STAMP benchmark suite is written from the ground-up specifically to evaluate transactional memory implementations. In contrast, RMS-TM [16] adapts a set of existing workloads to use transactional memory. As a result,

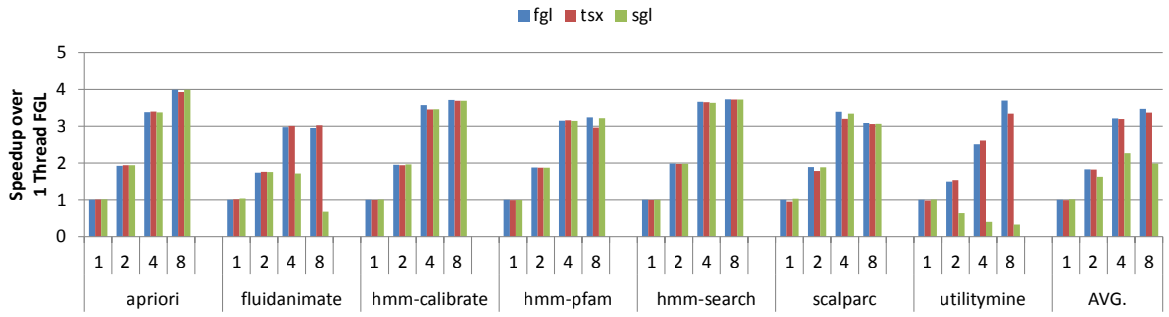


Figure 3: **RMS-TM benchmark results. Intel TSX provides comparable performance to fine-grained locking, even when system calls are made during a transactional execution.**

Workload	Description	Threading	Sync	Txn Technique		
				Lockset	StatC	DynC
graphCluster	Performs min-cut graph clustering. Kernel 4 of SSCA2 [1].	OpenMP	locks	√		√
ua	Unstructured Adaptive (UA) from NAS Parallel Benchmarks suite [9]. Solves a set of heat equations on an adaptive mesh.	OpenMP	atomics		√	√
physicsSolver	Uses PSOR to solve a set of 3-D force constraints on groups of blocks.	PThread	locks	√		√
nufft	Non-uniform FFT. Baseline reported in [15].	OpenMP	locks			√
histogram	Parallel image histogram construction.	PThread	atomics			√
canneal	VLSI router from PARSEC [2]. Performs simulated annealing.	PThread	lock-free			

Table 2: **Real-world workloads used in this study.** *Sync* **denotes the synchronization mechanism used by the original code.** *Txn Technique* **represents the transactional optimization techniques we apply (***Lockset* **= Lockset Elision,** *StatC* **= Static Coarsening, and** *DynC* **= Dynamic Coarsening).**

workloads in RMS-TM exhibit different characteristics from STAMP. Specifically, compared to the medium/large transactions used by STAMP, RMS-TM utilizes fine-grained locks. Therefore, the critical sections exhibit moderate footprint, and as in high-level transactional programming languages [27], no manual annotation is performed. On the other hand, the workloads perform (non-transactional) memory allocation and I/O within critical sections.

We use the RMS-TM benchmark suite to observe how Intel TSX-based synchronization fares in scenarios that are (1) either already optimized (through fine-grained locks) or are (2) not always friendly for transactional execution (i.e., memory allocation and I/O within critical sections). Specifically, we disable the `TM-MEM` and `TM-FILE` flags to perform native memory management and file operations within transactional regions, and use the larger input set provided by the benchmark. Figure 3 shows the results.

We compare the speedup of Intel TSX (**tsx**) to fine-grained locking (**fgl**), relative to fine-grained locking with a single thread. With fine-grained locking, RMS-TM workloads scale reasonably well. Using Intel TSX provides comparable performance, demonstrating that memory allocation and I/O within a transactional region do not require special handling, nor necessarily impact performance to a significant degree. As long as such a condition is detected early and the lock is acquired, system calls may not be a performance issue. We also observe that Hyper-Threading has less performance impact on Intel TSX, primarily because the data footprints are moderate as compared to some STAMP workloads.

