Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough

Richard M. Yoo[†]
yoo@ece.gatech.edu

Yang Ni[‡]
yang.ni@intel.com

Adam Welc[‡]
adam.welc@intel.com

Bratin Saha[‡]
bratin.saha@intel.com

Ali-Reza Adl-Tabatabai[‡]
ali-reza.adl-tabatabai@intel.com

Hsien-Hsin S. Lee[†]
leehs@ece.gatech.edu

[†]School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332

[‡]Programming Systems Lab
Intel Corporation
Santa Clara, CA 95054

ABSTRACT

Transactional Memory (TM) promises to simplify concurrent programming, which has been notoriously difficult but crucial in realizing the performance benefit of multi-core processors. Software Transaction Memory (STM), in particular, represents a body of important TM technologies since it provides a mechanism to run transactional programs when hardware TM support is not available, or when hardware TM resources are exhausted. Nonetheless, most previous studies on STMs were constrained to executing trivial, small-scale workloads. The assumption was that the same techniques applied to small-scale workloads could readily be applied to real-life, large-scale workloads. However, by executing several nontrivial workloads such as particle dynamics simulation and game physics engine on a state of the art STM, we noticed that this assumption does not hold. Specifically, we identified four major performance bottlenecks that were unique to the case of executing large-scale workloads on an STM: false conflicts, over-instrumentation, privatization-safety cost, and poor amortization. We believe that these bottlenecks would be common for any STM targeting real-world applications. In this paper, we describe those identified bottlenecks in detail, and we propose novel solutions to alleviate the issues. We also thoroughly validate these approaches with experimental results on real machines.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures*

General Terms

Design, Performance

SPAA'08, June 14–16, 2008, Munich, Germany.

Keywords

C/C++, Compiler, Measurement, Performance, Runtime, Software Transactional Memory

1. INTRODUCTION

Multi-core processors have brought multithreaded programming to the mainstream, forcing even the average programmers to deal with concurrency. Traditionally, programmers have used lock-based mutual exclusion to synchronize shared memory accesses. However, lock-based programming often leads to many software engineering problems, such as deadlock, non-scalable composition, conservative synchronization, etc [4].

Transactional Memory (TM) [4, 7] offers a new concurrency control mechanism that alleviates those pitfalls of lock-based synchronization. Originally proposed as a hardware mechanism [4], TMs have also been implemented in software (STM) [2, 5, 11, 12, 9]. Compared to Hardware Transactional Memory (HTM), STMs are better suited for transactional programming since they provide richer semantics and virtualized transactions, and they would run on legacy hardware. Hence, an efficient STM implementation is critical for enabling transactional programming.

However, most previous research on STMs were limited to two aspects. First, previous studies stressed the underlying STMs with synthetic kernels or workloads with small transactions. As a result, these studies either stressed an STM system in a non-realistic environment, or to a limited extent. Second, many of the STM systems used were API-based [5, 11, 2, 9]. An API-based approach makes it difficult to handle complex workloads with large transactions, since it requires a programmer to manually annotate every transactional memory accesses. In effect, this actually negates the advantages of TM's simple programming model. There existed several studies utilizing compiler-based approaches [15], but they were also limited since they utilized benchmark suites such as SPLASH-2 [16], which only exhibited small critical sections. In summary, previous studies failed to stress the STM system in a realistic environment.

Our current work overcomes these limitations by first preparing realistic multi-core workloads. Our benchmark suite is comprised of workloads such as particle dynamics simulation, game physics engine, speech recognition system, etc. We also ported the STAMP [10] benchmark suite, whose transactional characteristics were complex enough to stress the underlying STM. We then executed these workloads on an industrial-strength, compiler-based STM system; our compiler was a modified version of Intel C/C++

compiler v 10.0, and our runtime was based on the highly tuned framework introduced in [11].

By stressing our STM system in such a realistic environment, we found that even a highly tuned STM system showed significant scalability problems due to a number of performance bottlenecks. More specifically, we identified four major performance bottlenecks: false conflicts, over instrumentation of memory accesses, privatization-safety cost, and poor amortization of transaction startup / teardown costs over short transactions.

- **False conflicts** are transactional conflicts that occur due to the coarse granularity of the conflict detection schemes in STMs [17]. We improved upon [17] by further breaking down the category of such false conflicts. This enabled us to identify that different types of false conflicts actually stress different aspects of an STM system.
- **Over-instrumentation** occurs when the compiler generates excessive read / write barriers due to its lack of application-level knowledge. In general, excessive barriers incur more false conflicts, which again brings about significant performance degradation.
- **Privatization-safety cost** refers to the overhead of guaranteeing the correctness for common programming idioms such as privatization [11, 6, 14]. We found that privatization-safety significantly undermines the scalability of a program in cases where it is not required but conservatively enforced by the system.
- **Poor amortization** happens when the fixed costs of transaction startup / teardown get poorly amortized over short transactions. Under this circumstance, the underlying STM system fails to scale. This behavior was observed even when there were only few transaction aborts.

We believe these performance bottlenecks would be common to any similar STMs targeting real-world TM applications. In this paper, we propose several novel solutions that address the performance bottlenecks described above, and we demonstrate the effectiveness of these solutions by presenting a detailed performance evaluation (Section 3). In addition to addressing these bottlenecks, we also explore design issues that arise when rewriting large-scale workloads with transactions (Section 4). These include additional language constructs and performance metrics. Especially, we found that traditional metrics, such as overall transaction abort rate, do not suffice in fully characterizing the complex behavior of large-scale transactional workloads.

The main contributions of this paper are the following:

- We identify the performance bottlenecks when "the rubber hits the road" — that is, as large, realistic workloads are run even on highly tuned STM systems. This work provides instrumental design guidelines to STM implementers.
- We propose novel solutions to the identified bottlenecks which encompass language constructs and runtime optimizations. We also show how traditional performance metrics fail to characterize the complex behavior of realistic transactional workloads, and propose new metrics.
- We provide detailed performance results of a compiler-based STM system over a wide range of realistic TM workloads.

