TRANSACT 2011

Interchangeable Back Ends for STM Compilers *

Gokcen Kestor,[2,3] Luke Dalessandro,[4] Adrián Cristal,[1,3] Michael L. Scott,[4] and Osman Unsal [3]

[1] IIIA - Artificial Intelligence Research Institute CSIC - Spanish National Research Council [2] Universitat Politècnica de Catalunya,
[3] Barcelona Supercomputing Center, [4] University of Rochester,
{gokcen.kestor,adrian.cristal,osman.unsal}@bsc.es, {luked,scott}@cs.rochester.edu

Abstract

If transactional memory is to become a mature technology, TM research groups must be able to run each other's code and to perform apples-to-apples comparisons of implementation alternatives. For C++ on the x86, significant steps in this direction have been made by compilers from Intel, the University of Dresden, and the GNU Project, which aim to accept the same language API and target the same runtime ABI. Unfortunately, these three compilers currently connect to only two main STM libraries.

In the interest of greater interoperability, we have adapted the word-based "back end" libraries of the RSTM suite to the common ABI, and tested them with the Intel compiler. Notable changes to RSTM included support for true subword reads and writes along with in-library allocation, management, and rollback of checkpoints. To the best of our knowledge, RSTM supports the widest diversity of back ends currently available; our work makes these available, for the first time, to programs written with language-level transactions.

For testing purposes, we use applications from the RMS-TM and STAMP benchmark suites (the latter adapted to the C++ standard API). We describe our experience at both the ABI and API levels, and present preliminary performance comparisons relative to the Intel standard back end. Public release of our tools will allow developers to experiment with the full range of STM implementation options, and to pursue ongoing research in TM semantics and hardware/software hybrid TM systems.

1. Introduction

Transactional Memory (TM) [8, 10, 11] allows programmers to mark compound statements in parallel programs as `atomic` (in C++, `__transaction`), with the expectation that the underlying run-time implementation will execute such transactions concurrently whenever possible, generally by means of speculation—optimistic but checked execution, with rollback and retry when conflicts arise. The principal goal of TM is to simplify synchronization by raising the level of abstraction, breaking the connection between semantic atomicity and the means by which that atomicity is achieved. Secondarily, TM has the potential to improve performance, most notably when the practical alternative is coarse-grain locking.

We believe that simple hardware TM will eventually be standard on many-core machines. In the meantime, there are hundreds of millions of multicore machines already in the field. For the sake of backward compatibility, emerging TM-based programming models will need to be implemented in software on these machines. Moreover, a growing consensus holds that STM will be needed as a "fall-back" mechanism when hardware transactions fail due to buffer space limitations, interrupts, or other transient or deterministic causes [5, 7, 15, 24].

As researchers work to develop robust, mature STM, it becomes increasingly important to be able to share applications, compilers, and runtimes among groups, and to be able to modify one layer of the system stack while keeping the others constant, for "apples-to-apples" comparison.

Until recently, most STM systems were implemented as user-level libraries: the programmer was expected to manually instrument not only transaction boundaries, but also individual loads and stores within transactions. This library-based approach was adequate for early experiments with microbenchmarks, but it becomes increasingly tedious and error prone for larger applications [2]. The use of different library interfaces in different research groups has also made it difficult to share applications across groups, or to make reliable performance comparisons: experiments with different versions of the application source code inevitably raise questions of fairness and confidence.

A recent draft standard for transactions in C++ [1], and the release of compilers conforming to that standard, promises to significantly ease the construction of large transactional programs, and reduce the problem of source-level incompatibility among groups. Compilers also improve the interoperability of hardware and software TM, by automatically generating the instrumented loads and stores that are required by the latter but not the former. In the software case, the fact that calls to the back-end system are being generated by a compiler rather than a human programmer means that the back end can provide a wide, performance-oriented ABI instead of a narrow convenience-oriented API.

Unfortunately, much of the work on STM systems over the past 7 years remains incompatible with recent compilers because of interface issues. Indeed, the four publicly available C++ TM compilers support remarkably little back-end diversity. Oracle's compiler, which generates code only for the SPARC, employs the SkySTM back end [17]; Intel's compiler, for the x86, employs a modified version of the STM presented in Ni et al. [20]; and the Dresden and GNU compilers, also for the x86, employ TinySTM [9]. At the same time, the three x86 compilers and their two back ends employ (for the most part) a common ABI designed by Intel [13], which raises the prospect of interoperability.

To the best of our knowledge, the RSTM suite [23] comprises the widest diversity of STM algorithms currently available (13 in the version 5 release). In the interest of wider experimentation, we have adapted the "word-based" algorithms to the Intel ABI, allowing them to be used with any conforming compiler. To minimize per-algorithm effort, we introduce a "shim" layer that embodies most of the adaptation. As of this writing, we have successfully connected the Intel C++ TM compiler to three RSTM back ends: LLT (lazy detection, lazy versioning, with timestamps), which re-

* This work was conducted while Gokcen Kestor was a visitor at the University of Rochester, supported by a High-Performance Embedded Architecture and Compilation (HiPEAC) Collaboration Grant. This work was also supported by the cooperation agreement between the Barcelona Supercomputing Center National Supercomputer Facility and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625, and at the University of Rochester by NSF grant CCR-0963759.

```
1 int a;
2 int foo()
3 {
4   __transaction{
5     a = a + 5;
6   }
7 }
```

(a) C++ TM standard application programming interface (API)

```
 1 int foo()
 2 {
 3   _ITM_transaction * td = _ITM_getTransaction();
 4   int doWhat = beginTransaction(td, prop, str_loc);
 5     /* a = a + 5; */
 6     int a_tmp = (int)_ITM_RfWU4(td, (uint32 *)&a);
 7     a_tmp = a_tmp + 5;
 8     _ITM_WaWU4(td, (uint32 *)&a), (uint32)a_tmp);
 9   _ITM_commitTransaction(td, &outer_commit);
10 }
```

(b) Intel TM application binary interface (ABI)

Figure 1: Automatic read/write instrumentation of a simple TM program

sembles TL2 [6]; ET (extendable timestamps), which resembles TinySTM; and Precise (a.k.a. NOrec [4]), which provides unusually strong privatization semantics, and works particularly well as the software half of a hybrid TM system [5]. Future RSTM releases will include nearly universal support for the Intel ABI.