Figure 3 also shows the performance when we use single global lock (**sgl**) to synchronize all critical sections. Here, macros that mark critical sections are mapped to acquire and release a single global lock, instead. Therefore, the code section that is being synchronized is the same as Intel TSX.

Guarding the critical sections with fine-grained locks or a single global lock does not make significant performance differences, except in **fluidanimate** with lots of small critical sections, and **utilitymine** with more than 30% of execution spent in critical sections [16]. Here, single global lock fails to scale, while Intel TSX effectively exploits the parallelism, providing comparable performance to fine-grained locking.

5. EVALUATION ON REAL-WORLD WORKLOADS

In this section, we apply and evaluate Intel TSX on a set of real-world workloads. These applications use different types of synchronization mechanisms: lock-based critical sections, atomic operations, and lock-free data structures. Applying Intel TSX to the lock-based critical sections is straightforward. However, we modified the source code so that we could also apply Intel TSX to code regions that use atomic operations and lock-free data structures.

For each workload, we start with a straightforward translation, and then consider optimizations to improve the performance of transactional synchronization.

5.1 Workloads

Table 2 shows the workloads we use for this study. These workloads cover various threading and synchronization schemes, and some represent computations typically found in the HPC domain. In fact, **physicsSolver** and **histogram** were used to stress test a throughput-oriented processor [24].

Specifically, **graphCluster** is Kernel 4 of the SSCA2 benchmark [1]. The **ssca2** workload in STAMP, in contrast, reimplements Kernel 1 for transactional memory from the ground-up. **graphCluster** partitions a graph into clusters while minimizing edge cut costs. Vertices are observed in parallel, and based on the neighbors, they may be added/removed from the cluster. The original application uses per-vertex locks to synchronize updates on the vertex status.

ua is the Unstructured Adaptive workload from NAS Parallel Benchmarks suite [9]. To handle the adaptively refined mesh, **ua** utilizes the Mortar Element Method [9], where thread-local computations performed on *collocation points* are dynamically gathered (i.e., reduced) to *mortars* on a

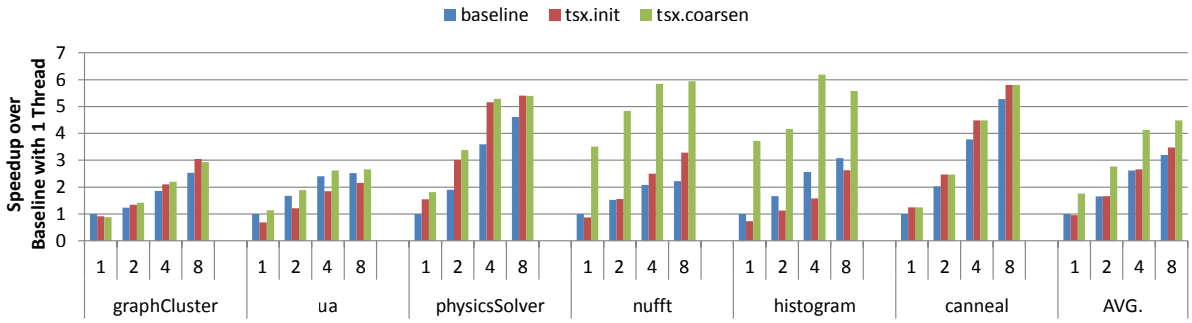


Figure 4: **Intel TSX performance on real-world workloads.**

global grid. Since the grid dynamically changes, gathers on each mortar require synchronization—the original application uses atomic operations. Reduced values are later scattered back to collocation points.

physicsSolver iteratively resolves constraints between pairs of objects, computing the force exerted on each other to prevent inter-penetration. A key critical section updates the total force exerted on both objects in a given pair. Since each object may be involved in multiple pair-wise interactions, the original application acquires a pair of locks to resolve each constraint, one lock for each object.

nufft performs 3-D non-uniform FFT. We use the baseline version reported in [15]. Specifically, we focus on the *adjoint NUFFT* operator, which reduces a set of non-uniformly spaced spectral indices onto a uniform spectral grid. Since the reduction combines an unpredictable set of non-uniform indices for each grid point, it requires synchronization. The original application uses an array of locks for this.

histogram is an image histogram construction workload. Multiple threads directly update the shared histogram; thus, the updates require synchronization. The original application uses an atomic operation for each bin update. While simple, histogram comprises the core compute of many HPC workloads, such as the two-point correlation function in astrophysics [3], and radix sort [17].