The rest of the paper is organized as follows. As a background, Section 2 describes the STM and workloads used in our study. Section 3 then describes the identified performance bottlenecks as well as proposed solutions, while also presenting experimental results. Section 4 discusses language design issues and novel performance metrics. Related studies in the STM area can be found in Section 5. Finally, we conclude in Section 6.

2. BACKGROUND

To provide a background for further discussions, in this section we describe the STM system (Section 2.1) and the benchmark suite (Section 2.2) used in our study.

2.1 C/C++ Software Transactional Memory

Our STM system extends the C/C++ language with TM language constructs, which are supported by the TM compiler and the McRT-STM runtime. Following is a brief, simplified description of these language constructs. The complete details can be found in [15].

In our TM extension to C/C++, a transaction is represented by an *atomic block*, which is a statement block annotated with pragma tm_atomic. Once the TM compiler recognizes an atomic block, it 1) generates the corresponding transaction prolog and epilog, and 2) inserts transactional read / write barriers for every memory accesses made by the atomic block.

```
#pragma tm_function
int foo (int);

int bar (int);

#pragma tm_atomic
{
    foo(3);
    bar(10);  // compiler error
}
```

Figure 1: Example for TM Language Constructs

A function called from inside of a transaction is referred to as an *atomic function*, and the function must be annotated by the programmer with pragma tm_function in its declaration. The TM compiler then generates two copies of codes for each atomic function: a regular version and its *transactional twin* with transactional read / write barriers inserted for each memory access. A call to an atomic function is also translated into a call to its transactional twin, if the call is made from inside an atomic block or an atomic function. Calling a non-atomic function from inside a transaction generates a compile-time error. Figure 1 (taken from [15]) shows an example demonstrating the previous two language constructs.

The code generated for transactions relies on the runtime to execute it atomically and in isolation from other transactions. Our runtime is based on McRT-STM [11], which is an in-place update, optimistic-read STM. It represents a design [7] taken by many other high performance STMs.

Internally, the runtime maintains a table of *transaction records*, called *ownership table*. Every memory address is mapped to a unique transaction record in this table. Moreover, for each transaction, the runtime maintains a write set, a read set, and an undo log, which are initialized in the prolog of a transaction.

On a memory store within a transaction, the write barrier tries to exclusively lock the transaction record. If the transaction record is already locked by another transaction, the runtime will try to resolve the conflict before continuing, which may abort the current transaction. If the transaction record is unlocked, the write barrier

Workload	Source	Original Type	Number of Transactions	Time Spent in Transactions (%)
Game Physics Engine (GPE)	Intel	Non-TM	10,661,289	36.6%
Smoothed Particle Hydrodynamics (SPH)	Intel	Non-TM	2,221,485	6.9%
Sphinx	UIUC / ALPBench	Non-TM	618,523	4.8%
Genome	Stanford / STAMP	TM	1,222,188	80.9%
Kmeans	Stanford / STAMP	TM	8,666,710	8.0%
Vacation	Stanford / STAMP	TM	8,866,711	93.6%

Table 1: Benchmark Suite for STM Performance Study

will 1) lock the transaction record, 2) record the old value and the address in its undo log, 3) add the transaction record to its write set, and then 4) update the memory location.

On a memory load within a transaction, the read barrier checks if the transaction record is locked, but does not try to lock it. If the transaction record is locked by another transaction, the conflict is handled. If it is unlocked, the read barrier will 1) return the value in the location, and 2) add the transaction record to the read set.

On commit, a transaction validates the read set, which makes sure no transaction records in the set has been updated after the current transaction read them. If the validation succeeds, the current transaction unlocks all transaction records in its write set. Otherwise, the current transaction aborts.

On the abort of a transaction, the old values recorded in the undo log will be written back to the corresponding addresses, and the transaction records in the write set will be unlocked.

2.2 Benchmark Suite

In the effective evaluation of an STM system, the selection of proper workloads is very important. There are several criteria for such workloads. First, a workload should represent an application that an end user will actually execute on her system today, and in the near future. Second, a workload should demonstrate enough transactional characteristics to stress a TM system. For example, a TM workload should launch significant number of transactions during its execution, and should spend significant amount of time inside transactions. Following these criteria, we selected a set of workloads for our study, as listed in Table 1.

Game Physics Engine (GPE) performs collision detection by solving ordinary differential equations. This application represents the core computation found in many 3D graphics applications such as 3D games.

Smoothed Particle Hydrodynamics (SPH) performs particle dynamics simulation, a typical application found in computer animated fluid.

Both workloads were strategically developed by Intel for multi-core processor research. They were originally written using fine-grained locks (pthread mutexes). We created a TM version and a single global lock (GLOCK) version by converting critical sections into atomic blocks and by forcing the critical sections to acquire a single global lock, respectively. GLOCK version of the workload serves as a good baseline for TM measurements, since it provides the lower bound on performance while maintaining a similar programming model.

Sphinx is a speech recognition program, which is part of ALPBench [8] — a multithreaded benchmark suite from the University of Illinois which utilizes thread-level, data-level, and instruction-level parallelism. The sequential version of Sphinx was originally written at the Carnegie Mellon University. We ported Sphinx to our STM stack by converting critical sections guarded by pthread mutexes into atomic blocks.

Genome, Kmeans, and Vacation are all part of STAMP [10], a benchmark suite from the Stanford University developed specifically for TM from the ground up. Genome is a gene sequencing program, while Kmeans is a machine learning algorithm applied to a set of data points. Vacation models the database transactions of a travel agency. For comparison, we created a GLOCK version of each STAMP program by replacing a transaction with a critical section acquiring a single global lock. As directed in the benchmark suite release, all the workloads were executed with the default input parameters for the execution in a non-simulator environment, except for Kmeans where the parameters were slightly adjusted to induce meaningful execution time.

