RESEARCH-ARTICLE

A study of transactional memory vs. locks in practice

VICTOR PANKRATIUS, Karlsruhe Institute of Technology, Karlsruhe, Baden-Wurttemberg, Germany

ALI REZA ADL-TABATABAI, Intel Corporation, Santa Clara, CA, United States

Published: 04 June 2011

SPAA '11: 23rd ACM Symposium on Parallelism in Algorithms and Architectures
June 4 - 6, 2011
California, San Jose, USA

Conference Sponsors:
SIGARCH
SIGACT

SPAA '11: Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures (June 2011)
https://doi.org/10.1145/1989493.1989500
ISBN: 9781450307437

A Study of Transactional Memory vs. Locks in Practice

Victor Pankratius
Karlsruhe Institute of Technology
76131 Karlsruhe, Germany
www.victorpankratius.com
victor.pankratius@kit.edu

Ali-Reza Adl-Tabatabai
Programming Systems Lab
Intel Corporation
Santa Clara, California
ali-reza.adl-tabatabai@intel.com

ABSTRACT

Transactional Memory (TM) promises to simplify parallel programming by replacing locks with atomic transactions. Despite much recent progress in TM research, there is very little experience using TM to develop realistic parallel programs from scratch. In this paper, we present the results of a detailed case study comparing teams of programmers developing a parallel program from scratch using transactional memory and locks. We analyze and quantify in a realistic environment the development time, programming progress, code metrics, programming patterns, and ease of code understanding for six teams who each wrote a parallel desktop search engine over a fifteen week period. Three randomly chosen teams used Intel's Software Transactional Memory compiler and Pthreads, while the other teams used just Pthreads. Our analysis is exploratory: Given the same requirements, how far did each team get? The TM teams were among the first to have a prototype parallel search engine. Compared to the locks teams, the TM teams spent less than half the time debugging segmentation faults, but had more problems tuning performance and implementing queries. Code inspections with industry experts revealed that TM code was easier to understand than locks code, because the locks teams used many locks (up to thousands) to improve performance. Learning from each team's individual success and failure story, this paper provides valuable lessons for improving TM.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming — Parallel programming. **General Terms:** Human Factors.

1. INTRODUCTION

Multicore is a challenge for software engineering, and we need mainstream languages that support productive and robust parallel programming in the large. In response to the problems of parallel programming with locks, Transactional Memory (TM) has been proposed as an alternative synchronization mechanism. Several new parallel programming languages such X10, Fortress, Chapel, and Clojure, all provide transactions in-lieu of locks as the primary concurrency control mechanism. Other research systems have extended existing languages such as C++ [21], Java [4], Haskell [12], and ML [23] with support for Transactional Memory.

Despite the recent advances in TM research, there is little experience using TM to develop more realistic parallel programs from scratch. Recent discussions [24] of TM versus locks focused on small, mostly numerical programs or micro-benchmarks to evaluate the worst case performance, but none of them took into account more complex applications and software engineering aspects such as the productivity of programmers over a longer period of time; the time needed for design, implementation, testing and debugging; the ease of code understanding; or problems with the usage of parallel language constructs. Other work studying the conversion of locks programs to TM failed to shed light on the issues encountered when parallelizing with TM from scratch [34].

This paper addresses a novel research question in an exploratory case study: How exactly do individuals program in parallel with locks and TM, given the same programming problem *specification*? Using diverse, real cases, we aim to analyze in-depth how the use of TM or lock-based constructs influence parallel programming and the resulting program. This approach differs from previous work providing a locks-versus-TM performance comparison on the same exact program. Moreover, we focus on learning from individual approaches rather than on testing hypotheses on statistically aggregated data.

We study graduate-level student programmers tasked with developing a parallel desktop search engine from scratch under realistic time pressures. The study was organized as a one-semester graduate-level computer science course. All subjects received the same four-week training at the start of the semester. The programming part of the project spanned ten weeks starting after the training. The study randomly assigned twelve graduate students to six teams. Three of the teams (teams L1, L2, and L3) had to use locks, while the other three teams (TM1, TM2, and TM3) had to use TM language constructs provided by the Intel C++ Software Transactional Memory (STM) compiler [21] – one of the most advanced STM compilers built on top of Intel's production C++ compiler.

The case study shows that TM was indeed applicable to a more complex, non-numerical program, and that a combination of TM with locks is useful and came out of necessity in practice. Locks teams spent more time on debugging due

SPAA'11, June 4–6, 2011, San Jose, California, USA.

to segmentation faults than TM teams. TM teams, however, spent more time on performance-related issues than locks teams. The parallel programs of TM teams were easier to understand, according to code inspections done jointly with industry compiler experts. Locks teams tended to have more complex parallel programs by employing up to *thousands* of locks to achieve scalability. The paper also shows that TM does not solve all concurrency control problems, and thus is not a silver bullet. In particular, both the locks and TM teams had data races because they used incorrect double-checked locking patterns [5].

The paper is a summarized version of [22] and makes the following contributions: (1) it is the first study to document how several teams wrote a realistic application from scratch using TM and locks over an extended period of time; (2) it provides insights by analyzing a combination of quantitative and qualitative data on performance, hours spent on various development phases, code metrics, ease of code understanding, and subjective psychological issues; (3) it shows that TM is indeed a valuable approach for parallel programming, although with an immature tool chain; (4) it provides evidence that it was beneficial to use TM and locks in combination, thus leveraging the advantages of both programming models; and (5) it collects evidence falsifying opinions that TM does not help building real-world parallel applications.

Empirical studies with human subjects are vital for assessing the true value of parallel programming proposals in practice and exposing problems and new directions to the research community. Unfortunately, case studies like this one are rare because they are costly, risky, and require a long time to conduct.

The paper is organized as follows. Section 2 presents the project requirements, the STM compiler, and collected evidence. Section 3 presents code metrics focusing on productivity and the use of parallelism constructs. Section 4 discusses the results of code inspections for all the programs. Section 5 breaks down the effort each team spent on parallelization, tuning, and debugging. Section 6 measures the performance of each team's search engine. Section 7 summarizes key results from our study. Section 8 discusses potential threats to validity. Section 9 contrasts related work. Section 10 provides our conclusion.

2. CASE STUDY DESIGN

Every team developed a desktop search engine based on the following requirements:

Indexing. The search engine works on text files only. It starts crawling in a pre-defined directory and recursively in all subdirectories. The index does not have to persist on disk. Different strategies for index creation may be employed (e.g., division into several sub-indices). All non-alphanumeric characters are treated as word separators. Case and hyphens between words are ignored. A progress indicator for indexing must show bytes and files processed so far, words found so far, and the number of words in the index. The number of indexing threads must be configurable via a command line parameter.