Lastly, **canneal** is a routing workload from PARSEC [2]. It performs simulated annealing, where each thread tries to randomly swap two elements to improve solution quality. To perform this swap in an atomic fashion, the original application implements sophisticated lock-free synchronization.

5.2 Unoptimized Performance Results

For these workloads, we map the lock-based critical sections to calls into the Intel TSX-enhanced synchronization library. The library converts these into critical sections protected by a single global lock, and applies Intel TSX to elide that lock. For atomic operations, we convert the LOCK-prefixed operation into a regular operation, and protect the update using a global lock-based critical section. This is then mapped into a call to the Intel TSX-enabled synchronization library. For the lock-free algorithms, we replace the entire algorithm to use global lock-based critical sections, discarding the atomic instructions and version checking codes from the original algorithm. Intel TSX-based synchronization library is then applied.

Figure 4 shows the results. In the figure, **baseline** represents the performance of the original, lock- and atomics-based code. **tsx.init** represents the performance of the Intel TSX-enabled version. We discuss **tsx.coarsen** later.

Even with straightforward porting, Intel TSX provides a noticeable performance improvement. For example, **nufft** has significant concurrency within a critical section hidden under lock contention, which we exploit with transactional execution. For **canneal**, we confirm the observation in [5]: Replacing the complicated lock-free algorithm with a transactional region not only makes the code much simpler, but since some atomic read-time checks can now be removed, provides significant performance improvement as well.

```
myLock = omp_test_lock(&vLock[w]);
if (myLock) {
    // Non-blocking path
    ... update graph ...
    omp_unset_lock(&vLock[w]);
} else {
    // Blocking path
    omp_set_lock(&vLock[w]);
    ... update graph ...
    omp_unset_lock(&vLock[w]);
}
```

Listing 1: *graphCluster* **code example.**

5.2.1 Lockset Elision

For **graphCluster** and **physicsSolver**, on the other hand, we find *lockset elision* to be the key reason for performance improvement. On these workloads, for some critical sections, a set of locks need to be acquired before we can enter the section. Lock acquisitions typically involve costly atomic operations, and the overhead of acquiring a set of locks can be even higher. By replacing *a set of lock acquisitions* with a single transactional begin, we can reduce the overheads—we call this *lockset elision*. We similarly replace the set of lock releases with a single transactional commit.

For example, for **physicsSolver**, we substitute a single transactional begin for a set of two lock acquisitions, one for each object in the pair that is being processed.

Lockset elision can be more subtle. Listing 1 shows an example from **graphCluster**. To synchronize when updating a vertex, the original code utilizes two critical sections, a non-blocking path and a blocking path. Using `omp_test_lock()`[2], a thread first tries to acquire the lock in a non-blocking fashion. If it succeeds, the thread enters the non-blocking path. If the non-blocking lock acquisition fails, the thread enters the blocking path, and calls `omp_set_lock()` to invoke the blocking lock acquisition code.

When there is little contention for the locks, if the OpenMP implementation has lower lock acquisition overhead for `omp_test_lock()` than `omp_set_lock()`, this code will be

[2]The OpenMP specification [21] defines that 'These routines attempt to set an OpenMP lock but do not suspend execution of the task executing the routine.' That is, the routine is a *try-lock* operation, despite the name.

```
// Compute collocation point indices
il1, il2, il3, il4 = ...;

// Compute mortar indices
ig1, ig2, ig3, ig4 = ...;

#pragma omp atomic
    tmor[ig1] += tx[il1]*third;
#pragma omp atomic
    tmor[ig2] += tx[il2]*third;
#pragma omp atomic
    tmor[ig3] += tx[il3]*third;
#pragma omp atomic
    tmor[ig4] += tx[il4]*third;
```

Listing 2: *ua* **code example.**

```
for (int y = start_row; y < end_row; y++) {
    for (int x = 0; x < width; x++) {
        if (x % TXN_GRAN == 0)
            TM_BEGIN();
        // Update histogram bin
        UPDATE_BIN(x);
        if (x % TXN_GRAN == TXN_GRAN - 1)
            TM_END();
    }
    src_ptr += width;
}
```

Listing 3: *histogram* **code example.**

more efficient than using `omp_set_lock()` everywhere. However, under high contention, the code performs an additional lock check just to find that the non-blocking path cannot be taken. These two lock checks can be replaced with a single transactional begin, reducing overheads.