In Table 1, we also list two key statistics for each workload: the number of transactions and the percentage of time spent inside transactions. To collect these statistics, we manually instrumented all the programs with profiling code. For STAMP programs and GPE, the statistics were collected in single thread. Sphinx and SPH were measured with two threads, since they generated no-lock code for single thread.

3. PERFORMANCE BOTTLENECKS AND SOLUTIONS

We evaluated our C/C++ STM with the workloads described in Section 2.2. All the experiments were performed on an 8-way system with dual-socket and 8 GBytes of memory. Each socket had an Intel Xeon X5355 processor (quad-core, 2.66 GHz, 8 MB L2). The OS was Red Hat Enterprise Linux AS 4 (kernel version 2.6.9) with SMP support, and our TM compiler was a modified version of Intel C/C++ compiler v 10.0.

The initial performance results were not optimal, which led us to identify four major performance bottlenecks in our STM. We then came up with a series of optimizations and tunings, which improved the performance to a various extent. In the rest of this section, we describe each performance bottleneck and its corresponding solution, while validating the approach with in-depth analysis of our experimental results.

3.1 False Conflicts

In the initial performance analysis, workloads Genome and Vacation did not scale particularly well. From the TM statistics collected, we found that more than 99.9% of the data conflicts in Genome and Vacation were false conflicts. The same STM runtime, however, did not exhibit any scalability issues on SPLASH-2 [16] previously. As defined in [17], a *false conflict* occurs on an STM when two different addresses are mapped to the same transaction record. We improved upon [17] by further classifying the false conflicts in STAMP into 3 categories. Figure 2 shows the breakdown.

Our STM implements cache line based conflict detection; it does not distinguish two different addresses on the same cache line. This

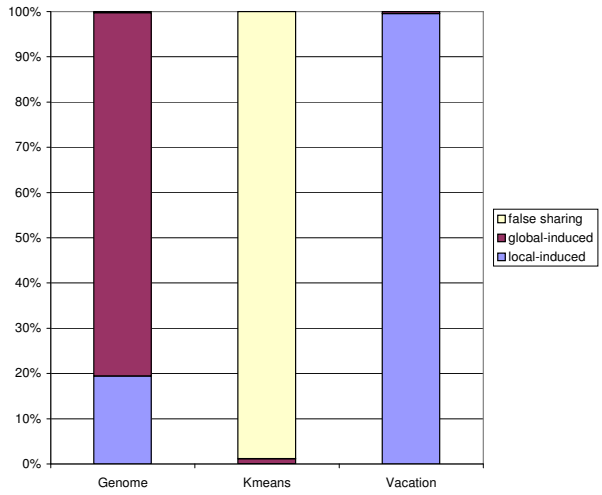

Figure 2: Breakdown of False Conflicts on STAMP

may lead to the first category of false conflicts, referred to as *false sharing* in Figure 2.

False conflicts can also happen between addresses from different cache lines, since the limited size of ownership table introduces aliasing. For such a false conflict, if either of the two addresses is thread-local, we categorized it as *local-induced*. If neither of them are thread-local, we categorized the false conflict as *global-induced*.

Figure 2 shows that for Genome and Vacation, more than 99% of the false conflicts are from accesses to different cache lines. Unlike the case of false conflicts induced from false sharing, those false conflicts can be reduced by proper support from compiler and runtime. More specifically, global-induced conflicts can be reduced by modifying the hash function used to map memory addresses to transaction records, and local-induced conflicts can be reduced by avoiding instrumentation of thread-local memory accesses (as we will discuss in Section 3.2). The figure also shows that most false conflicts for Kmeans are false sharing; that is, due to the memory accesses to the same cache lines. However, false conflicts only accounted for 40.4% of the total conflicts observed in Kmeans, and the workload scaled well even with those false conflicts.

In our original implementation, the hash function utilized 14 bits out of 32 bit address to generate a hash value. Each ownership table entry was cache line size (64 Bytes), and we stored only one transaction record (4 Bytes) per table entry; the rest of the space were reserved to avoid false sharing on transaction records. In our new implementation, we still used the same 14 bits to hash into a table entry, but we then used 4 additional bits to store 16 transaction records into each table entry. These additional 4 bits are now used to differentiate between different cache lines. Under this arrangement, no two addresses that map to transaction records residing in different table entries under the old hash function, would map to transaction records located in the same table entry under the new hash function. This way, we were able to cram in 16 times more transaction records into the same memory space, while not increasing false sharing.

Figure 3 shows the result of applying the new hash function to Genome workload. In this figure, the STM with old hash function exhibits serious scalability problem. The new hash function, on the contrary, was very effective in making Genome scale. In Figure 4, Vacation also benefits from the new hash function. Throughout the rest of this paper, we only report the performance results of STM with the new hash function.