Search. The search must allow different types of queries: (1) queries for coherent text passages (e.g., "this is a test"); (2) queries with wildcard at the beginning or the end of a word (e.g.,"hou*" or "*pa"); (3) queries containing several words representing *AND* concatenation (e.g., "tree house garden"); (4) queries with word exclusion (e.g., "-fruit"). Queries must be allowed to execute while indexing is in progress, but it is not required to run more than one query at a time in parallel. It was up to the teams to decide whether to parallelize each query; the number of query threads was not required to be configurable from the command line, but the teams had to provide a reasonable default for the benchmarking. We assume that the files to be indexed do not change while the desktop search engine executes. In addition, no files are deleted and no new files are added. Features that are *not* required are an "OR" operator in queries, stemming or word similarity search, and regular expressions.

Output. Files for which the query is true must be output in a sorted order according to a primary and secondary criterion: (1) the sum of occurrences of all query words, needed if several criteria exist, such as in AND queries; and (2) alphabetically by file name. The default output of a query consists of the first 50 paths and files sorted as mentioned before, the total number of files matching a query, and the query time.

Scenario. To simulate a real-world industry scenario, we allowed the teams to use any data structures that they wanted. To be even more realistic, we allowed them to reuse any library or open-source code from the Web. This diversity was intentional because it helped find out which approaches worked and which did not work. All students had Bachelor-equivalent degrees in computer science and were pursuing Master's degrees in computer science. Students with inappropriate skills were not accepted in the project, and all accepted students received the same training on TM and locks prior to project start. All teams except team TM3 had one member with course experience on parallelism. More details on the experience profiles, which differ for each team, are shown in [22].

Before the study started, we gave the same parallel programming training to all students covering common parallel programming issues such as race conditions and deadlocks. We also conducted a feasibility study to make sure that the timeframe set for all teams is sufficient to finish the project and implement all features.

Compiler. The TM teams were required to use Intel's STM compiler, an industrial-strength C/C++ compiler that has been extended with a prototype implementation of transactional language constructs for C++ [21]. We used the first version of this compiler when it was available for download from Intel's website. We decided not use a library based approach [6] because we felt a language-based approach was more productive and less error prone. The Intel STM compiler provides new statements that allow the programmer to define and abort transactions easily. The `__tm_atomic` keyword defines an atomic block of statements that will execute as a transaction. Atomic blocks can be nested, and the effects of inner transactions are visible only when the outer transaction commits. The `__tm_abort` statement rolls back a transaction and reverses the effects to the state that existed on the entry to the innermost transaction enclosing the abort statement. The compiler extensions also include annotations to functions and classes to mark functions that will be called inside transactions. The `__tm_callable` annotation marks functions that can be called inside transactions and instructs the compiler to generate a transactional clone with the necessary instrumentation on shared memory accesses. The instrumentation calls into the STM run-time, which tracks conflicts between transactions. On detecting a

conflict, the run-time rolls back the effects of a transaction and retries it. The `__tm_pure` annotation marks functions that the compiler does not need to instrument; the programmer must make sure that such functions can be safely called inside transactions without instrumentation.

The TM teams were allowed to use Pthreads in combination with the TM extensions so that they could create and manage threads. It is technically possible to use locks, semaphores, and condition variables in combination with transactions, and subjects were allowed to do so.

Throughout this study, we followed the recommendations of [25, 32, 33] and used several sources of qualitative and quantitative evidence: (1) The teams used diaries to take notes, track progress, explain ideas and successful or unsuccessful approaches, document technical or non-technical problems, and capture events that had an impact on the work. (2) A time report sheet capturing effort on a daily basis, split according to predefined task categories. Section 5 analyzes these times reports. (3) Notes from weekly (semi-structured) interviews [25] asking open-ended questions about current status, problems, and plans without requiring any particular format in the response. Tables 1 and 2 summarize the interviews starting on the fourth week of the project, which was the first week with clearly visible progress. (4) A post-project questionnaire, filled out individually by each student. The detailed questions and all answers are listed in [22]. (5) The source code produced by each team. (6) The subversion repository that all teams were required to use. (7) Personal observations of the instructor.

3. CODE METRICS

This section presents measurements of objective code metrics gathered from all programs.

Summary of insights. The results show that in this study, the locks-based programs were more complex parallel programs, because the locks programmers tended to use many locks; our code inspections in Section 4 reinforce with additional observations that locks programs were more complex. Although the locks and TM teams programmed parallel search engines with similar functionality, the TM teams used fewer critical sections that often had fewer lines of code than the critical sections of the locks teams.

The results also show that two of the TM teams combined TM with Pthreads synchronization primitives, and that the TM teams rarely used the more advanced TM language constructs – only one team used the `__tm_pure` keyword, and only one team used the `__tm_abort` keyword. Our code inspections, presented in Section 4, provide further insight into these results. Inspections revealed that the Pthreads synchronization primitives were used for I/O and producer-consumer co-ordination. Inspections further revealed that `__tm_pure` was used for printing debug messages and optimizing access to immutable data, and that `__tm_abort` was used incorrectly in a racy fashion to optimize performance instead of being used for failure atomicity.

Figure 1 shows the total lines of code (LOC) produced by all teams, excluding comments, blank lines, or code from foreign libraries. All teams produced about the same amount of code; on average, locks teams produced 2160 LOC, and TM teams produced 2228 LOC.

TM teams have a higher standard deviation of LOC compared to locks teams, which can be explained by the fact that team TM1 (the most inexperienced team) had incomplete code that did not work on the final benchmark. On the other hand, team TM3 had more code than any other team, because they decided to implement themselves many thread-safe helper functions due to lacking library support for TM programs.

3.1 Usage of parallel constructs

Locks and TM teams clearly differ in how many lines of code contain parallel constructs (Figure 1). Between 5% and 11% of the locks teams code had parallel constructs (179 LOC on average). By contrast, between 2% and 5% of TM teams code had parallel constructs (83 LOC on average).

All locks teams used condition variables, but none of the TM teams did. Two of the TM teams used Pthread constructs in addition to the constructs for thread creation or destruction: As will be discussed in Section 4, team TM2 used one lock to protect a large critical section containing I/O, and team TM3 used two semaphores for producer-consumer synchronization.

Synchronization constructs were rarely lexically nested, with at most one level of lexical nesting. Later code inspections revealed for all TM teams that the nesting of their nested transactions was not necessary.

The special-purpose TM constructs offered by Intel's compiler were used very differently. Team TM2 used the `__tm_callable` annotation in 2 lines of code, but team TM3 used it in 115 lines. Team TM2 were the only team that used the `__tm_pure` annotation; later code inspections show that one usage of `__tm_pure` was for a declaration of `printf` so that they could debug the program. This is evidence that we need better debugging tools for TM. Only team TM1 used the `__tm_abort` construct, but as later code inspections show, they did not use it as it was intended to be used for failure atomicity – most of the time they used it incorrectly to optimize performance and implemented a racy double-checking pattern [5].