Our compiler-ready back ends allow us, for the first time, to run large applications on top of RSTM without hand-instrumenting loads and stores. As a first installment toward "apples-to-apples" comparison, we present performance results in Section 4 for both RSTM and the Intel back end on several applications from the RMS-TM benchmark suite [14]. We also present results for a selection of microbenchmarks and for applications from the STAMP suite [19]. For STAMP we consider both the original code, which uses hand instrumentation of (only) "important" loads and stores, and new versions written to the C++ TM standard. One new version lets the compiler instrument everything inside transactions; another uses Intel's `__transaction [[waiver]]` extension to disable instrumentation of many "unimportant" loads and stores. Our results suggest that the scalability of STAMP depends critically on minimizing instrumentation.

2. Design and Implementation

2.1 Draft Specification for TM in C++

The draft standard for C++ TM [1], written jointly by representatives of Intel, Oracle, and IBM, defines language extensions for TM applications. In particular, the `__transaction{}` construct brackets sequences of statements to be executed "all at once."

A transaction can be declared as either `atomic` (the default) or `relaxed`. Atomic transactions are restricted to perform only *safe* operations—loosely, those that a compiler and runtime are sure to be able to execute speculatively, and roll back on abort. In a data-race-free program, an atomic transaction never appears to interleave with execution in other threads or with behavior in the outside world.

Relaxed transactions are allowed to perform *unsafe* operations. They may or may not be executed speculatively. Operations inside a relaxed transaction are isolated from other transactions, but may, if unsafe, appear to interleave with (nontransactional) execution in other threads or with the outside world—even if the overall program is data-race free.

Functions called in an atomic transaction must be declared with the `transaction_safe` attribute, and cannot themselves contain unsafe operations, or calls to unsafe functions. Functions called in a relaxed transaction may be declared with the `transaction_callable` attribute, to increase the likelihood that the compiler will be able to execute the transaction speculatively. A `transaction_callable` function, like a relaxed transaction, is permitted to perform unsafe operations. The compiler can be expected to generate two clones of a `transaction_safe` or `transaction_callable` function—one for use outside transactions, one (with instrumented loads and stores) for use inside. The C++ draft standard calls for transactional function pointers to be statically typed with the same `transaction_safe` or `transaction_callable` attributes as the functions being assigned into them.

Some unsafe operations are said to be *irrevocable*, meaning that they cannot be rolled back. Examples include I/O and writes to `atomic` variables. If a relaxed transaction performs an irrevocable action, the STM implementation can be expected to preclude concurrent execution of certain other transactions [27]. Note that not all unsafe operations are necessarily irrevocable. For example, a read of a `volatile` variable is an unsafe operation but it will probably not be irrevocable.

The Intel compiler, which we use for our experiments, implements certain extensions to the C++ TM standard. For example, a function can be declared with the `transaction_pure` attribute, meaning that the programmer guarantees it to be idempotent, and thus safe to execute—even within an atomic transaction—without instrumentation on its loads and stores. Finally, the `__transaction [[waiver]] {}` construct can be used to bracket a sequence of statements inside a transaction that should not be rolled back on abort. Waivered code is essentially unstructured open nesting; example use cases include debugging, statistics gathering, and semantically neutral operations like tree rebalancing.

The current version of Intel's compiler does not implement transactionally typed function pointers. It supports the transactional use of function pointers by dynamically detecting if the indirect call target has a transactional clone, calling it if it does, and switching to *serial irrevocable* mode to perform the indirect call nontransactionally if it doesn't. This has two side effects: indirect calls through function pointers are only valid in `relaxed` transactions as they might require *serial irrevocable* execution, and incorrectly annotated source may lead to poor performance due to transactions silently switching to *serial irrevocable* mode.

2.2 Intel ABI Overview

Figure 1(a) shows a simple program fragment using the C++ TM API. The Intel compiler automatically generates an equivalent version instrumented for the Intel ABI (Figure 1(b)). Implementations of the functions in the ABI are provided by the underlying STM library. This subsection describes the instrumentation performed by the Intel compiler; the following section details how we link the instrumented code to the RSTM back ends.

The code in Figure 1(a) executes a transactional read of variable `a`, increases its value by 5, and writes back the result (line 5). In Figure 1(b), the thread performing the transaction allocates a *transaction descriptor* by calling `_ITM_getTransaction` (line 3). The `beginTransaction` function (line 4) takes several parameters: the transaction descriptor, a set of bit values encoding infor-

mation about the transaction's properties, and the source location where the atomic block begins. Given these, `beginTransaction` saves the machine state (callee-saves registers, stack pointer) and starts the transaction. If the transaction aborts internally, execution will resume with a second return from `beginTransaction`—it effectively has `setjmp` semantics in this way.