5.2.2 Transactional Coarsening

In contrast, our initial **ua** and **histogram** ports to transactional execution are slower than the original code; these workloads use LOCK prefix-based atomics to perform individually synchronized updates to shared data structures. As seen in Section 4.1, using a critical section to perform an update of a single memory location, whether using Intel TSX or not, has higher overhead than atomics (in Figure 1, compare **Small TM** and **Small Atomic**). On the other hand, batching the updates can amortize overheads; for CLOMP-TM, batching just a few updates allows transactional execution to outperform atomics (i.e., **Large TM** performs better than **Small Atomic** at that point).

We consider *transactional coarsening*, or applying batching to our applications. Specifically, we apply two techniques: *static* and *dynamic coarsening*. *Static coarsening* merges different critical sections (or atomic updates) into one transactional region, at the source code level. Listing 2 shows a code example from **ua**. The code snippet shows the gather phase, where atomics are used to synchronize the reduction on the dynamically changing grid. Our original port places each atomic update into its own transactional region, and removes the now-unnecessary `atomic` pragmas. To amortize the synchronization overhead, we now place all of these updates in a single transactional region.

In contrast, *dynamic coarsening* combines multiple dynamic *instances* of the same transactional region. Listing 3 shows a **histogram** code section amenable to dynamic coarsening. Basically, the code skips some XBEGIN and XEND instances based on the loop index, to combine TXN_GRAN updates into a single transactional region. We explore the best value for TXN_GRAN later.

We expect that dynamic coarsening could be easily applied with compiler loop unrolling, or through lightweight runtimes. Static coarsening requires more complex static analysis, but is still amenable to automation.

5.3 Optimized Performance Results

We selectively apply transactional coarsening to our workloads. Table 2 shows the specific techniques we apply. In Figure 4, **tsx.coarsen** demonstrates the performance after coarsening. We see that for those two workloads where Intel TSX performed worse than the baseline (i.e., **ua** and **histogram**), Intel TSX now provides a significant speedup. Other workloads benefit from transactional coarsening as well: On average, with 8 threads, Intel TSX provides 1.41x speedup over the baseline. Transactional coarsening may not change scalability noticeably, but by reducing the overhead even in single thread, absolute performance is higher for all thread counts.

5.4 Discussion

5.4.1 Alternative Optimizations

Implementing the equivalent of lockset elision or transactional coarsening using traditional synchronization constructs is difficult. The only static method is to increase the granularity of a critical section. While this could improve performance at low thread counts, coarse-grained locks may not scale. Techniques like multiple granularity locking could be used to dynamically adapt locking granularity, but this is challenging for the programmers to use. Further, it can sometimes be nontrivial to determine prior to a critical section which set of shared locations will be accessed.

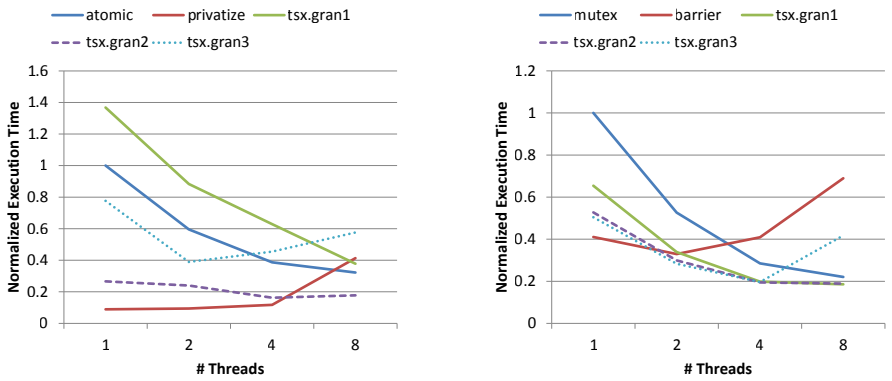
In contrast, with Intel TSX-based synchronization, one can readily convert multiple lock acquisitions or critical sections into a single transactional region, to better amortize the overhead while not significantly sacrificing scalability. Similarly, workloads that perform synchronized updates on multiple memory locations using atomic operations will also benefit by merging them into a single transactional region. In these situations, Intel TSX can improve performance over even fine-grained locking.