3.2 Comparison of critical sections

The critical sections differ for locks teams and TM teams. Figure 2 shows details on how many critical sections each team had and the cumulative lines of code. We see, for example, that team L2 has 25 critical sections with a size less than or equal to 1 LOC, 36 critical sections with a size less than or equal to 4 LOC, and so on.

We statically approximated a lower bound on the length of critical sections by manually counting the LOC enclosed by lock/unlock operations, semaphore operations, or *atomic* blocks, and excluding comments and blank lines. Information from code inspections shows that some locks teams had arrays with thousands of locks, but these lock definitions showed up as just one line of code; we counted a function call within a critical section as one LOC and omitted dynamic analyses.

A key observation is that most critical sections are short. TM teams have fewer critical sections than locks teams, even though all teams implement similar functionality. The locks teams have many critical sections with just one line of code, which could have been easily expressed as atomic blocks.

4. CODE INSPECTIONS

In this section, we present observations from code inspections. The authors and leading industry compiler experts inspected each team's code in detail, but in an anonymized

	Locks Teams			TM Teams		
	L1	L2	L3	TM1	TM2	TM3
Total Lines of Code (excl. comments, blank lines)	**2014**	**2285**	**2182**	**1501**	**2131**	**3052**
	avg: 2160 stddev: 137			**avg: 2228 stddev: 780**		
LOC pthread*	157	261	120	17	23	12
	8%	11%	5%	1%	1%	0%
LOC tm_*	0	0	0	36	22	139
				2%	1%	5%
LOC with paral. constr. (pthread* + tm_*)	157	261	120	53	45	151
	8%	11%	5%	4%	2%	5%
	avg: 179 stddev: 73			**avg: 83 stddev: 59**		
Total effort in pers. hours	151	334	208	208	261	141
Productivity in person hours/LOC	0.07	0.15	0.10	0.14	0.12	0.05
	avg: 0.105 stddev: 0.037			**avg: 0.102 stddev: 0.049**		

	Locks Teams			TM Teams		
	L1	L2	L3	TM1	TM2	TM3
Selected details on Pthreads constructs						
LOC pthread_create*	8	13	8	6	3	3
LOC pthread_cond*	10	18	6	0	0	0
LOC sem_*	0	0	0	0	0	10
LOC pthread_mutex_t*	14	28	9	0	1	0
LOC pthread_mutex_lock*	43	45	24	0	1	0
LOC pthread_mutex_unlock*	43	49	34	0	2	0
Average LOC per critical section	7.1	3.1	9.2		85	4.5
- no. of crit. sect. with nested locks (levels)	1 (1)	0	1(1)			
Selected details on Transactional Memory constructs						
LOC tm_atomic (= #atomic blocks)				12	17	24
- average LOC per atomic section				5.9	3.5	6.4
- no of nested atomic sections (level)				0	2 (1)	1 (1)
LOC tm_callable				18	2	115
LOC tm_pure				0	3	0
LOC tm_abort				6	0	0

Figure 1: Code metrics for the parallel desktop search engines of all teams.

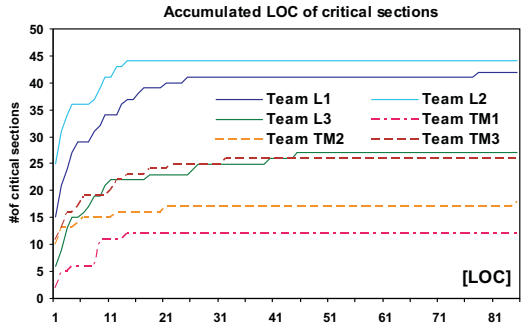

Figure 2: Code distribution in critical sections.

form. These inspections allow us to analyze the use of constructs, the kinds of parallel programming mistakes, and code and bug patterns. We present selected highlights from each team on architecture, major data structures, synchronization, ease of code understanding, and problems.

Summary of insights. The code inspections revealed several interesting usages of the parallelism primitives. First, the lock-based programs used many fine-grain locks to get scalability, and the use of fine-grained locks was difficult to validate by code inspection. Second, code inspections revealed why the TM teams combined Pthreads synchronization with atomic blocks, and how they used the more advanced TM language constructs: Realistic TM programs require producer-consumer synchronization, perform I/O, and need ways to optimize access to immutable data.

Despite being taught otherwise, our inspections revealed that all teams – both locks and TM teams – incorrectly assumed in several places that it is safe to read shared data without synchronization and had obvious data races due to racy double-checked locking patterns [5].

4.1 Code inspections for locks teams

In general, all locks teams parallelized the indexing using a crawler thread to generate work for a set of worker threads that created the index in parallel. The granularity of work differed between the teams: In team L2's program the crawler thread generated work at the granularity of files, while in team L1's and L3's programs the crawler thread parsed each file and generated work at the granularity of words. All teams could query at the same time as indexing, but team L3 did not parallelize the query itself. All teams had a shared index data structure that was updated in parallel by the indexing worker threads and concurrently read by a query thread.

The code inspections show that realistic programs may require many fine-grain locks in order to have scalable performance. All teams attempted fine-grain locking of the index data structure to allow concurrent access to disjoint parts of the index structure; to protect the index structure, team L2 used 1600 (!) locks, team L3 used 80 locks, and team L1 used 54 locks. Team L2's program, which had the largest number of locks, was the only locks program to scale on indexing. Locks are mostly used in a block-structured manner; however, team L2 and L3 have cases where unlocking is performed in both *then* or *else* statements due to a function return from the middle of a critical section (explaining why there are more unlock operations than lock operations in Figure 1). Many locks are created dynamically, so the total number of locks is larger than the count of lines of code count containing the `pthread_mutex_t` construct.

Some locks teams used the high number of locks to compensate for their insecurity when writing complex parallel programs. Team L2, for example, emulated the Java synchronized construct. They introduced a lock for every object knowing that they would sacrifice performance, yet they still had races. Most teams made the common mistake of believing that unprotected reading of shared state is safe (despite being taught otherwise), thus they had races. Only team L1 had critical sections protecting a single shared variable read.

Team L1 has index data structure consisting of an array of sub-indices for every character. Each sub-index consists of a map storing all words starting with the particular character of that sub-index. They use a lock to protect the work queue, several locks to protect code that displays information, and two arrays of locks and two additional locks protecting accesses to the two sub-index arrays. Each lock in these two arrays of locks protects a different character in one of the two sub-index arrays, so there are 54 (27 x 2) locks protecting the index structure. The number of dynamic lock instances is therefore greater than in Figure 1. To avoid deadlocks, the team specified an order for acquiring locks in these lock arrays, documented as comments in a header file. This protocol is not complete, however, because it misses some locks that are acquired in a nested fashion; in addition, the code violates the protocol in at least one

place. The team also used a copy-and-paste approach for many critical sections.