At line 6, the compiler knows that the read of `a` will be followed by a write. It therefore instruments the access with a call to `_ITM_RfWU4`—Read for Write, 4 bytes. In an eager-acquire STM this routine could pre-acquire a write lock on `a`, avoiding the need to promote a read lock later, and return the value of `a`, which the compiler saves in temporary variable `a_tmp`.

The next write operation is instrumented with `_ITM_WaWU4`—Write after Write, 4 bytes, which can avoid the complexities of lock promotion. The `_ITM_WaWU4` (line 8) updates `a` with its new value (`a_tmp`). The last call in the generated code (`_ITM_commitTransaction`) attempts to commit the transaction. If the function detects that the transaction has conflicts, then the transaction will abort and perform a `longjmp` back to `beginTransaction`. If no conflicts are detected, the transaction commits and execution continues with whatever lies after line 10.

2.3 RSTM Back Ends

RSTM includes a variety of STM algorithms, some of which have several variants. The selection of an STM library can be handled simply by re-compiling the code with a different back end. The object-based back ends are not compatible with the Intel ABI. Among the word-based back ends, we began with ET, LLT and NOrec, which reflect popular but divergent points in the STM design space.

LLT is a lazy versioning system patterned after TL2 [6]. A transaction begins by reading the value t in a global "clock." Ownership records (orecs), found by address hashing, indicate the last time at which one of the corresponding locations was modified. If a transaction encounters a location that was written after t, it assumes it is inconsistent, aborts, and retries. At commit time, the transaction locks the orecs for all locations that need to be modified, checks to make sure that all of the locations it read still have a timestamp earlier than t, increments the global time, stores the new time into all the locked orecs, writes out all the updates, and then unlocks the orecs.

ET is an eager conflict detection, eager versioning system with extendable timestamps, patterned after TinySTM [22]. Extendable timestamps avoid false positives in which a transaction is aborted despite having seen a consistent view of memory. If a transaction encounters a location that was written after start time t, it checks to see whether any previously read location has been modified since t. If not, it re-reads the global clock and continues, pretending it started at this new time t' instead of t.

NOrec [4], like LLT, is a lazy versioning system: it delays the resolution of conflicts until some transaction is ready to commit. It uses a single sequence lock [16], however, rather than ownership records to serialize commit and write-back. A transaction checks, after each read, to see if any writer has committed since start time t; if so, it performs value-based validation [21] to see if its prior reads, if performed right now, would return the values previously seen; if so, as in ET, it reads a new start time from the global clock and continues. Writers can speculate in parallel, but only one can commit at a time. This serialization ultimately limits scalability, but the simplicity of the system yields surprisingly good performance for up to a few dozen cores. Moreover, NOrec is inherently *privatization safe*; ET and LLT require additional code (and nontrivial overhead, not included in our experiments) for correct execution of programs in which data transition back and forth between shared and private status [18].

2.4 Design Details

Several technical challenges made the adaptation of RSTM to Intel ABI an interesting and nontrivial task. As noted in Section 1, the principal design decision was to introduce a "shim" library that maps the ABI function calls generated by the compiler (sometimes with a bit of "glue" code) to the function signatures provided by (one of) the RSTM back ends. This strategy allows most of the adaptation work to be done once rather than once per back end. The main disadvantage of the shim approach is potentially extra overhead. Fortunately, most of the back end routines in RSTM were intended to be inlined into manually instrumented source. We inline them into the shim instead, allowing us to incur only one function call, rather than two, at each instrumentation point.

Subword accesses. The existing RSTM back ends were designed to support only 4-byte loads and stores, but the Intel ABI requires 1-, 2-, 4-, 8-, 12-, 16-, 24-, and 32-byte accesses as well. Multi-word accesses are easily implemented (if slightly inefficiently) as sequences of word accesses. Subword accesses, however, raise the possibility of false sharing. If x and y occupy opposite halves of the same word, for example, then a transaction that modifies x may force the abort of a transaction that reads y, even though no conflict has actually occurred. Worse, if nontransactional code modifies y during the execution of a transaction that modifies x, commit-time write-back of the word containing x may overwrite the modification of y, leading to incorrect behavior—even though the program is data-race free.

Perhaps the simplest solution would be to maintain read and write logs at byte granularity, but this would quadruple the cost of instrumentation for common-case word-sized accesses. A second alternative might be to maintain separate logs for word, halfword, and byte level access, but this leads to significant complexity when a transaction accesses the same word at multiple granularities. We ultimately chose to add a bit mask to each entry in the read and write logs, to identify which part(s) of the word have been accessed. Appropriate bits are or-ed into the mask on each access. During write-back, only modified bytes are updated. During value-based validation (as in NOrec), only accessed bytes are compared.

For orec-based conflict detection (as in LLT and ET), we see no easy way to keep track of subword updates. Per-byte timestamps would again quadruple the cost of common operations, and bit mask schemes suffer from the fact that different words mapping to the same orec may have different update patterns, and different bytes may be updated at different times. For the sake of simplicity and modest overhead, we have chosen to maintain orec-based conflict detection at the word level only. This can lead to unnecessary aborts, but not to incorrect behavior.

Two small optimizations streamline the code path for load and store instrumentation. First, a "fast path" always checks for full-word granularity, since that is the common case. Second, to simplify masking, bitmaps are full-word width, with 8 identical bits in every byte.

Inevitability (irrevocability). The Intel ABI defines a function (`changeTransactionMode`) that can be used to make completion of a transaction inevitable prior to I/O, calls to uninstrumented functions, or other irreversible operations. The RSTM back ends currently support inevitability only when requested prior to performing any loads or stores. To support the Intel ABI routine, we arrange to abort a transaction that has already performed memory accesses, and restart it in inevitable mode.