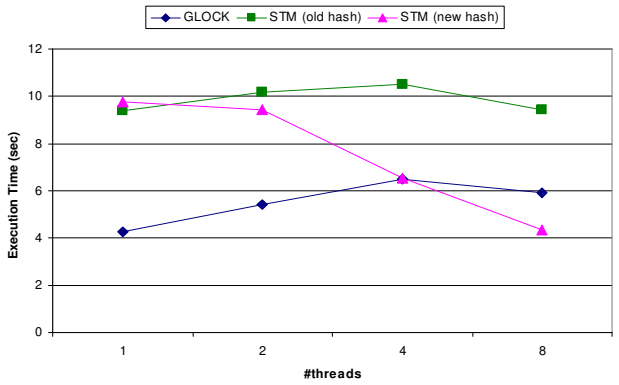

Figure 3: Application of New Hash Function on Genome

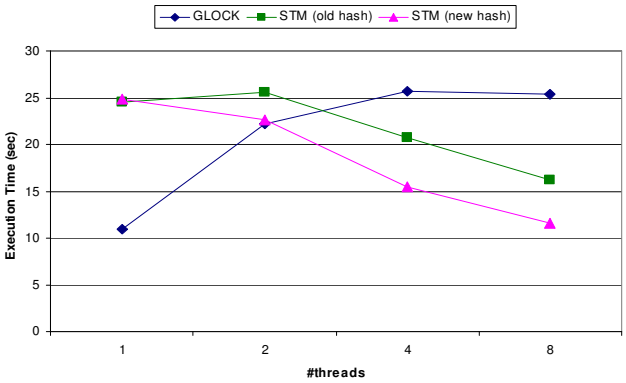

Figure 4: Application of New Hash Function on Vacation

3.2 Over-Instrumentation

Figure 2 also shows that the percentage of local-induced false conflicts is 99% for Vacation, and 19% for Genome. Thread-local memory accesses should not be monitored for transactional conflicts in the first place, since no other thread would be trying to write in that memory area. However, unless the results of whole-program pointer analysis are available, it is difficult for a compiler to differentiate thread-local memory accesses from regular shared memory accesses. As a result, a compiler is likely to generate more read / write barriers than necessary.

We quantified the barrier overhead of the compiler-driven approach with STAMP suite. STAMP workloads were originally targeted for an API-based STM — read / write barriers were manually annotated in the source code. So the original STAMP code represents the optimal number of barriers that are necessary to guarantee program correctness. To measure the barrier overhead from compiler-driven approach, we first mapped those barriers into plain read / write statements, and then let the compiler automatically generate barriers whenever it deems necessary. We then compared the number of barriers executed by running both versions of STAMP on our STM. Table 2 shows the results.

As evidenced by Genome and Vacation, the compiler-generated code may execute significantly larger number of read / write barriers. This over-instrumentation would directly attribute to higher STM overheads for bookkeeping and conflict detection, and more importantly, cause transactions to abort unnecessarily on false conflicts.

It turns out that the original STAMP code heavily relies on application-level knowledge to annotate read / write barriers. More specifically,

Workload	Read Barriers		Write Barriers		Read Overhead	Write Overhead
	Manual	Compiler	Manual	Compiler		
Genome	58,701,959	624,073,490	2,252,291	19,078,705	10.6x	8.60x
Kmeans	86,666,710	255,662,754	86,666,710	88,666,711	2.95x	1.00x
Vacation	785,775,435	925,584,125	26,300,714	122,543,905	1.18x	4.66x

Table 2: Transactional Barrier Counts on STAMP

two common code patterns were repeatedly observed in STAMP, where the compiler would over-instrument.

First, STAMP workloads use a transaction-aware memory allocator. Under this allocator, each thread maintains its own memory pool, and allocates memory only from the local pool. Obviously, memory allocation routines manipulating the local pool need not be instrumented with transactional barriers. However, without the proper information, it would be difficult for the compiler to infer this behavior.

Secondly, a shared data structure might remain constant during a particular program execution phase. Many shared data structures are typically initialized at program startup, and then remain constant throughout the rest of program execution. Alternatively, they could alternate between modification phase and constant phase. This kind of memory access pattern is hard to detect without application-level knowledge. For example, in Genome, a transaction measures the size of a shared hash table that remains constant during the execution of the transaction. Without knowing that the hash table remains constant, the compiler would generate unnecessary transactional barriers.

A practical solution to this over-instrumentation issue is to let the programmer directly convey such application-level knowledge to the compiler. For this purpose, we introduced a new language construct pragma tm_waiver. A block or function marked with pragma tm_waiver would not be instrumented by the compiler for memory accesses inside.

In more detail, tm_waiver pragma lexically overrides tm_function pragma. Hence, when a tm_function annotated function is called inside a function annotated with tm_waiver, the function will behave as if itself was also denoted as tm_waiver. As described in Section 2.1, our compiler generates both regular version and its transactional twin for a tm_function annotated function, and the runtime dynamically determines which version of the function to call. Same overriding applies to the pragma tm_waiver applied to a statement block.

This allows programmers to gradually optimize a TM application as their understanding of the application increases. However, the inappropriate use of tm_waiver can lead to incorrect transactional code. Similar to the previously proposed early release feature [5], tm_waiver is an optimization technique that should be applied with caution.

Figure 5 shows the performance result of applying tm_waiver to Genome. In annotating the code with tm_waiver, thread-local memory allocation routines and accesses to constant data structures were specifically targeted. From the figure it is clear that by reducing the number of barriers, this approach improves the performance of Genome even further after the aforementioned new hash function (Section 3.1) was applied. Originally, the STM managed to match the performance of GLOCK at 4 threads. After the application of tm_waiver, STM performance matches GLOCK at 2 threads, and continues to outperform GLOCK as the number of threads is increased.

In Figure 6, we can see that Vacation also benefits from tm_waiver, where accesses to the thread-local random number generator were denoted with tm_waiver. On the contrary, Kmeans did not benefit further from tm_waiver since most of its false conflicts were due to false sharing. Figure 7 shows the result.