Team L2 used an inverted index data structure in which stored words are accessed using the first two characters. They don't speed up wildcard searches using a reverse index. The team assumes 40 possible characters and creates $40 \times 40 = 1600$ disjoint map structures, each of which maps a word to the document and position within the document. With this many maps, they hope to insert and access the index in parallel without causing much conflict. It is difficult to spot the high number of locks in the code of the indexing data structure:

```
//vocabulary.h:
 class Vocabulary {
 private:
   std::map<std::string, InvertedList> invertedLists;
   pthread_mutex_t access_mutex;
...
//bigvocabulary.cc
...
characters="abcdefghijklmnopqrstuvwxyzäöüSS0123456789"
// creates the index-structure
 for(int i=0; i < (int)characters.length();i++){
   std::map<std::string,Vocabulary>tmp_map;
   for(int j=0; j < (int)characters.length();j++){
     Vocabulary tmp_voc;
     tmp_voc.initialize();  ...
```

Later on, this nested loop creates 1600 vocabulary objects, each of which contains a lock and the map. Team L2's code has clear data races. The getter accessor functions on most classes don't use locks while updater functions do, so this team assumes that writing to a shared data structure must be protected by a lock, but reading does not.

Team L3 used a BurstTrie based on [13] for their index data structures, a more complex tree-based data structure compared to the maps used by teams L1 and L2. This team used an array of 40 locks at the root of the index data structure, and 40 at the root of the reverse index. The locks are acquired depending on the first letter of the word to be indexed. An insertion into the index requires acquiring two locks. This leads to the same scalability problems as for team L1, which is lots of contention for words with a frequent first letter. They also have racy code:

```
//called by each indexer thread
void BurstTrie::Insert(...) {
 ...
 if(rootNode == NULL){
 rootNode = new BurstNode(); //unprotected
 rootNodeReverse = new BurstNode(); //unprotected
 ...
}
```

4.2 Code inspections for TM teams

Like some of the locks teams, teams TM1 and TM3 implemented a crawler thread that produced a list of files to index into a shared work queue from which a pool of indexer threads grabbed work. Except for team TM1, none of the TM teams parallelized queries. Unlike all of the other teams, team TM2 used a persistent index on disk and ran queries in a separate program that read the on-disk index.

The code inspections show that realistic TM programs need to perform producer-consumer synchronization. Team TM3 used a semaphore. Team TM2 avoided producer-consumer synchronization because each indexing thread performed part of the crawling. Team TM1 did not consider producer-consumer synchronization because an indexer thread exits once it detects an empty work queue. The C++ TM model must therefore either be extended to handle these operations, or TM must be allowed to be combined with other lock-based primitives.

In addition, realistic TM programs need to do I/O and optimize access to immutable data inside transactions. Team TM2 used a global lock in a critical section that performed many I/O operations. They also used `__tm_pure` to optimize comparisons of immutable strings inside of a transaction. It was hard for the code reviewers to validate the correct usage of `__tm_pure`. A compiler-enforceable approach would clearly have been better.

Like the locks teams, TM teams incorrectly assumed that unprotected reading of shared state is safe. Most teams systematically tried to optimize transaction performance by first checking a condition outside a transactions and then checking it inside, similar to incorrect implementations of the double-checked locking pattern [5].

Team TM1 used a two-level index based on linked lists. On the first level there is an entry for each character a word can start with. For each of these entries, there is a list of characters a word can end with on the second level. Attached to each entry on the second level is a list of all words (with document positions) that start and end with a certain character. Team TM1's code has clear data races. The `__tm_atomic` construct mostly protects short code passages. The `__tm_abort` construct was used six times. In five times, they used it incorrectly to implement a racy double-checking pattern:

```
while (added == 1) {
 //check outside atomic
 if (dokulist->get_counter() < DOKU_NUM) {
 __tm_atomic { //check inside atomic
   if (dokulist->get_counter() >= DOKU_NUM) {
     __tm_abort; }
   else {dokulist->add_to_DokuNode(newDoku,newPosi);
     added = 0;}}}}
```

Team TM2, unlike all other teams, does not have a crawler thread. Instead, each indexer thread updates a shared directory stack that keeps track of the current directory to crawl. This is also the only team to store the index on disk. This team used a modified B-tree according to the approach described in [18]. They incorrectly tried to avoid transaction overhead in a double-checked locking style:

```
//bufferload.c
...
if (dl->length < DLCHUNK) { //check outside
__tm_atomic {
 if (dl->length < DLCHUNK)  { //check inside
  dl->entry[dl->length].docid = docid;
  dl->entry[dl->length].freq = 1;dl->length++;
  return 0;} }
```

Team TM2 also assumed that reading shared variables without protection is safe, thus introducing data races:

```
 //bufferload.c
 ...
 while (word = getWord(p)) {
    node = findBufferWord(&b, word);
    __tm_atomic {
      node = findBufferWord(node, word);  ...
```

Team TM3 used a crawler thread that goes breadth-first through the directories and produces a list of files to be indexed into a single work queue. A pool of indexer threads each opens the files, invokes a lexer to produce term-frequency pairs, and updates the shared index. For the index data structure, they use a vocabulary trie as in [7], which is a tree-like data structure with nodes representing shared prefixes of index terms. Two semaphores, `fillcount` and `emptycount` are used in the thread pool for producer-consumer synchronization. `__tm_atomic` mostly protects short code passages. They used several small transactions back-to-back instead of big transactions to optimize performance. Their indexer code has a race as it uses an incorrect variant of double-checked locking [5]. They check outside a transaction if their stack of files is empty, and then perform a *pop* operation inside a transaction. To work correctly, both operations should be inside the same transaction:

```
while(true){ //consumer
 sem_wait(&fillcount);
  if (new_files->is_empty()) {break;}
  __tm_atomic {filename = new_files->pop();}
 sem_post(&emptycount);  ...
```

5. PROGRAMMING EFFORT

Throughout the project, each team filled out a form tracking how many hours they spent per day on a certain task category. Figure 3 summarizes the overall results (the complete form is shown in [22]). TM teams spent less total effort than the locks teams. In particular, TM teams spent 28 hours less on reading documentation, 80 hours less on implementation, and 14 hours less on debugging than the locks teams. However, TM teams also implemented fewer query types, as shown in Figure 4.

Another source of evidence of programming effort is the interview results shown in Tables 1 and 2. We first summarize the results of the interviews and then analyze the efforts on parallelization, tuning, and debugging using both the data from Figure 3 and the interview data.