Missing functionality. Support for some of the Intel ABI routines was missing entirely in RSTM and had to be added to the shim. The `addUserUndoAction` and `addUserCommitAction` routines allow user code to register functions to be called when a transaction

rolls back or commits. In the absence of explicit guidance in the ABI, we arrange to call these functions in the order in which they were registered. The `registerThrownObject` routine allows user code to register exception objects. Updates to such objects are not rolled back on abort, and for redo-log implementations buffered writes to such objects must be performed during aborts. The C++ draft standard as well as the Intel ABI expose an `abort` construct that allows a user to explicitly abort the innermost transaction scope. Support for this functionality requires that transactional logs support limited closed nesting—it is not necessary to determine the level of nesting at which a conflict occurs, but the logs must support partial rollback and merging on commit. We have not yet implemented the required functionality; however, it is not required for the benchmarks tested here. Future RSTM releases will fully support the `abort` construct.

3. Experimental Setup

In the Section 4 we use our Intel/RSTM shim to (1) explore the overhead of automatic (as opposed to manual) read and write instrumentation, and (2) compare the performance of the default Intel back end to three of the RSTM alternatives. In our experiments we employ three RSTM microbenchmarks (HashTable, DoubleList, and RBTree) and selected applications from the STAMP [19] and RMS-TM benchmark [14] suites.

The STAMP suite comprises eight applications with 30 configuration sets. The applications are drawn from bioinformatics, engineering, computer graphics, and machine learning. They vary significantly in transaction lengths, read- and write-set sizes, and degree of contention. All were written with explicit calls to a transactional library API. They needed to be modified by hand to employ the C++ TM standard API instead. In the time available we were able to complete three of the eight applications: Kmeans, SSCA2, and Vacation. Kmeans and SSCA2 were straightforward: their transactions are relatively simple, with no nested subroutine calls, transactional `libc` library calls, or unsafe operations. Vacation was more of a challenge (as would be the five remaining applications). We annotated functions called from within transactions in Vacation as either `transaction_safe` or `transaction_callable`, depending on whether they include unsafe operations. We then defined transactions as `atomic` or `relaxed` accordingly.

STAMP implements generic data structures using function pointers. A set of objects of opaque type, for example, is represented with a list of `void*` and a pointer to a function that can be used to test for object equality. STAMP's initial implementation uses pointers to uninstrumented functions in such contexts: the original developers determined that the lack of instrumentation would not compromise program correctness. As described in Section 2.1, the Intel compiler currently generates code that will silently switch to *serial irrevocable* mode when it encounters such pointers. To mimic the behavior of the original STAMP application, we can use Intel's `__transaction [[waiver]]` extension, which allows us to call through these pointers nontransactionally. Alternatively, we can declare the target functions as `transaction_safe` and call them transactionally, without the waiver. This leads to significant overhead, however, because the functions are called frequently during core data structure traversals, and the compiler must now use instrumented versions of the code. For completeness we test both "with waiver" and "without waiver" versions of Vacation.

The RMS-TM suite comprises seven applications from the Recognition, Mining and Synthesis (RMS) domain. As in STAMP, transactions vary greatly in length, read- and write-set size, and degree of contention. RMS-TM applications also exercise a variety of special TM features, including nested transactions, I/O, and system calls and complex function calls inside transactions. Unlike STAMP, the RMS-TM suite was developed using the C++ TM

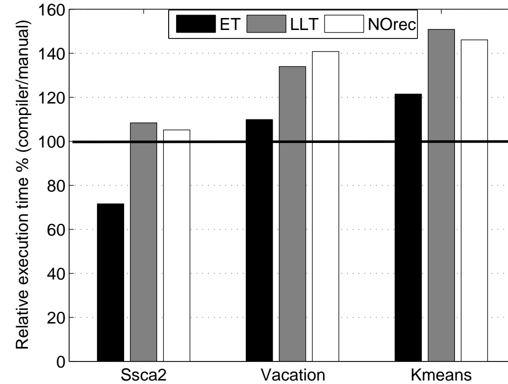


Figure 2: Execution time of compiler-instrumented code, relative to manually instrumented code, for single-threaded STAMP applications. Vacation represents "without waiver" execution.

standard rather than a library-level API. Running these applications directly on the RSTM back ends, without the shim library, would have required large amounts of tedious and error-prone hand instrumentation. We report results for one application from each of the RMS-TM application domains: HMMcalibrate (from Bioinformation), UtilityMine (from Datamining) and FluidAnimate (from Physics).

We perform our experiments on a 2.27 GHz, 2-processor Intel Xeon (E5520) system. Each processor contains four hyperthreaded cores serviced by private 32KB L1 Icache and 32KB Dcache, a private 256KB L2 cache, and a shared 8MB L3 cache. The system is equipped with 8GB of RAM that each processor access through a QPI memory controller. Benchmarks are written using the subset of the C++ TM draft API [1] supported by the Intel® C++ STM Compiler Prototype Edition 4.0 [12], and compiled using –O3 settings. The reference input sets were used where applicable. Experiments were performed on Linux version 2.6.30. We rely on the default Linux thread scheduler which prefers to distribute threads across processors before cores before hyperthreads. The tested benchmarks and implementations do not benefit from hyperthreading, so we report results up to 8 threads only.