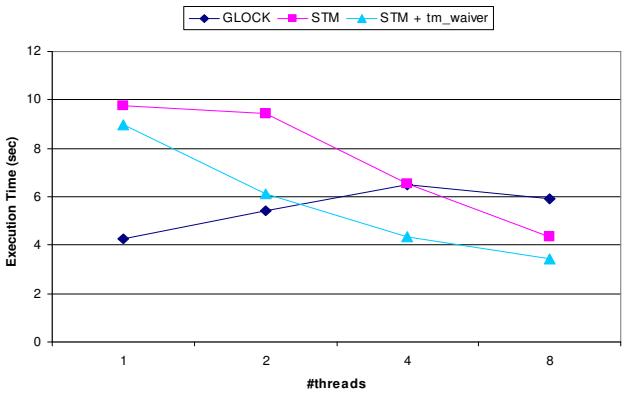

Figure 5: Application of tm_waiver on Genome

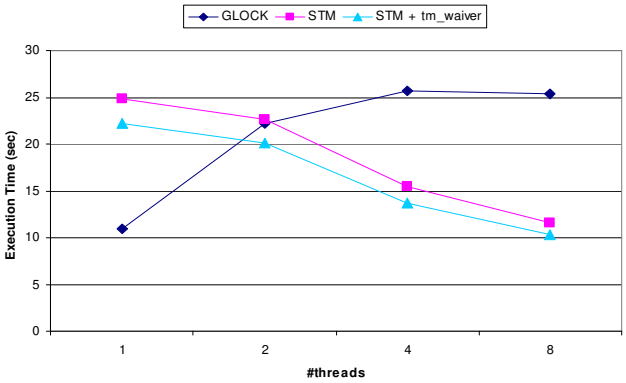

Figure 6: Application of tm_waiver on Vacation

3.3 Privatization-Safety Cost

Privatization is a common programming idiom where a thread privatizes a shared object inside a critical section — usually by nullifying the global pointer to the shared object — then continues accessing the object outside the critical section. Privatization is a favorable programming technique for STM because it reduces the number of transactional memory accesses. However, privatization can cause data races and generate unexpected results on an optimistic-read STM [7]. Pessimistic-read STM [7] using reader locks, on the contrary, can handle privatization correctly, but its read barriers could be much slower than optimistic-read STM.

McRT-STM guarantees correctness for privatization by implementing the *quiescence* algorithm [15, 6]. Under the algorithm, a committing transaction waits before it starts executing non-transactional

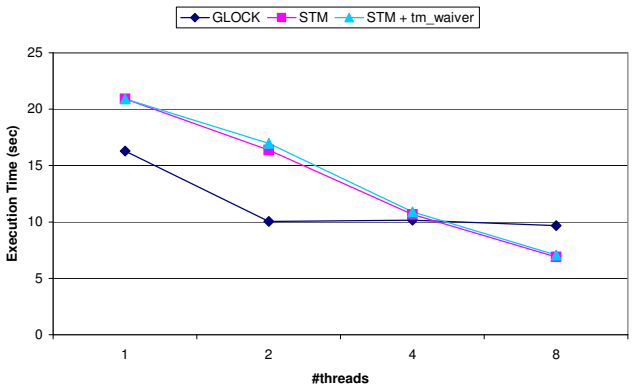

Figure 7: Application of tm_waiver on Kmeans

code, until all other transactions acknowledge that their speculative reads do not overlap with the committing transaction's write set. This way, isolation is provided over the privatized data between non-transactional accesses from one thread and transactional accesses from another thread.

In our actual implementation, the quiescence algorithm is implemented with timestamps. The STM maintains a global timestamp, which is incremented whenever a transaction commits or aborts. Each transaction also maintains a local timestamp. This timestamp is updated to the latest global timestamp in post validation of every transactional read or write.

After a transaction commits, it performs quiescence before the thread enters non-transactional code. Specifically, the committing transaction records the global timestamp at the moment it commits, called *committing timestamp*, and compares it to the local timestamps of all other active transactions. If any of them is less than the committing timestamp, the committing transaction waits until all other local timestamps become greater than or equal to its committing timestamp.

From our performance results we observed that quiescence consumed a significant amount of execution time for the workloads we selected. In more detail, the cost of quiescence could be broken down into two. First, it caused coherent cache misses on local timestamps which were written by each transaction and read by all others. Secondly, it made transactions wait for each other.

Costly as it is, privatization is not used in many programs. For those programs, quiescence is a pure overhead squandered in prevention of a problem that would not even occur. As an optimization feature, we introduced a new construct that allows the programmer to annotate an atomic block as not using privatization, so that quiescence could be skipped. When the programmer is not certain and leaves the atomic block unmarked, the runtime would perform quiescence by default.

By carefully reviewing the source code of our selected workloads, we determined that privatization is not used in any of them. Using the interface proposed above, those workloads have been instrumented not to enter quiescence phase upon transaction commit. Figure 8 shows the performance improvement by applying such optimization over the case where quiescence is performed.

We can see that the proposed quiescence optimization significantly improves performance. On average (geometric mean), the scheme improves performance by 15%, 26%, and 32% for 2, 4, and 8 threads respectively, while the peak performance improvement reaching 2.07x on Vacation with 8 threads. Except for SPH, the benefit from the optimization tends to increase with increasing number of threads, since the cost of quiescence loop itself increases with more threads. SPH was suffering from a different serialization

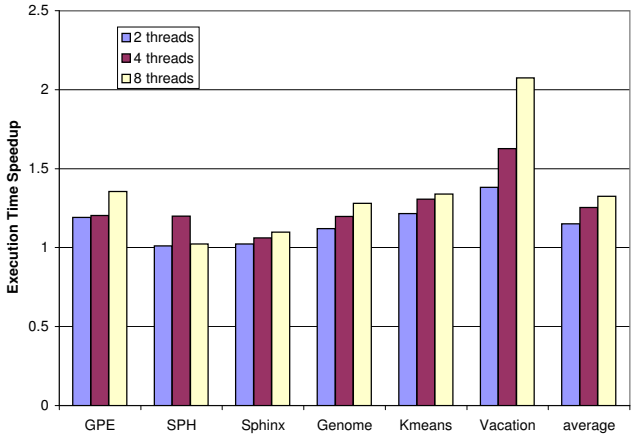

Figure 8: Performance Impact of Quiescence

issue, which we will explain in Section 3.4. It is also interesting to note that quiescence assumes a significant amount of execution time both for workloads that were specifically written for TM and for workloads that were originally written to use mutual-exclusion locks and then transactionized.

3.4 Amortizing Bookkeeping Cost

Current multithreaded workloads utilizing locks are written in such a way that critical sections are as short as possible. When transactionized, these critical sections amount to extremely short transactions. Table 3 describes the load and store barrier counts per committed transaction for SPH workload. (First column denotes the number of threads.) Note that the code generates no-lock code for 1 thread.