5.4.2 Conflict-Free Approaches

Lock- or atomics-based synchronization assumes that concurrent threads perform possibly conflicting updates directly to a shared data structure, and serializes the conflicting accesses. Some HPC applications instead use a *conflict-free* approach, where concurrent updates are *pre-arranged* to be independent. Arranging those updates requires effort and causes overheads, but can skip synchronization for each update. Privatization and barrier-based synchronization are two popular mechanisms used.

Under privatization, each thread (or processor, or node) maintains a *local copy* of the shared data structure on which it performs modifications. Once the updates are done, the application *reduces* the copies, merging the updates. Creating the copies and performing the reduction, however, are overheads that increase with the size of the shared data structure. Privatization therefore scales only if the number of updates is large, relative to the size of the data structure.

With barriers, we eliminate the possibility of conflicts by separating updates that access the same part of the shared data structure into different groups. The application then executes one group of updates at a time in parallel, with a



(a) *histogram* **performance results.** (b) *physicsSolver* **performance results.**

Figure 5: **Comparison of different synchronization schemes. Execution time is normalized to baseline (*atomic* and *mutex*, respectively) with a single thread. *tsx.gran** represent different transactional granularities.**

barrier between each group to enforce any dependencies between groups. Here, the barriers and formation of the groups are overheads. Also, if the groups are not large enough, this scheme may introduce load imbalance.

Figure 5 demonstrates two cases where the overheads of these conflict-free schemes outweigh the benefits (i.e., smaller number of synchronizations). In Figure 5a, **atomic** denotes the baseline version of **histogram** that uses atomics for each update. In contrast, **privatize** denotes an alternate version that privatizes the histogram. In Figure 5b, **mutex** denotes the performance of the baseline **physicsSolver**, and **barrier** denotes an implementation that uses barrier-based synchronization. For this workload, we omit the time for forming the groups of independent tasks, since those groups are used repeatedly, amortizing the overhead. In both graphs, **tsx.gran*** denote the performance of Intel TSX across varying transactional granularities. Execution time is normalized to that of the baseline with one thread.

In these experiments, while both privatization and barriers provide good performance with low thread counts, they do not scale. For **histogram**, the number of histogram bins is large relative to the number of items being binned. Therefore, the reduction overhead eventually dominates the execution time. For **physicsSolver**, the input scene has a few objects with many updates, causing large load imbalance.

In these cases, an approach with more synchronization is faster. At 8 threads, even locks and atomics outperform the conflict-free approaches. This may not be true in all applications, nor for all inputs and input parameters—the best performing scheme will depend on several factors, including the overhead for a single synchronization operation. However, as Figure 5 (and our earlier results) demonstrates, Intel TSX can reduce such overheads; and as the overhead decreases, conflict-free approaches look relatively worse. Combined with the promise of significantly easier parallelization, Intel TSX makes an approach with more synchronization more attractive.

5.4.3 *Choosing the Right Granularity*

Figure 5 also shows workload performance sensitivity to transactional coarsening (the **tsx.gran*** lines—a larger number indicates coarser transactional regions). In general, coarsening improves performance by better amortizing the transactional overheads. However, as the transactional region/-footprint gets larger, it becomes more prone to conflicts. Thus, we expect (and observe) a performance inflection point as we increase transactional granularity; e.g., in Figure 5b, at 8 threads, largest granularity (**tsx.gran3**) does not provide the best performance. A hardware or runtime-assisted approach to dynamically adjust transactional coarsening could be necessary.