4. Experimental Results

4.1 Overhead Analysis of Automatic Instrumentation

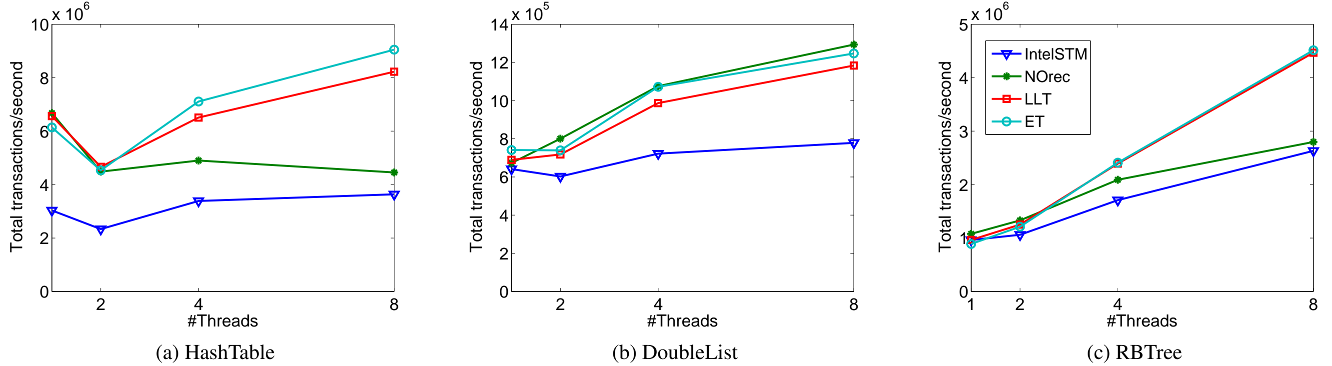
While relying on a compiler to automatically instrument read and write accesses simplifies the instrumentation of complex programs relative to manual instrumentation, it may lead to over-instrumentation due to the need for conservative assumptions about aliasing and lack of idempotence. On the other hand, the compiler may identify optimization opportunities that were missed during manual instrumentation, therefore improving performance. To assess these potential effects, we compared the performance of the original, manually-instrumented STAMP applications to that of the automatically-instrumented versions that use our shim library. We could not perform the same analysis for RMS-TM, as manually instrumented versions are not available.

Linking with RSTM through the TM ABI shim library introduces some additional overhead, unrelated to the compiler, relative to manually instrumentation. We expect this overhead to be small—at most one additional function call per instrumented access.

As noted in Section 3, the manually instrumented versions of the `List` and `RBTree` data structures in Vacation use uninstru-

Application	IntelSTM			NOrec			ET			LLT		
	2	4	8	2	4	8	2	4	8	2	4	8
HashTable	0.05	0.17	0.28	0.02	0.07	0.24	0.85	11.91	5.02	1.68	5.53	11.11
RBTree	0.00	0.00	0.01	0.00	0.00	0.00	0.02	0.01	0.09	0.18	0.48	1.57
DoubleList	13.85	36.31	52.13	10.09	27.56	49.48	7.75	29.15	57.35	14.81	37.38	63.16
SSCA2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.01	3.00
Kmeans	0.05	0.02	0.00	2.23	5.59	13.48	47.39	56.82	74.95	37.15	55.61	76.24
Vacation	0.01	0.04	0.08	0.00	0.00	0.00	0.80	1.13	4.26	0.08	0.26	0.66
HMMcalibrate	15.24	39.36	66.76	4.52	14.99	43.55	98.16	99.54	99.94	91.01	97.29	99.05
UtilityMine	0.01	0.05	0.26	0.00	0.03	0.10	0.09	1.44	0.80	0.11	0.41	0.93
FluidAnimate	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.02	0.04	0.02	0.05

Table 1: Abort Rates (percentage of all dynamic transaction instances that abort) for 2, 4 and 8 threads.



(a) HashTable

(b) DoubleList

(c) RBTree

Figure 3: Throughput results for the microbenchmarks. Y axis shows total number of transactions per second: higher is better.

mented functions internally for frequently executed comparison operations. The compiler cannot possibly generate equally efficient code for these without a global understanding of the program, as the comparisons access shared memory locations. In the next section we provide Vacation results both with and without `__transaction [[waiver]]`. The former requires the same level of programmer understanding as the original implementation; the latter illustrates the overhead of leaving code generation entirely up to the compiler.

Figure 2 shows the overhead of automatic instrumentation for the single-threaded execution of the STAMP benchmarks (without `__transaction [[waiver]]`) on the three RSTM back ends. The results depend on both the applications and the back end. SSCA2 shows performance improvement for ET and limited overhead for LLT and NOrec. Since ET shows a net benefit, we believe that the compiler does a good job of instrumenting the code and identifying optimization opportunities, and that the different behavior of LLT and NOrec is specific to the STMs. For Kmeans and Vacation, on the other hand, all of the back ends suffer significant performance loss compared to the manually-instrumented version—from 10–50%. Here the compiler clearly introduces read/write instrumentation that the manually instrumented version was able to avoid, and extra optimization opportunities, if any, are insufficient to compensate.

Conservative instrumentation can have an effect on scalability as well. The resulting larger read and write sets lead to longer transactions, due to increased validation times and to an increase in the probability of false conflicts in orec-based implementations. In Kmeans, for example, manually instrumented code sees 8-thread abort rates for ET and LLT of 3% and 58%, respectively. For compiler instrumented code (Table 1), the corresponding rates are both around 75%. Clearly the extra instrumentation inserted by the compiler in this case interacts badly with eager conflict detection.