#t	#commits	TxLD	TxST	LD /txn	ST /txn
1	0	0	0	N/A	N/A
2	2,221,485	2,356,785	4,384,905	1.06	1.97
4	7,404,950	7,860,685	14,616,350	1.06	1.97
8	17,771,880	18,868,815	35,079,240	1.06	1.97

Table 3: Transactional Barrier Counts on SPH Workload

SPH workload is a particle simulation workload originally written in pthread. The workload has been transactionized by converting each critical section into an atomic block. From Table 3 we can see that a transaction in SPH on average comprises only 1 transactional load and 2 transactional stores. Although small, those transactions are executed millions of times. During the experiments, transactions were rarely aborting: about 0.01%.

In an STM, there are fixed costs in executing a transaction: startup costs and teardown costs. Startup costs are attributed to routines that initialize transaction descriptor and logs. Teardown costs are attributed to bookkeeping routines that are executed upon transaction commit / abort.

We found that when transactions are extremely short, these fixed costs of transaction startup and teardown get poorly amortized over the length of the transaction. The problem is that these routines are usually serial in their nature. For example, in an STM system that maintains a global timestamp, increasing the timestamp at commit / abort time is usually implemented with CAS instructions.

When transactions are very short, these CAS instructions become the dominant overhead in executing a transaction.

To handle this type of short transactions, we implemented a special execution mode in STM. Under this mode, transactions are serialized with a scalable global lock implementation, while the transaction startup / teardown routines and transactional barrier invocations are neglected altogether. The compiler inserts a hint for the runtime denoting that the length of transactions are below certain threshold. This threshold was empirically determined to suit our particular STM implementation. The runtime then forces those transactions to execute in the special execution mode.

Figure 9 shows the result of applying this special execution mode to SPH. GLOCK trend line denotes the performance of the workload when all the critical sections are synchronized with a single pthread lock. FGL trend line denotes the original workload with Fine-Grained Locking, while the STM trend line denotes the STM system without the special execution mode.

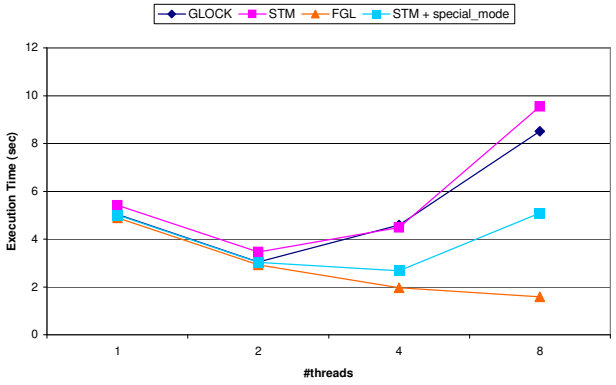

Figure 9: Application of Special Execution Mode for Short Transactions (SPH)

In the figure, we can observe that although transactions were rarely aborting, the scaling of the baseline STM closely resembles that of GLOCK. On the contrary, STM with special execution mode significantly outperforms both the baseline STM and GLOCK. This is due to the fact that the system does not suffer from transaction startup / teardown costs and barrier overheads. However, after 4 threads the serial nature of the execution mode becomes prevalent, hence rendering the performance trend more close to GLOCK.

4. DISCUSSIONS

In addition to the major performance bottlenecks we identified in Section 3, in this section we describe the language design issues we faced while porting large-scale workloads (Section 4.1 and 4.2), and the statistics that were crucial in characterizing the complex behavior of realistic transactional workloads (Section 4.3).

4.1 Condition Variables Inside Transactions

Condition variables are an instance of a signaling mechanism used to deter the execution of a thread consulting the state of a condition variable. In the pthread implementation of condition variables, each condition variable should be guarded with a mutex.

The top portion of Figure 10 depicts a typical use of pthread_cond_wait() function to pause a thread's execution. Notice that pthread_cond_wait() function is not an atomic function unless the pthread library itself was annotated with pragma tm_function. Therefore, the critical section in the top portion of Figure 10 cannot be transactionized.

```
pthread_mutex_lock (&mutex);
condition = ...;
if (! condition)
    pthread_cond_wait(&cond, &mutex);
pthread_mutex_unlock (&mutex);
```

```
pthread_mutex_lock (&mutex);
...
pthread_mutex_unlock (&mutex);
```

Figure 10: Condition Variable Inside Transaction

Now let us consider another critical section which is also guarded by the same mutex. The bottom part of Figure 10 shows such critical section. It could be tempting to transactionize the critical section, but transactionizing the critical section would break the exclusion with the previous critical section with condition variable wait. More generally put, when a critical section is transactionized, all the critical sections that synchronize against the same lock should be transactionized. Conversely, if one critical section could not be transactionized, all the other critical sections that synchronize against the same lock should not be transactionized.

This *all-or-nothing* nature of critical section porting can sometimes significantly limit the degree to which an application could be transactionized. In the worst case, this could lead to a situation where un-transactionized portion of the code serializes the entire workload regardless of the performance benefits obtained from the transactionized portion of the code.

To allow more critical sections to be transactionized, pthread library itself should be transaction-aware. For example, pthread_cond_wait() function could be transactionized with retry construct [3].

```
#pragma tm_function
int pthread_cond_wait (pthread_cond_t *cond,
                       pthread_mutex_t *mutex)
{
    #pragma tm_atomic {
        if (! IS_SET(cond)) retry;
    }
}

#pragma tm_function
int pthread_cond_broadcast (pthread_cond_t *cond)
{
    SET(cond);
}
```

Figure 11: Application of Retry Construct in Condition Variable Synchronization

Figure 11 shows a possible implementation of transaction-aware condition variable synchronization. In this figure, pthread_cond_wait() has been replaced with a retry construct. The waiting transaction would first transactionally read the condition variable, then the retry construct would force the transaction to pause until the condition variable gets updated by the opponent. The transaction

which calls pthread_cond_broadcast() will update the condition variable, then the other transactions that paused execution at the retry construct would resume to re-execute the atomic block.