5.1 Interviews

Locks teams. In the eighth week, two weeks before the deadline, all locks teams had parallel implementations, but none of them could show a full demonstration. In the ninth week all teams had running search engines, but two of them appeared experimental: Team L1's program was unstable, and team L2's had segmentation faults. Team L3 focused on performance testing, but in the following week they found a bug. By the end of the project in the tenth week, team L1 had run out of time and skipped performance tests, team L2 was not finished with performance tests, and team L3 had discovered a concurrency bug that they were trying to fix before submission.

TM teams. In the eighth week, team TM1 had just a serial program, team TM2 had an incomplete parallel indexer and no queries working, while team TM3 had a full-fledged working demo. In the ninth week, team TM1 was still incomplete, team TM2 had a running, but buggy parallel program with bad performance, and team TM3 fixed many bugs in their search engine. By the end of the project in the tenth week, team TM1's program failed on the final benchmark, team TM2 had parallel indexing and queries working with reasonable performance, and team TM3 had an even more improved search engine.

Teams TM1 and TM2 procrastinated parallelization due to various reasons. Team TM1 lacked experience; both students were hesitant and insecure, especially during implementation. Team TM2 procrastinated parallelization because they wanted to have a more or less perfect sequential program as a basis on which to introduce transactions. Despite being the most experienced team in the study, they thought that query implementation would be trivial and underestimated its complexity. It appears that TM encourages the procrastination of parallelization in cases such as those of TM1 and TM2. All TM teams reported that it was difficult to find out where and how to apply atomic blocks and TM function annotations in a larger code base. In addition, TM performance was hard to predict. We need better tools to simplify these tasks.

Interestingly, even though teams TM2 and TM3 complain about difficulties in using TM constructs late into the project, the evidence shows that these teams merely had the impression that they could not make good progress and were hampered by TM constructs — the objective data shows that they did pretty well, even better compared to the locks teams.

5.2 Parallelization

TM teams spent in total about half as much time as the locks teams on writing parallel code (see [22] for details). TM allowed the experienced TM teams more time to think sequentially, which is backed up by (1) the hours spent on sequential code versus parallel code, and (2) the time lag between the first day of work on sequential code and the first day of work on parallel code. The locks teams had a shorter time lag between the first day of work on sequential code and the first day of work on parallel code: team L1, 1 day; team L2, 13 days; and team L3, 19 days. By contrast, TM teams have larger time lags: team TM3, 19 days; team TM2, 23 days; and team TM1, 29 days.

The effort data generally backs up several of our observations related to parallelization from the interviews. First, the larger time lags for teams TM1 and TM2 back up our observations that these teams procrastinated parallelization. Team TM3, who were also the first to have a working parallel version, started parallelization after locks teams L1 and L2. The locks teams L1 and L2 were the first to start parallelizing, whereas the teams TM1 and TM2 were the last to start parallelization.

5.3 Performance tuning

The collected effort data shows that TM teams had more problems with performance tuning than the locks teams. Late into the project, the TM teams had to experiment with performance and restructure their programs to deal with performance problems. Refactoring effort increased for all TM teams by the end of the project. Team TM2 mentioned during the interviews that they had to split up large transactions into smaller ones, pointing to a late restructuring problem for TM programs. In order to understand TM performance, all TM teams had sharp increases of effort by the end of the project to do performance experiments with smaller programs. All these results suggest that further research is needed into developing performance analysis tools and refactoring techniques for TM-based programs. In addition, research on programming patterns or anti-patterns for TM can help reduce performance problems.

Total Effort (Person Hours)

	Reading	Search for Libraries	Design	Implementation	Additional Experiments	Testing	Debugging	Other	Total
Team L1	6	3	9	80	10	14	29	0	151
Team L3	24	1	17	72	7	52	16	19	208
Team L2	29	12	14	196	12	21	48	2	334
Team TM3	18	4	12	55	6	18	19	9	141
Team TM1	7	6	33	74	18	38	22	10	208
Team TM2	6	6	21	139	12	39	38	0	261
sum all	**90**	**32**	**106**	**616**	**65**	**182**	**172**	**40**	**1303**
	7%	2%	8%	47%	5%	14%	13%	3%	100%
sum L	**59**	**16**	**40**	**348**	**29**	**87**	**93**	**21**	**693**
	9%	2%	6%	50%	4%	13%	13%	3%	100%
sum TM	**31**	**16**	**66**	**268**	**36**	**95**	**79**	**19**	**610**
	5%	3%	11%	44%	6%	16%	13%	3%	100%
sum L - sumTM	*28*	*0*	*-26*	*80*	*-7*	*-8*	*14*	*2*	83

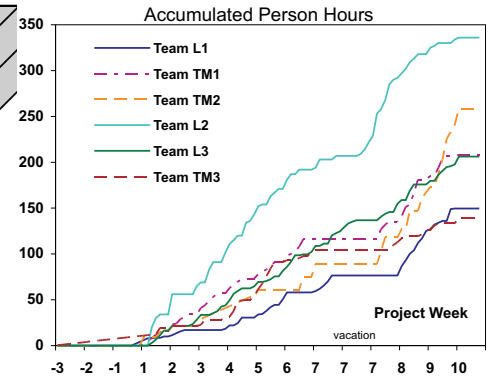

Figure 3: Total effort of all teams in person hours.

Prj. Week	
4	• **L1:** The team discussed the index design and the placement of locks, but did not have any code running yet. • **L2:** The team finished a sequential indexer and assessed its performance. They were the first team to elaborate thoughts on how threads might traverse the index in parallel. • **L3:** The team did not have a running program yet. The team discussed indexing strategies and data structures choices, but had no code running.
5	• **L1:** The team assessed two prototypes for parallel indexing in various experiments. First, they used one global mutex, which yielded bad performance. Then, they decided to go for several independently locked sub-indexes. • **L2:** The team implemented a rudimentary parallel indexer. • **L3:** The team had implemented a sequential prototype with an index structure, and they were testing the performance. They had a customized, small benchmark that was unrelated to the case study benchmark.
6	• **L1:** The team had a prototype of parallel indexing and parallel queries working, but the prototype had performance problems. The file crawler – a key component for indexing – was not implemented, but just simulated. • **L2:** Parallel indexing worked. Queries could be executed while indexing was in progress. • **L3:** Parallel indexing worked. Queries were implemented in a rudimentary way.
7	• **L1:** Parallel indexing and parallel queries still worked with the simulated file crawler. They were working on query result ranking but were not finished yet. • **L2:** The team has made little progress due to problems with C. • **L3:** The team showed how they used the Linux system monitor for performance testing and debugging. There was not much other progress to see.
8	• **L1:** The team had finished all components except the file crawler, but they hadn't tested it yet on the real benchmark. • **L2:** The team found out that they had problems compiling their code on other machines and were about to fix that. • **L3:** Parallel indexing and parallel search worked, but only on a subset of the case study benchmark.
9	• **L1:** The team finished implementing the file crawler. Parallel indexing and parallel queries worked, but were unstable. • **L2:** Parallel indexing parallel queries worked. The team fixed segmentation faults and did performance tests. • **L3:** Parallel indexing and queries worked. The team continued with performance testing.
10	• **L1:** The team still had not tested performance on the given benchmark. • **L2:** The team was about to test performance on the given benchmark. • **L3:** The team was about to fix a bug with the file statistics update of their indexer.