NOrec, which is lazy like LLT, sees an abort rate of approximately 14% with both manual and automatic instrumentation. It would be vulnerable, however, to increases in the number of instrumented writes, since its write-back operations are globally serialized. An even larger issue would arise in any application where compiler the instrumented writes in what could otherwise be a read-only transaction.

4.2 Back-end Comparisons

In this section we present performance results for the three sets of benchmarks mentioned in Section 3.

Microbenchmarks: In our first experiment we consider microbenchmarks in which a set of threads use transactions to continually insert, delete, and look up keys in a set. The set is prepopulated with half of the possible keys and we execute an instruction mix that consists of 33% of each operation. Approximately half of the insert and delete operations find the target key and modify the set, so transactions should be 66% read only. The IntelSTM compiler does not introduce any unnecessary writes, so the results presented here meet this goal. We consider three different set implementations—a hash table, a red-black tree, and a doubly linked list. Figure 3 reports the throughput (total number of transactions per second) for these microbenchmarks when varying the number of threads from one to eight.

In HashTable (Figure 3a) we test 8-bit keys (maximum set size of 256), and transactions are tiny, performing a maximum of five reads and three writes. This results in few conflicts, as seen in the low abort rates for all the back ends (between 0.24% and 11.11% with eight threads, as reported in Table 1). This configuration should be extremely scalable, however we immediately see the effect of Linux's default scheduling policy. Placing the second thread across the QPI interconnect results in long latencies and high overheads for data and metadata access once we have two threads.

ET, LLT, and the IntelSTM can overcome this initial drop given enough threads, but NOrec's reliance on a single global sequence lock will not scale across the processors with such small transactions. Further investigation shows a high number of commit time re-validations for HashTable compared to the other microbenchmarks (23% of all commits with eight threads), which implies that NOrec transactions spend much of their time waiting in their commit barrier due to their need to validate after each writer commit. ET and LLT show better scalability at eight threads than IntelSTM; this may be attributed, at least in part, to the overhead of privatization safety in IntelSTM (not needed in the microbenchmark, and not provided by default in ET or LLT).

In DoubleList (Figure 3b) we again test 8-bit keys, but experience much more contention due to the linear structure of the list-based set. As with HashTable, DoubleList transactions perform a small number of writes, however they may perform up to 300 reads. These longer transactions reduce the relative overhead of metadata bottlenecks, resulting in better scalability for the RSTM back ends. The large number of conflicts means that, in contrast to HashTable, ET and LLT validate nearly as frequently as NOrec. NOrec's higher throughput is a result of its lower abort rate, which stems in turn from value-based conflict detection and the resulting lack of false conflicts. It is currently unclear why larger transactions do not benefit IntelSTM as well. We suspect that contention management may play a role.

Finally, RBTree (Figure 3c) expands the key set size to 20 bits and illustrates the behavior of memory-bound applications. With set sizes approaching a million elements, RBTree transactions may perform over 100 instrumented reads and up to 50 writes during rebalancing. Data cache misses dominate execution time, with ET and LLT's weak privatization guarantees affording them better scalability than IntelSTM and NOrec. As with HashTable, NOrec's scalability is impacted by its need to validate when any writer commits.

STAMP: Figures 4a, , 4b, and 4c show performance results for the selected STAMP applications on the tested back ends.

Figure 4a shows Vacation results using the recommended "high" contention parameters, both with (dotted lines) and without (solid lines) `__transaction [[waiver]]`. With the waiver, Vacation exhibits large, read-dominated transactions—more than 1300 instrumented reads and 150 instrumented writes—with low contention, evidenced by low abort rates in Table 1. As expected, all back ends provide good scalability with performance improvement up to eight threads. Without the waiver, the number of instrumented reads roughly doubles, to more than 2500. IntelSTM continues to scale well in these conditions. The RSTM back ends, however, have a clear performance problem with read sets this large. As of this writing, the source of the problem is unclear, and is a subject of ongoing investigation. We would not have been aware of the issue without the availability of the Intel ABI to RSTM shim.

SSCA2 transactions (Figure 4b) consist of up to three reads and two writes, and are effectively independent of one another. Each transaction performs at least one write, and transactions form the bulk of application execution time. This represents the pathological workload for NOrec, where writer commits are serialized. We see this in NOrec's lack of scalability. In contrast, ET, LLT, and IntelSTM allow non-conflicting writers to commit in parallel and scale well. IntelSTM shows high overheads similar to those seen in the HashTable microbenchmark, where transactions are similarly small and nonconflicting. We speculate that the cause of this overhead may be related to mechanisms used to provide privatization safety [20].

As discussed in Section 4.1, the Intel STM compiler appears to dramatically over-instrument Kmeans transactions. This results in larger read and write sets and, consequently, higher abort rates than those reported by Minh et al. [19]. For ET and LLT, the abort rates are particularly high: 75% or more at eight threads (Figure 1). The fact that NOrec sees only a 13% abort rate at eight threads suggests that most of the problem in ET and LLT is due to false conflicts. At the same time, compiler instrumentation results in all transactions being writers, which penalizes NOrec disproportionally, giving it the longest 8-thread execution time. Notice that, while the abort rate is very low with IntelSTM, its performance is similar to that of the other STMs. This suggests that its performance is dominated by other components.

RMS-TM: Figures 4d, 4e, and 4f show the execution time of the selected RMS-TM applications. HMMcalibrate exhibits short transactions with high contention. As shown in Table 1, it has the highest abort rate (between 44% and 99.9%, with eight threads). At the same time, it spends only a tiny fraction of its execution time inside transactions, allowing it to exhibit good scalability for all the back ends.