6. EVALUATION IN LARGE-SCALE SOFTWARE SYSTEMS

In the previous section, we focused on improving individual workload performance through Intel TSX. In this section we explore how Intel TSX can be applied in a large software framework, to benefit a larger set of workloads that utilize the framework. In particular, we apply Intel TSX to a parallel user-level TCP/IP stack, and measure the performance of some network intensive applications. Findings should be beneficial to OS, hypervisor, and library development.

6.1 Case Study: User-Level TCP/IP Stack

Version 3.0 of the PARSEC benchmark suite[3] includes a multithreaded user-level TCP/IP stack, which is a user-level port of a BSD network stack. In the stack, all the synchronization constructs—locks, condition variables, etc.—and routines are implemented in a single locking module, which acts as a wrapper to the underlying PThread library.

We replace the PThread library with an Intel TSX-enabled synchronization library that uses RTM *instructions* to elide the locks; in this scenario, we use the existing critical section locks in the stack. Instruction-based specification of a transactional region does not require lock acquisition and release to be in the same code scope. As reported in [28, 25], however, using scoped transactional programming constructs to achieve the same can be non-trivial. By enhancing the locking module with Intel TSX, all the workloads utilizing this TCP/IP stack can take advantage of Intel TSX without any changes to the workload code.

One significant challenge we faced, however, is the interaction with condition variables. For reference, Listings 4 and 5 show the PThread condition variable wait and signal routines, respectively. Since we substitute `pthread_mutex_lock()` with XBEGIN, in Listing 4, when a thread finds it needs to call `pthread_cond_wait()`, it does not hold a PThread lock to call the function with.

With Intel TSX, a relatively straightforward workaround would be to just unconditionally abort the transactional exe-

[3]http://parsec.cs.princeton.edu/parsec3-doc.htm

```
pthread_mutex_lock(&lock);

while (monitor state not true) {
    // Wait till condition met
    pthread_cond_wait(&lock, &cond);
}

pthread_mutex_unlock(&lock);
```

Listing 4: **PThread condition variable wait routine.**

```
pthread_mutex_lock(&lock);

... update monitor state to true ...

// Signal waiting thread
pthread_cond_signal(&cond);

pthread_mutex_unlock(&lock);
```

Listing 5: **PThread condition variable signal routine.**

cution upon encountering `pthread_cond_wait()`, and to acquire the lock. Once a thread acquires the lock in the fallback handler, it could use the lock to manipulate the condition variable. However, transactional aborts could limit the performance benefits from Intel TSX-based synchronization.

The signaling thread, on the other hand, experiences similar issues in Listing 5, since calling `pthread_cond_signal()` may lead to a system call, which would abort the transactional execution.

Since PThread condition variables are tightly coupled with the locking mechanism, applying Intel TSX to locks requires handling the condition variables as well. Such issues with condition variables have also been reported in other case studies of large-scale software systems [30, 28, 25].

Therefore, we implement a transactional execution-aware condition variable [8]. Instead of the PThread condition variables, this implementation uses Linux `futex`, which does not require holding a lock. Specifically, to reduce transactional aborts, a thread tries to commit partial results when it finds the need to wait on a condition variable. Once it commits, the thread calls `futex` to atomically put itself on the waiters list. The signaling thread, in contrast, registers a callback if it finds the need to signal a condition variable. Upon a transactional commit, the thread will execute the callback to update the `futex`. Once the waiting thread resumes, it starts the transactional execution again. We next compare the performance of the abovementioned implementation options for condition variables.

6.2 Performance Results and Analysis

We evaluate different implementations of the TCP/IP stack on three PARSEC workloads that leverage the stack. These workloads are organized in a client-server fashion, where the client sends the input data over the network, and the server compresses or analyzes the data.

Figure 6 shows the performance of the workloads, normalized to **mutex**, the original TCP/IP stack implementation using PThread locks and condition variables. In the graph, we report the server-side read bandwidth, since it lies on the critical path of the execution. For accurate measurements, workloads with pipeline parallelism (i.e., **netferret** and **netdedup**) have been set to execute the input stages in full before executing the rest of the pipeline.