Table 1: Progress summary from locks teams interviews.

5.4 Debugging

The total time for debugging was higher for locks teams than for TM teams (93 hours vs. 79 hours, respectively). Debugging due to segmentation faults was the major debugging cause for all teams. In total, locks teams spent 55 hours (59%) of debugging time on segmentation faults, whereas TM teams spent 23 hours (29%) of debugging time [22]. The time for debugging unexpected program behavior, was comparable for locks and TM teams; locks teams spent 20 hours (22%) of debugging time, and TM teams spent 16 hours (20%) of debugging time.

The effort spent on debugging segmentation faults seems to be influenced by the number of lines of code containing parallel constructs. Teams L3 and TM2 spent the least effort (4 hours) on debugging segmentation faults [22]. According to Figure 1, team L3 had the lowest number of lines of code with parallel constructs among the locks teams (120 LOC; 5% of the code); similarly, team TM2 had the lowest number of lines of code with parallel constructs among the TM teams (45 LOC; 2% of the code). By contrast, team L2 spent most effort on debugging segmentation faults (35 hours) and had the most lines of code (261 LOC; 11% of code) with parallel constructs. In addition, team L2 had the most extensive usage of condition variables, and team L3 the least among the locks teams. If future empirical studies confirm these observations, then TM programs requiring fewer parallel constructs than comparable locks programs will have an advantage in the debugging phase, as there would be a reduced probability for mistakes.

In the questionnaire responses, the TM teams felt that they had many segmentation faults and unexplainable crashes compared to the locks teams. Objective data shows, however, that the TM teams spent less effort than the locks teams on fixing segmentation faults [22].

6. PERFORMANCE MEASUREMENTS

Figure 4 shows the indexing and query performance for each team's programs. It shows what was possible for the subjects to achieve in a realistic programming environment with freedom of design decisions, but given the same programming problem, the same limited amount of development time, and the same starting conditions for all teams.

Prj. Week	
4	• **TM1:** The team discussed design alternatives. • **TM2:** The team was about to implement their index data structure and planned to have an executable version in 1–2 weeks. • **TM3:** The team had a rudimentary indexer. They had problems understanding and applying TM constructs.
5	• **TM1:** The team was testing a first sequential indexer. They had not thought how to parallelize or how to use TM. • **TM2:** The team implemented sequential index reading and writing operations. No thoughts on parallelism. • **TM3:** This was the first team of all with a working parallel indexer. Performance tests are done on the final benchmark. Segmentation faults appeared due to missing thread-safe TM libraries; they start implementing low-level functions by themselves.
6	• **TM1:** The team's entire code was sequential and incomplete. They had no new thoughts on parallelism or TM; many of their ideas were not well-developed. They planned to parallelize their program the following week. They were worried about the performance of their sequential program and hoped that parallelism would make it faster. The memory consumption of their program began to grow. • **TM2:** The team's entire code was still sequential. Neither of them had thought of parallelism or transactions yet. • **TM3:** The team's parallel indexing worked. A rudimentary query could execute while indexing was in progress.
7	• **TM1:** The team had made some unsuccessful parallelization attempts. They tested their program with just one of the files from the benchmark. They had memory leaks they couldn't find. • **TM2:** The team evaluated the TM annotations for functions on the their index. Part of the sequential code for insertions had to be restructured. They developed a strategy to minimize transaction overhead. Search was not implemented yet; they assumed it was trivial, though in the end almost no query worked. • **TM3:** The team finished implementing their thread-safe library functions.
8	• **TM1:** The team had procrastinated much of the parallelization work; indexing was serial. The few parallelization attempts were superficial. The memory leak was still there. Only one word could be used in a query. • **TM2:** The team had not yet finished parallel indexing. No performance tests had yet been done. Queries did not work yet. • **TM3:** The team showed a full-fledged working demo of parallel indexing and search. They used compiler statistics (such as #TMaborts, #TMretries, etc.) for performance optimization.
9	• **TM1:** The team' parallel indexing and queries were almost finished. Queries allowed just the inclusion or exclusion of one word. • **TM2:** The team's indexing and queries worked in parallel, but were not error-free. The program performance was still bad. Too much of the code was enclosed by atomic blocks. They started a lot of non-trivial refactoring to shrink the size of atomic blocks. • **TM3:** The team fixed a segmentation fault and many bugs.
10	• **TM1:** The team's indexing did not work for the case study benchmark, due to the memory leak they did not fix. Turning on compiler optimizations caused segmentation faults, which was a bug in the compiler. • **TM2:** The team's parallel indexing and queries worked. Turning on compiler optimizations caused segmentation faults. The frustrated team said that TM did not really relieve them from their problems, but just shifted them to transactions. They had problems understanding the performance overhead of *__tm_atomic* blocks; they were more expensive than expected. • **TM3:** The team's search engine was complete. They used TM frequently, but the team said it was difficult and tedious to find the places where to employ the *__tm_callable* function annotation.

Table 2: Progress summary from TM teams interviews.

Performance measurements were done by instructors on a Dell eight core machine with a dual-socket Intel Quadcore E5320 QC processor, clocked at 1.86 GHz, with 8 GB of RAM, and running Ubuntu Linux 2.6. Each point represents the average of five measurements. Only results of correctly working features are shown. All teams were requested to provide a configurable command line parameter to specify the number of indexing threads, and only this parameter was varied when measuring performance. All source codes were compiled with Intel's C compiler, using STM extensions for the TM teams. All source codes were inspected to ensure that they measure execution time in the same way; `printf` statements within time measurement blocks were commented out. The input file set used for benchmarking consists of directories containing a diverse selection of ASCII text files (50,887 files, 742 MB MB in total). It includes the Calgary Text Compression Corpus (which is used to evaluate compression programs), one big text file, four larger files, and many small files [22].

Team TM3 had the best indexing performance of all teams, completing the benchmark in 178 seconds. Compared to the fastest locks team on indexing (team L2) that finished in 532 seconds, TM was three times faster. TM teams had the best execution times for half of the queries; they were 13%–95% lower than the fastest locks team. Despite differing program designs, the measurements of the submitted search engines represent counter-examples that contrast the literature overgeneralizing that Software Transactional Memory always performs poorly [9].