Notice that there are some subtleties in the semantics of transaction-aware condition variable synchronization. Especially, from the implementation in Figure 11, the paused transactions would wake up after the transaction enclosing the pthread_cond_broadcast() commits, as opposed to the original semantic where paused threads resume execution after the pthread_cond_broadcast() function returns. Moreover, in the case of pthread_cond_signal(), the order in which transactions resume would have to be defined.

4.2 Quantifying the Function Annotation Overhead

As described in Section 2.1, in our compiler-based approach a programmer has to manually annotate atomic functions with the tm_function pragma. This overhead would be negligible when the code size is small. However, we found that the overhead could be formidable for large-scale workloads. Table 4 shows the function annotation ratio for the workloads described in Section 2.2.

Workload	Annotated Functions	Total Functions	Annotation Ratio
GPE	14	1,691	1%
SPH	0	30	0%
Sphinx	22	376	6%
Genome	11	133	8%
Kmeans	0	38	0%
Vacation	74	183	40%

Table 4: Function Annotation Ratio for Workloads

In this table, most of the workloads exhibit less than 10% annotation ratio. For this type of workloads, the function annotation approach would incur little overhead. However, the ratio could be as high as 40% for Vacation.

For those workloads that exhibit high annotation ratio, we propose to utilize *all double-versioned* approach. Under all double-versioned approach, a programmer would selectively turn on a compiler flag, then the compiler would generate transactional twin for all the functions regardless of programmer annotation. The runtime would then dynamically determine which version of the function to call [15].

This approach would double the static code size, but we expect the performance impact to be limited to a minimum degree since the dynamic code size, which affects the instruction cache miss rate, would remain about the same.

4.3 Per Atomic Block Statistics

In Section 3 we described the techniques necessary to obtain adequate performance when executing complex workloads on a compiler-based STM. In the process of devising these techniques, we also noticed that traditional transactional characteristics, such as execution time and overall abort rate, are not helpful in characterizing the complex behavior of realistic TM workloads. More specifically, we observed that "averaging" transactional statistics could be quite deceiving. This was especially true for abort rates. Figure 12 shows the execution time result for Sphinx.

Sphinx workload was originally written in GLOCK. The performance of GLOCK and STM are equal at single thread, since Sphinx generates no-lock code for 1 thread. However, neither GLOCK nor STM scales with increasing number of threads, although the overall abort rate for STM was as low as 1.39% for 4 threads.

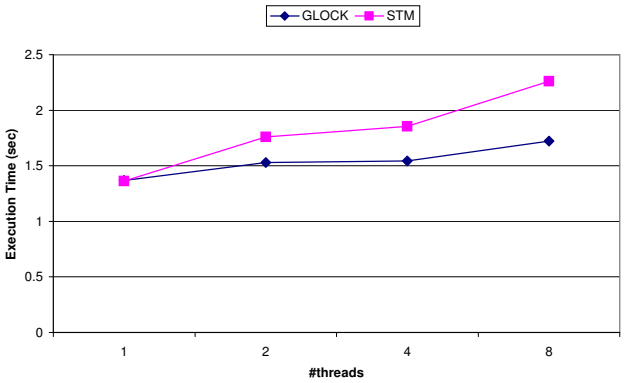

Figure 12: Execution Time on Sphinx

In our STM runtime, we implemented routines to collect per atomic block transaction statistics. By measuring per atomic block abort rate, we were able to pinpoint the specific atomic block that was being the scalability bottleneck of the entire workload. Table 5 shows the per atomic block transaction statistics collected for Sphinx.

Atomic Block	Tx Begin	Tx Commit	Tx Abort	Abort Rate	Code Size (lines)
602	1,314	1,312	2	0.15%	O(1)
542	222,481	221,043	1,438	0.65%	O(1)
559	220,908	220,908	0	0.00%	O(1)
601	12,306	6,194	6,112	49.67%	O(1000)
571	42,917	42,889	28	0.07%	O(1)
588	42,770	42,770	0	0.00%	O(1)
301	1,313	1,312	1	0.08%	O(1)

Table 5: Per Atomic Block Transaction Statistics for Sphinx (4 threads)

The first column gives the ID of each atomic block determined by their location in the source code, while the last column gives the size of each atomic block measured by the number of code lines. In essence, most of the atomic blocks committed without problems, except for atomic block 601 which was dominant in its size. Due to the absolute size of atomic block 601, high commit rates in other atomic blocks did not add to the scalability of the workload when atomic block 601 suffered from high abort rate.

We found that those workloads that were not originally written for TM tend to exhibit more irregular abort rates. For example, short critical sections such as induction variable increments rarely abort when converted into transactions. At the same time, these transactions are too small to dominate the overall program execution time. On the contrary, complex critical sections, when transactionized, usually suffer from high abort rates while being dominant in determining the overall execution time and program scalability.

The problem is that simple transactions tend to be executed much more frequently than complex transactions. When this happens, the high abort rate for complex transactions are overshadowed by the high commit rate of simple transactions. Blindly measuring the overall transaction abort rate simply fails to account for this fact. Figure 13 shows similar, but more subtle behavior observed on GPE.

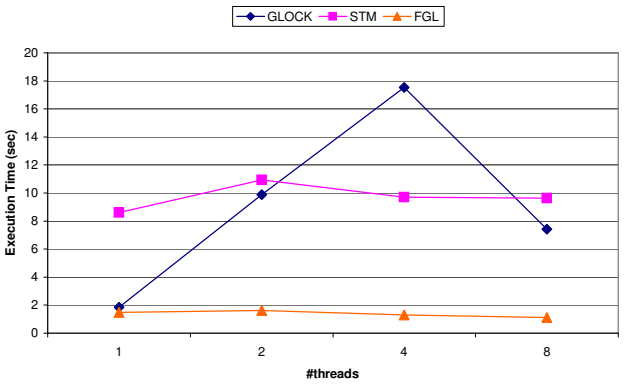

Figure 13: Execution Time on GPE

In this figure, FGL trend line stands for the performance of the original Fine-Grained Locking. As can be seen, the workload does not scale on the STM runtime. Nonetheless, when the overall transaction abort rate was measured, only 1.6% of the transactions were aborting at 8 threads. Per atomic block breakdown of abort rates could reveal the problem. Table 6 shows the breakdown of abort rates for different number of threads.