In UtilityMine (Figure 4e), IntelSTM shows high run-time overhead even with two threads. ET and LLT keep improving up to eight threads, but NOrec does not: a large number of threads increases the number of re-validations, leading to very little improvement beyond four threads. IntelSTM scales similarly to ET and LLT beyond two threads, but overall performance is dominated by the high instrumentation overhead.

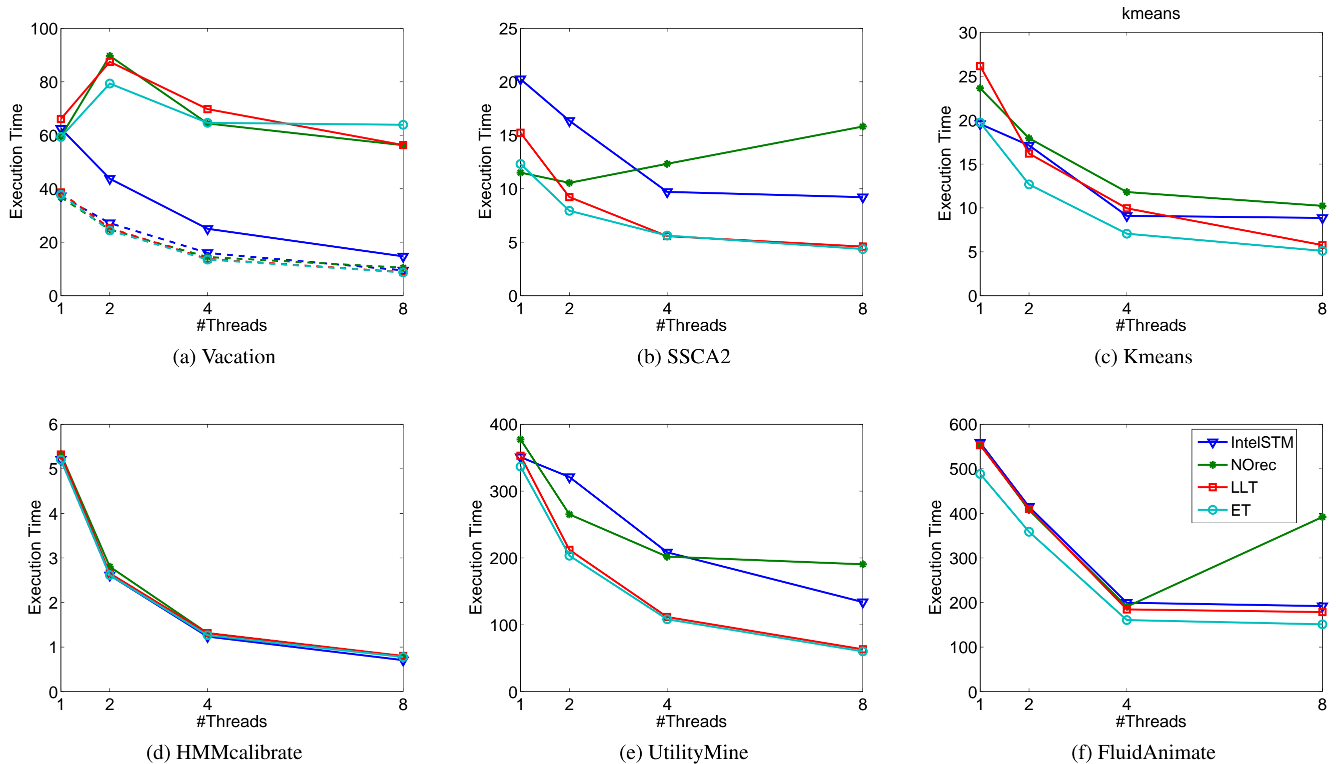
FluidAnimate also has short transactions, but in contrast to HMMcalibrate, its contention is low. When increasing the number of concurrent threads, the number of transactions per thread remains constant, so the total number of transactions increases. On the other hand, the work done per thread decreases with the number of threads: as a result, FluidAnimate shows strong scalability up to four threads. With eight threads, however, the ratio between the computation and synchronization phases decreases, which limits scalability (Figure 4f). The high frequency of writer transactions (read/write ratio of 1.16:1) leads to a performance bottleneck in NOrec at eight threads.

Summarizing, our results show that scalability and overall performance depend heavily on both the application and the choice of back-end system. Generally speaking, high instrumentation overhead limits overall performance. IntelSTM shows significantly higher overhead than the RSTM back ends for some applications (e.g., HashTable and SSCA2). For these, even single-thread performance is significantly lower than with the other STMs. If the abort rate is high, value-based conflict detection (NOrec) helps reduce false conflicts and, therefore, improves performance. DoubleList and HMMcalibrate illustrate this effect. When the read/write ratio is low (i.e., the application has multiple active writer transactions), STMs that allow concurrent writers (ET, LLT and IntelSTM) show higher performance compared to single-writer STMs. We can see this effect strongly in SSCA2, and to a lesser extent in HashTable and Kmeans.

5. Conclusion and Future Work

As Transactional Memory moves towards a more robust and mature stage, it becomes essential to be able to share and run applications, compilers, and run-time systems among groups. Standardization is a key step in this direction. However, while releases of compilers with support for TM are available, much of the work that has been done on STM runtimes is not compatible with those compilers, because of interface issues.