Except for team TM1 who had memory consumption problems not fixed until the deadline, all teams had executable parallel programs at the final deadline. No search engine was perfect, however, as all implementations had queries that were either too slow, missing, or executing incorrectly. The number of working queries was the only difference in feature completeness, which is rather minimal considering the complexity of the overall project. Out of 18 queries, locks teams implemented 18 (L1), 17 (L3), and 10 (L2), while TM teams implemented 14 (TM3), 4 (TM2), and 0 (TM1). The locks teams implemented more queries than the TM teams, though locks team's queries were slow in many cases (see Figure 4). Our observations suggest that locks teams implemented more queries by skipping thorough software engineering practices such as testing and debugging (i.e., they risked that their features might not work). We assume that the effort gap between locks and TM teams would be even wider than in Figure 3 if the locks teams had tested their programs in a fashion comparable to the TM teams.

7. KEY RESULTS

The case study shows in the given setting that TM was indeed applicable to a more complex, non-numerical program. The results also show that TM needs better mechanisms for coordination and better handling of I/O. Programmer-initiated aborts were almost never used, and when used, they were used mostly incorrectly. We provide evidence that a combination of TM with locks is needed in practice and discovered real use cases of how locks and TM need to be combined: Two of the TM teams used TM as well as locks in the same program. One team combined TM with semaphores for producer-consumer coordination, and another team combined TM with a lock to protect a critical section that performed many I/O operations. This is an important insight because TM and locks were used as complementary approaches, not as alternatives excluding each other. This is a new point of view that has not been fully explored in the literature so far. While TM implementations have used locks under the covers (e.g., the STM runtime used in the Intel STM compiler [21]), researchers have not fully explored programming models that provide both

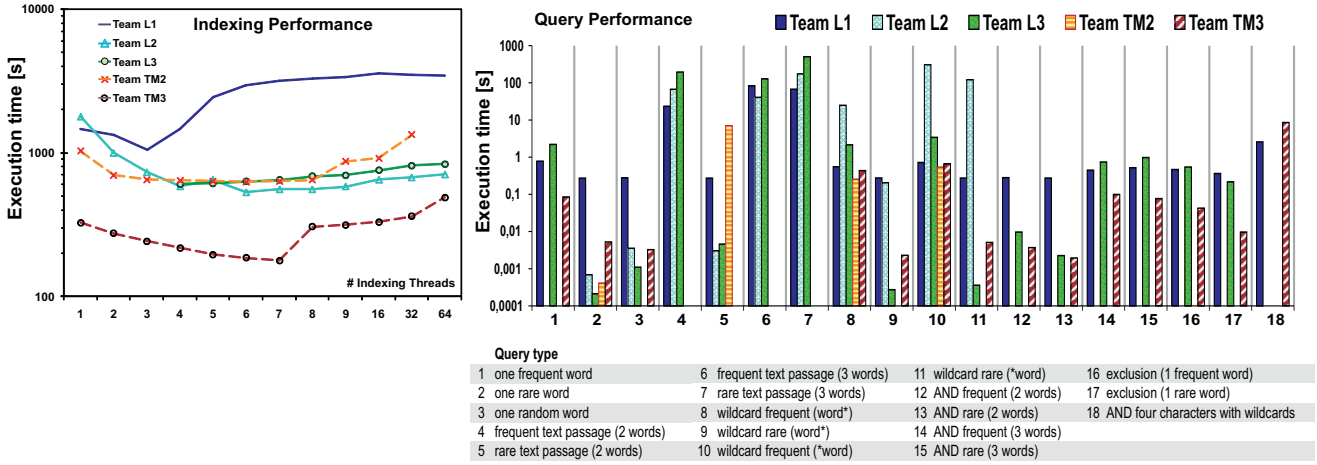

Figure 4: Indexing and query performance. The right graphs excludes for each team the queries that are not implemented or not executing correctly.

locks and atomic blocks with clear semantics. Intel's recent Draft Specification of Transactional Language Constructs for C++ [3] allows locks inside of atomic blocks. The authors of this specification were in large part influenced by the results of this study.

TM team's program performance was not bad compared to the locks teams. Team L1 was the only team to have implemented all queries, but they had the worst indexing performance and a slow search. Locks teams spent more time on debugging due to segmentation faults than TM teams. TM teams, however, spent more time on performance-related issues than locks teams, which also indicates that we need better TM performance tuning tools.

The parallel programs of TM teams were easier to understand, according to code inspections done jointly with industry compiler experts. Although all teams implemented similar program functionality, all TM teams used significantly fewer parallel constructs than the locks teams. Locks teams tended to have more complex parallel programs by employing many locks, sometimes *thousands* of locks due to the indexing data structure. All teams had races that were detected after the project by code inspection.

We detected differences in how teams perceived their progress by comparing subjective data from the questionnaire and interviews with objective data from the code and time report sheets. Team TM3, for example, thought that they were not advancing fast enough because they had to use transactions, but at the same time they had the first working parallel program and least effort of all teams. By contrast, locks teams subjectively believed they made good progress but actually needed more effort.

The study also shows that TM is not a silver bullet for parallel programming. The most inexperienced team using TM (TM1) did not produce a working program; parallel programming remains difficult.

8. THREATS TO VALIDITY

A case study provides detailed insights on one case being studied [25]. Our study describes observations and explores a variety of previously unknown issues when programmers with different experience use TM and locks in the realistic environment of a large project. It is easy to disprove general statements even with a small number of subjects, but difficult to prove general statements. By contrast, experiments would require a totally controlled environment (at the expense of realism) and a very narrow and previously known hypothesis to test. Even though the study focuses on just one type of application, it is possible that results differ for other applications; however, many of the encountered problems are representative and occur in other contexts as well. Such effects can be compensated by triangulating data from several sources. Internal validity is created by triangulating multiple sources of evidence and different types of data to reduce bias. In addition, we employed randomization in two places: once when creating the student teams, and once when assigning the programming model to the teams. Before the study started, we gave the same parallel programming training to all students to create similar starting conditions. We conducted a feasibility study to make the sure that the amount of time should suffice to complete the project in the given time. Even if we assume teams L2 and TM2 were outliers (the teams with the highest efforts), the study results would still lead to the same conclusions that the total implementation and debugging effort for locks teams is higher.

The employed STM compiler was a prototype and had some bugs, sometimes producing crashes when compiling with optimizations. However, this was the most advanced C++ STM compiler available at the time of the study. Other studies reported similar problems [34]. Due to the different types of collected data (e.g., interviews, questionnaire, personal observations), we were able to isolate situations in which the experienced problems were due to compiler bugs.