Our first Intel TSX-based implementation, **tsx.abort**, is the version where we apply Intel TSX to transactionally execute critical sections, but abort the transactional execution

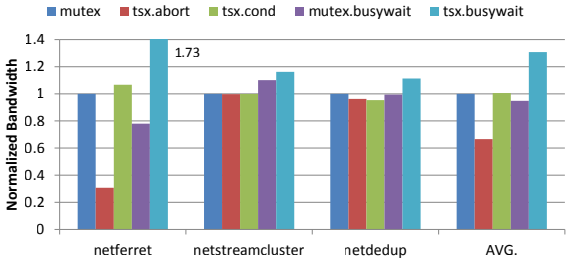


Figure 6: **Intel TSX performance on user-level TCP/IP stack. Reports server-side read bandwidth.**

```
pthread_mutex_lock(&lock);

while (monitor state not true) {
    // Busy-wait till condition met
    pthread_mutex_unlock(&lock);
    pthread_mutex_lock(&lock);
}

pthread_mutex_unlock(&lock);
```

Listing 6: **Busy-wait substitution for conditional wait.**

whenever we have to update a condition variable. The performance drops drastically on **netferret**, since the workload sends/receives many small packets over the network.

Our second implementation, **tsx.cond**, uses the transactional execution-aware condition variable. This implementation has much better performance on **netferret** than **tsx.abort**, and even provides some benefit over **mutex**. However, the other workloads observe little benefit, so the average performance is very similar to **mutex**.

With performance analysis tools, we identify there exists certain delay to putting a thread to sleep and waking it up, and while Intel TSX speeds up the rest of the code, this delay dominates the critical path of the network stack. As a last resort, we replace the wait routine in Listing 4 with busy waiting, as shown in Listing 6. While crude, this waiting conforms to the blocking monitor semantics [18].

In Figure 6, **mutex.busywait** and **tsx.busywait** show the performance of such busy waiting with PThread locks and Intel TSX, respectively. As can be seen, the Intel TSX-enabled stack with busy waiting provides significant performance improvement on all the workloads, albeit with some wasted CPU cycles and energy. This demonstrates that increased speculation via *redundant execution*, i.e., spinning, can translate into application level performance.

7. RELATED WORK

Other commercial designs for transactional execution exists. Sun Microsystems® announced transactional execution capability in its Rock [6] processor. However, it was not made commercially available. IBM® introduced transactional support to its Blue Gene®/Q line of supercomputers [29], later extending it to System z® mainframes [14]. Vega processors from Azul Systems® also had hardware support for transactional execution. This was used to elide locks in the Java™ stack [4]. In general, industrial implementations (1) detect transactional conflicts at the time of access, and (2) buffer modifications on the on-chip storage (e.g., caches). Such designs incur the least modifications to the existing cache coherence and core designs.

These designs, however, differ in where they buffer speculative updates, and whether they provide register check-

pointing. For example, Blue Gene/Q utilizes the L2 cache, Rock [6] and System z utilize store queues (store caches), and the first Intel TSX implementation uses the L1 data cache to buffer speculative writes. The choice of buffering location has microarchitectural implications, such as the capacity for speculative states, latency of commits, and messaging between caches. Intel TSX, Blue Gene/Q, and System z have sufficient buffering capacity to handle moderate-sized transactional regions, except for Rock, which can hold 32 lines. For Rock, System z, and Intel TSX, hardware provides register checkpointing, while Blue Gene/Q relies on software. Such difference may have been a factor in the reported Blue Gene/Q performance results [23].

8. CONCLUSION

We describe Intel Transactional Synchronization Extensions and show that the first implementation has significant performance potential. Using a set of transactional memory benchmark suites running on a processor with Intel TSX support, we first demonstrate that Intel TSX has low overheads. Next, we show that on a set of real-world, high-performance computing workloads, Intel TSX provides 1.41x average speedup over lock- and atomics-based implementations. Finally, we apply Intel TSX-based synchronization to a parallel user-level TCP/IP stack. We observe an average of 1.31x bandwidth improvement on a set of network intensive applications.