In this paper we described work that makes back ends from the RSTM suite (specifically, LLT, ET and NOrec) compatible with the Intel TM ABI, and with compilers that conform to that ABI. This work entailed modest changes to RSTM itself, plus the creation of a shim library that adapts the Intel ABI to the RSTM API. Using the newly available back ends, we evaluated the performance of several



(a) Vacation (b) SSCA2 (c) Kmeans
(d) HMMcalibrate (e) UtilityMine (f) FluidAnimate

Figure 4: Scalability results for STAMP and RMS-TM. The Y axis shows execution time in seconds: lower is better. In Figure 4a, dotted lines represent the version with `__transaction [[waiver]]`

applications from the STAMP and RMS-TM benchmark suites; the former required manual re-writing to eliminate the manual instrumentation of the STAMP API and to accommodate the need for annotations on functions called within transactions in the C++ TM API.

Our work makes it possible, for the first time, to run large applications from other groups on the RSTM back ends, and to obtain an "apples to apples" comparison of back ends using such applications. It also allows us, in the case of STAMP, to compare automatically and manually instrumented applications.

We find that memory footprint, abort rate, and consequent performance depend heavily on both the particular application and the choice of back-end system. This result confirms earlier findings with microbenchmarks; from it we conclude that diversity in back ends is essential, and that dynamic adaptation among back ends (as explored, for example, by Spear [25]) is a promising research direction. Our experiments also show that, while unnecessary instrumentation introduced by the compiler may induce considerable run time overhead, the Intel STM compiler is able to exploit optimization opportunities that may actually improve performance over hand-instrumented code in certain cases.

We plan to extend the set of RSTM back ends compatible with the Intel ABI to include RingSTM [26] and TML [3], and to complete the porting of the missing STAMP applications. We also plan to compare the Intel STM compiler to the GNU (GCC-TM) and Dresden (DTMC) compilers.

Acknowledgments

Our thanks to the anonymous reviewers and to Tim Harris and Roberto Gioiosa for their helpful comments. Much of both the original and ongoing development of RSTM was performed by Mike Spear of Lehigh University.

References

[1] A.-R. Adl-Tabatabai and T. Shpeisman. Draft specification of transaction language constructs for C++, Aug. 2009. Version 1.0, IBM, Intel, and Sun Microsystems.

[2] L. Dalessandro, V. J. Marathe, M. F. Spear, and M. L. Scott. Capabilities and limitations of library-based software transactional memory in C++. In *Proc. of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

[3] L. Dalessandro, D. Dice, M. L. Scott, N. Shavit, and M. F. Spear. Transactional mutex locks. In *Proc. of the European Conf. on Parallel Processing*, Ischia-Naples, Italy, Aug.–Sept. 2010.

[4] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Jan. 2010.

[5] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory. In *Proc. of the 16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, USA, Mar. 2011.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. of the 20th Intl. Symp. on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[7] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proc. of the 14th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, Washington, DC, USA, 2009.

[8] U. Drepper. Parallel programming with transactional memory. *Queue*, 6:38–45, Sept. 2008.

[9] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Salt Lake City, UT, USA, Feb. 2008.

[10] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2nd edition, 2010.

[11] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of the 20th Annual Intl. Symp. on Computer Architecture*, San Diego, CA, USA, May 1993.

[12] Intel C++ STM Compiler, Prototype Edition. `software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/`.

[13] Intel Corporation. Intel transactional memory compiler and runtime application binary interface, May 2009. Document No. 318523-002US, revision 1.1.

[14] G. Kestor, V. Karakostas, O. S. Unsal, A. Cristal, and M. Valero. RMS-TM: A comprehensive benchmark suite for transactional memory systems. In *Proc. of the Intl. Conf. on Performance Engineering*, Mar. 2011.

[15] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proc. of the 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 209–220, New York, NY, USA, Mar. 2006.

[16] C. Lameter. Effective synchronization on Linux/NUMA systems. In *Proc. of the Gelato Federation Meeting*, San Jose, CA, USA, May 2005.

[17] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *Proc. of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Raleith, NC, USA, Feb. 2009.

[18] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable techniques for transparent privatization in software transactional memory. In *Proc. of the Intl. Conf. on Parallel Processing*, Portland, OR, USA, Sept. 2008.

[19] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proc. of the IEEE Intl. Symp. on Workload Characterization*, Seattle, WA, USA, Sept. 2008.

[20] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for C/C++. In *Proc. of the 23rd ACM SIGPLAN Conf. on Object-oriented Programming Systems, Languages and Applications*, pages 195–212, Nashville, TN, USA, Oct. 2008.

[21] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In *Proc. of the 16th Intl. Conf. on Parallel Architecture and Compilation Techniques*, Brasov, Romainia, Sept. 2007.

[22] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *19th ACM Symp. on Parallelism in Algorithms and Architectures*, San Diego, CA, USA, June 2007.

[23] Rochester Software Transactional Memory Runtime. Project web site. `www.cs.rochester.edu/research/synchronization/rstm/`.

[24] A. Shriraman, S. Dwarkadas, and M. L. Scott. Implementation tradeoffs in the design of flexible transactional memory support. *J. Parallel Distrib. Comput.*, 70:1068–1084, Oct. 2010.

[25] M. F. Spear. Lightweight, robust adaptivity for software transactional memory. In *Proc. of the 22nd Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[26] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: Scalable transactions with a single atomic instruction. In *Proc. of the 20th Symp. on Parallelism in Algorithms and Architectures*, pages 275–284, June 2008.

[27] M. F. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott. Implementing and exploiting inevitability in software transactional memory. In *Proc. of the Intl. Conf. on Parallel Processing*, Portland, OR, USA, Sept. 2008.