9. RELATED WORK

Empirical studies for parallel programming with TM are scarce. This is supported by a comprehensive overview of the TM literature [2]. Various Transactional Memory implementations have been proposed based on hardware [14, 20], software [4, 6, 15, 21, 28], or a hybrid of the two [10, 17]. These studies have either used small programs that exercise lists, hash tables, and other data structures, or have transformed lock-based benchmarks into TM programs [16, 34] –

for example, the Stanford ParalleL Applications for SHared memory (SPLASH-2) [31], the PARSEC benchmark suite [8], or SPEC OMP [29]. In addition, TM-specific benchmark suites have been developed, such as the Stanford Transactional Applications for Multi-Processing (STAMP) [19]. All of these benchmarks consist mostly of numerical applications. Various case studies have assessed the performance of non-trivial applications using TM (e.g., Lee's algorithm for circuit routing [1], the Linux sendmail application [26], among others [27, 11, 30]). These studies did not pay attention to software engineering aspects of TM.

Rossbach et al. [24] looked at errors in the programs of a larger number of undergraduate students from different classes, but has serious methodological shortcomings, which we avoided in our study. In [24], the groups of students and the employed STM implementations are changed over time, which does not ensure continuity. In addition, [24] is flawed because the students were asked to assign themselves to groups, and students wrongly assigned themselves and ended up in groups of different sizes, which should have been all equally sized. Moreover, our study is conducted with graduate students, an advanced industrial-strength C++ STM compiler (rather than a Java API), and a complex application (rather than a toy program). Our study has new sources of evidence (Section 2) that provide more detailed insights into how the TM language constructs were used, the thought processes of each programmer, and the actual program performance (not discussed in [24]). We also show actual examples of code patterns and anti-patterns.

10. CONCLUSION

This is the first case study to provide insights for TM programming from a variety of data, including code quality and metrics, performance, effort, and subjective programmer impressions. The study provides evidence for the necessity to employ TM and locks in a complementary way, and that they should not be considered as alternatives excluding each other. Our evidence also shows that to realize fully the benefits of TM in C++ we need language refinements supporting condition synchronization, and better debugging and performance tuning tool support. The lessons learned from this study significantly influenced the development of Intel's new generation STM Compiler. Even with TM, however, parallel programming remains difficult, so the quest for new parallel programming language features must continue.

Acknowledgements. We thank Frank Otto for helping with lab organization, and Matthias Dempe and Nikolay Petkov for assisting with performance measurements. We thank the Intel STM team, Ravi Narayanaswamy, Yang Ni, Tatiana Shpeisman, Xinmin Tian, and Adam Welc for help with code inspections. We thank Chris Vick at Sun Labs for feedback.

11. REFERENCES

[1] M. Ansari et al. Lee-TM: A non-trivial benchmark suite for transactional memory. In *Algorithms and Architectures for Parallel Processing*, pages 196–207, 2008.

[2] TM bibliography. http://www.cs.wisc.edu/trans-memory/biblio/index.html, March 2011.

[3] A.-R. Adl-Tabatabai and T. Shpeisman (editors). Draft specification of transactional language constructs for C++ (v.1.0), 2009.

[4] A.-R. Adl-Tabatabai et al. Compiler and runtime support for efficient software transactional memory. In *Proc. ACM PLDI '06*, pages 26–37, 2006.

[5] D. Bacon et al. The "double-checked locking is broken" declaration. http://www.cs.umd.edu/∼pugh/java/memoryModel/DoubleCheckedLocking.html, Dec. 2010.

[6] W. Baek et al. The OpenTM transactional application programming interface. In *Proc. ACM PACT '07*, 2007.

[7] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, May 1999.

[8] C. Bienia et al. The PARSEC benchmark suite: characterization and architectural implications. In *Proc. ACM PACT '08*, pages 72–81, 2008.

[9] C. Cascaval et al. Software transactional memory: why is it only a research toy? *Commun. ACM*, 51(11):40–46, 2008.

[10] P. Damron et al. Hybrid transactional memory. In *Proc. ACM ASPLOS-XII*, pages 336–346, 2006.

[11] R. Guerraoui et al. STMBench7: a benchmark for software transactional memory. *SIGOPS OSR*, 41(3), 2007.

[12] T. Harris et al. Composable memory transactions. In *Proc. ACM PPoPP '05*, pages 48–60, 2005.

[13] S. Heinz et al. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223, 2002.

[14] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proc. ACM ISCA '93*, pages 289–300, 1993.

[15] Intel. Intel C++ STM compiler prototype edition 2.0. language extensions and user's guide, 2008.

[16] C. J.Rossbach et al. Txlinux: using and managing hardware transactional memory in an operating system. In *Proc. ACM SOSP '07*, pages 87–102, 2007.

[17] S. Kumar et al. Hybrid transactional memory. In *Proc. ACM PPoPP '06*, pages 209–220, 2006.

[18] N. Lester et al. Efficient online index construction for text databases. *ACM Trans. Database Syst.*, 33(3):1–33, 2008.

[19] C. C. Minh et al. STAMP: Stanford transactional applications for multi-processing. In *Proc. IISWC*, 2008.

[20] K. Moore et al. LogTM: log-based transactional memory. In *Proc. HPCA'06*, pages 254–265, 2006.

[21] Y. Ni et al. Design and implementation of transactional constructs for C/C++. In *Proc. ACM OOPSLA*, 2008.

[22] V. Pankratius et al. Does transactional memory keep its promises? Results from an empirical study. Technical report, 2009-12, University of Karlsruhe, Germany, 2009.

[23] M. F. Ringenburg and D. Grossman. AtomCaml: first-class atomicity via rollback. In *Proc. ACM ICFP*, 2005.

[24] C. J. Rossbach et al. Is transactional programming actually easier? In *Proc. ACM PPoPP*, 2010.

[25] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[26] B. Saha et al. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proc. ACM PPoPP '06*, pages 187–197, 2006.

[27] M. Scott et al. Delaunay triangulation with transactions and barriers. In *Proc. IEEE IISWC*, 2007.

[28] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, V10(2):99–116, 1997.

[29] Standard Performance Evaluation Corporation. SPEC OpenMP Benchmark Suite. www.spec.org/omp, 2009.

[30] I. Watson et al. A study of a transactional parallel routing algorithm. In *Proc. ACM PACT '07*, pages 388–398, 2007.

[31] S. C. Woo et al. The SPLASH-2 programs: characterization and methodological considerations. In *ACM ISCA*, 1995.

[32] R. K. Yin. *Case Study Research: Design and Methods*. Sage Publications, Inc, 3rd edition, 2002.

[33] C. Zannier et al. On the success of empirical studies in the International Conference on Software Engineering. In *Proc. ACM ICSE '06*, pages 341–350, 2006.

[34] F. Zyulkyarov et al. Atomic quake: using transactional memory in an interactive multiplayer game server. In *Proc. ACM PPoPP '09*, pages 25–34, 2009.