Through our work with the benchmarks and applications, we also developed techniques to best utilize Intel TSX. In particular, lockset elision and transactional coarsening provide significant benefits. We also encountered a significant challenge in making condition variables transactional execution-aware. While we present a solution that provides good performance, library-level support for transactional condition variables may be necessary.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. We also thank Pradeep Dubey, Ronak Singhal, Joseph Curley, Justin Gottschlich, and Tatiana Shpeisman for their feedback on the paper.

10. REFERENCES

[1] D. A. Bader and K. Madduri. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *Proceedings of the 12th International Conference on High Performance Computing*, pages 465–476, 2005.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.

[3] J. Chhugani, C. Kim, H. Shukla, J. Park, P. Dubey, J. Shalf, and H. D. Simon. Billion-particle SIMD-friendly two-point correlation on large-scale HPC cluster systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1:1–1:11, 2012.

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

[5] D. Dice, Y. Lev, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 325–334, 2010.

[6] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, 2009.

[7] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 2006 International Conference on Distributed Computing*, pages 194–208.

[8] P. Dudnik and M. M. Swift. Condition variables and transactional memory: Problem or opportunity? In *the 4th ACM SIGPLAN Workshop on Transactional Computing*, 2009.

[9] H. Feng, R. F. V. der Wijngaart, R. Biswas, and C. Mavriplis. Unstructured Adaptive (UA) NAS parallel benchmark, version 1.0. Technical report, NASA Technical Report NAS-04-006, 2004.

[10] V. Gajinov, F. Zyulkyarov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. QuakeTM: Parallelizing a complex sequential application using transactional memory. In *Proceedings of the 23rd International Conference on Supercomputing*, pages 126–135, 2009.

[11] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 1993 Annual International Symposium on Computer Architecture*, pages 289–300.

[12] Intel Corporation. Intel architecture instruction set extensions programming reference. Chapter 8: Intel transactional synchronization extensions. 2012.

[13] Intel Corporation. Intel 64 and IA-32 architectures optimization reference manual. Chapter 12: Intel TSX recommendations. 2013.

[14] C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for IBM System z. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 25–36, 2012.

[15] D. Kalamkar, J. Trzasko, S. Sridharan, M. Smelyanskiy, D. Kim, A. Manduca, Y. Shu, M. Bernstein, B. Kaul, and P. Dubey. High performance non-uniform FFT on modern x86-based multi-core systems. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium*, pages 449–460, 2012.

[16] G. Kestor, V. Karakostas, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. RMS-TM: A comprehensive benchmark suite for transactional memory systems. In *Proceedings of the 2nd Joint WOSP/SIPEW International Conference on Performance Engineering*, pages 335–346, 2011.

[17] C. Kim, J. Park, N. Satish, H. Lee, P. Dubey, and J. Chhugani. CloudRAMSort: Fast and efficient

large-scale distributed RAM sort on shared-nothing cluster. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 841–850.

[18] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Commun. ACM*, 23(2):105–117, 1980.

[19] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the 2008 IEEE International Symposium on Workload Characterization*, pages 35–46.

[20] M. Mitran and V. Vokhshoori. Evaluating the zEC12 transactional execution facility. *IBM Systems Magazine*, 2012.

[21] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 3.1, 2011.

[22] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 294–305, 2001.

[23] M. Schindewolf, B. Bihari, J. Gyllenhaal, M. Schulz, A. Wang, and W. Karl. What scientific applications can benefit from hardware transactional memory? In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 90:1–90:11, 2012.

[24] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 papers*, pages 18:1–18:15.

[25] A. Skyrme and N. Rodriguez. From locks to transactional memory: Lessons learned from porting a real-world application. In *the 8th ACM SIGPLAN Workshop on Transactional Computing*, 2013.

[26] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. Multiple reservations and the Oklahoma update. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(4):58–71, 1993.

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

[28] T. Vyas, Y. Liu, and M. Spear. Transactionalizing legacy code: An experience report using GCC and memcached. In *the 8th ACM SIGPLAN Workshop on Transactional Computing*, 2013.

[29] 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 *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 127–136, 2012.

[30] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee. Kicking the tires of software transactional memory: Why the going gets tough. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 265–274, 2008.

Notice and Disclaimers