In this table, the same as before, the Atomic Block column gives the ID of each atomic block based on the location in source code. Notice that atomic block 74 does not exhibit high abort rates when the number of threads is small. However, with increasing number of threads, atomic block 74 fails to scale. This scalability problem was shadowed by the sheer amount of transaction commits in atomic block 215.

Atomic Block	Tx Begin	Tx Commit	Tx Abort	Abort Rate
74	22,596	22,596	0	0.00%
1120	171,607	171,607	0	0.00%
1131	170,666	170,666	0	0.00%
215	10,296,420	10,296,420	0	0.00%

(a) 1 thread

Atomic Block	Tx Begin	Tx Commit	Tx Abort	Abort Rate
74	29,763	22,598	7165	24.07%
1120	171,658	171,658	0	0.00%
1131	170,716	170,716	0	0.00%
215	10,304,252	10,299,481	4771	0.05%

(b) 2 threads

Atomic Block	Tx Begin	Tx Commit	Tx Abort	Abort Rate
74	98,810	22,601	76,209	77.13%
1120	172,023	172,023	0	0.00%
1131	171,083	171,083	0	0.00%
215	10,327,854	10,321,462	6392	0.06%

(c) 4 threads

Atomic Block	Tx Begin	Tx Commit	Tx Abort	Abort Rate
74	156,165	22,599	133,566	85.53%
1120	172,094	172,094	0	0.00%
1131	171,155	171,155	0	0.00%
215	10,334,072	10,325,713	8,359	0.08%

(d) 8 threads

Table 6: Per Atomic Block Transaction Abort Rate for GPE

5. RELATED WORK

TM [4, 7] was originally proposed as a hardware component [4]. STM was first proposed by Shavit and Touitou in [12], which has seen subsequent releases of more implementations [5, 11, 2, 9]. Under these API-based systems, however, a programmer had to statically denote the set of memory locations accessed by transactions. More recently, compiler-based STMs have been proposed [15, 1]. Under such configuration, a compiler automatically generates transactional barriers for transactional memory accesses. However, previous STM research was mostly limited to small, kernel-level workloads. This paper focuses on language supports and optimization techniques that are necessary for a complex workload to perform well on compiler-based STM systems.

General discussions on the false conflict issues in STM can be found in [17]. In the paper, discussions on similar hash function improvements can also be found. However, in our current paper, we discovered a new, compiler assisted performance optimization opportunity by further breaking down the categories of false conflicts. Our tm_waiver pragma is strictly aimed at this new opportunity.

Privatization issues in STMs are discussed in [15, 6, 14, 13]. Quiescence [11] is one way to solve the privatization problem. However, the performance penalty of quiescence on real workloads has not been quantified previously. Our paper identifies that quiescence routine is a significant overhead both for workloads specifically written for TM and for workloads obtained by transactionizing critical sections.

Moreover, we propose to expose an interface to allow programmers to selectively turn off quiescence routines when there are no privatization instances. RSTM library [9] exposes similar but opposite functionality; the interface exposes functionality to selectively turn on quiescence routines, while suppressing quiescence by default. To guarantee correctness, our implementation performs quiescence by default.

6. CONCLUSION

In this paper we have shown that current compiler-based software transactional memory techniques are not sufficient to obtain performance from a complex, realistic transactional memory workload. We identified four bottlenecks that are specific to such configuration, and proposed novel solutions to address the issues. Especially, we introduced a new language construct to directly convey program-level knowledge to compiler in an aim to reduce excessive barrier generation. Moreover, we quantified the effect of quiescence on performance, and proposed a new interface to optimize quiescence instances. Under this optimization the performance improvement was 32% on average, while the peak performance improvement reaching 2.07x. We also pointed out that traditional transactional characteristics are not helpful in characterizing complex transactional memory workloads, and proposed a new set of metrics that we found crucial in characterizing the workload behavior.

7. ACKNOWLEDGEMENTS

This research was conducted under the Programming Systems Lab Research Intern Program at Intel Corporation. Intel Corporation generously supplied all the software and experimental equipment used in this work. We also thank Ravi Narayanaswamy and Xinmin Tian for developing the compiler infrastructure used in this study.

8. REFERENCES

[1] W. Baek, C. C. Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun. The OpenTM transactional application programming interface. In *Proceedings of the 16th Intl. Conference on Parallel Architecture and Compilation Techniques*, September 2007.

[2] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, 2006.

[3] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005.

[4] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[5] M. P. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, July 2003.

[6] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg. McRT-Malloc: A scalable transactional memory allocator. In *Proceedings of the 5th International Symposium on Memory Management*, 2006.

[7] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[8] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The ALPBench benchmark suite for complex multimedia applications. In *Proceedings of the 2005 IEEE International Symposium on Workload Characterization*, October 2005.

[9] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer, III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. In *TRANSACT: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[10] 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 *Proceedings of the 34th Intl. Symposium on Computer Architecture*, June 2007.

[11] 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 *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2006.

[12] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[13] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 78–88, 2007.

[14] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, 2007.

[15] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *Proceedings of the 2007 International Symposium on Code Generation and Optimization*, pages 34–48, March 2007.

[16] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA-22*, pages 24–36, 1995.

[17] C. Zilles and R. Rajwar. Implications of false conflict rate trends for robust software transactional memory. In *Proceedings of the 2007 IEEE International Symposium on Workload Characterization*, September 2007.