Theory Comput Syst (2014) 55:555–590
DOI 10.1007/s00224-013-9452-5

Software Engineering with Transactional Memory Versus Locks in Practice

Victor Pankratius · Ali-Reza Adl-Tabatabai

Published online: 3 March 2013

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 article, 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 article provides valuable lessons for improving TM.

Keywords Parallel programming · Transactional memory · Language design · Human factors · Synchronization · Programming techniques

Present address:
V. Pankratius (✉)
Massachusetts Institute of Technology, Cambridge, MA, USA
e-mail: pankrat@mit.edu

A.-R. Adl-Tabatabai
Programming Systems Lab, Intel Corporation, Santa Clara, CA, USA
e-mail: ali-reza.adl-tabatabai@intel.com

1 Introduction

Multicore is a challenge for software engineering, and we need mainstream languages that support productive and robust parallel programming. 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++ [22], Java [2], Haskell [14], and ML [27] 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 such as [29] of TM versus locks focused on small programs or micro-benchmarks to evaluate the worst case performance, but none of them took into account more complex applications to evaluate software engineering aspects over a longer period of time. Other work studying the conversion of locks programs to TM did not shed light on the issues encountered when parallelizing with TM from scratch [41].

This article 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 at Karlsruhe Institute of Technology (KIT) in Germany.[1] 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 [22]—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 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 article also shows that TM does not solve all concurrency control problems, and thus is not a silver bullet. In particular,

[1] This study has been conducted while the first author was at KIT in Germany.

both the locks and TM teams had data races because they used incorrect double-checked locking patterns [4].

The article is an extended version of [24] 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 article is organized as follows. Section 2 summarizes the parallel programming models in this study. Section 3 presents the project requirements, the STM compiler, and collected evidence. Section 4 presents code metrics focusing on productivity and the use of parallelism constructs. Section 5 discusses the results of code inspections for all the programs. Section 6 breaks down the effort each team spent on parallelization, tuning, and debugging. Section 7 measures the performance of each team's search engine. Section 8 summarizes key results from our study. Section 9 discusses potential threats to validity. Section 10 contrasts related work. Section 11 provides our conclusion.

2 The Parallel Programming Models in This Study

Most programmers today use shared-memory parallel programming techniques to program multicore computers. Mainstream programming languages provide constructs to create concurrent threads of control, to synchronize concurrent access to shared data, and to co-ordinate thread execution. While earlier languages such as C or C++ use standardized APIs (e.g., Pthreads, the Posix Threads library [8]) to provide parallel programming constructs, more recent languages like Java and C# have native language support. Large-scale scientific, industrial, and open-source projects mostly use C, C++, and Pthreads. We therefore pick the Pthreads approach as representative for programming with locks and provide a brief overview next.

2.1 Programming with Locks

Pthreads is a standardized, platform independent API for parallel programming in C and C++ [8]. The programming approach with Pthreads is very low level: programmers must manually create and manage threads, insert locks (mutexes) for mutual exclusion, define condition variables to coordinate concurrent producer-consumer processing, and manage thread-local storage.

The motivation for this style of programming is performance, giving developers more control and reducing overhead. This flexibility comes at a price, with well-known pitfalls. Simplistic coarse-grain locking can result in poor scalability due to lock contention, which can be eliminated in several ways: Fine-grain locking associates separate locks with individual shared data items accessed inside critical sections so that threads that access disjoint data items can execute in parallel. Reader-writer locks allow more than one thread to read shared data in parallel inside critical sections. Unfortunately, all these optimization techniques expose a programmer to concurrency bugs, namely deadlocks, data races, and atomicity violations (also known as high-level data races). Moreover, incorrect use of lock-based condition synchronization can lead to lost wake-up bugs.

In addition to risking new bugs, locks also don't support programming in the large very well, in which distributed development teams build large programs out of separately-authored software components. After optimizing the locking inside a software component, a programmer is not guaranteed that the performance of the optimized component will scale once it is composed with other components in a parallel program. Locks also make providing exception safety guarantees at component boundaries more difficult; a programmer must carefully release the right locks in the right order inside exception handlers, and avoid exposing broken invariants to other threads and introducing data races by releasing locks before recovering from the exceptions.

2.2 Programming with Transactional Memory

Software Transactional Memory employs atomic transactions instead of locks. A programmer defines a transaction by enclosing a set of programming language statements in an atomic block. Such a block represents a critical section and must contain only statements with reversible effects. A run-time system allows threads to execute atomic blocks concurrently while making it appear that only one thread at a time executes within an atomic block. If a concurrently executing transaction conflicts with another transaction, the run-time aborts it (i.e., undoes its effects) and retries it later on; otherwise, it commits it and makes its effects visible to all other threads. The run-time system basically enforces the atomicity, consistency, and isolation properties known from database transactions [12] that now apply to programming language statements.

TM promises to alleviate many of the challenges of parallel programming with locks. It relieves the programmer from low-level locking details, such as dealing with multiple locks and complex locking protocols. It also eliminates deadlocks due to incorrect ordering of locks. TM can make it easier for programmers to recover from exceptions and errors by providing failure atomicity. The typical approach is that TM systems implement a rollback mechanism exposed to the programmer via an explicit abort construct or an via implicit rollback on uncaught exceptions.

TM also has pitfalls. It is still possible to have data races. Large transactions can hurt scalability and performance. For example, programmers may attempt to optimize their transaction-based code by shrinking the size of atomic blocks, moving code out of atomic blocks, or breaking atomic blocks into smaller ones. These transformations

may inadvertently introduce data races in which a variable is accessed concurrently both inside and outside a transaction.

Recent research on adding TM support to programming languages has also uncovered numerous tradeoffs in language design and in the kind of TM-related features that can be provided to the programmer. More empirical studies and experiments with real-world parallel programs are needed to help assess the right tradeoffs to make, and when it is appropriate to use TM.

Despite these potential pitfalls of TM, its proponents argue that the combination of automatic fine-grain concurrency control, automatic failure atomicity, and reduced potential for deadlocks, all allow TM to support modular programming in the large better than locks can. Supporters also argue that although TM is not a parallel programming silver bullet, it is a step in the direction of providing a much more robust and productive concurrency control mechanism compared to today's locks.

2.3 The Intel STM Compiler

In this study, we used Intel's STM compiler as a representative implementation of the TM approach, because it is one of the most advanced STM compilers so far. The Intel compiler is an industrial-strength C and C++ compiler that has been extended with a prototype implementation of transactional language constructs for C++ [22]. Part of the extensions are annotations to functions and classes to mark functions that will be called inside transactions.

The `__tm_atomic` keyword defines an atomic block of statements. Atomic blocks can be nested, which means that the effects of inner transactions are only visible 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 `__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; it is the programmer's responsibility to make sure that such functions can be called inside transactions without instrumentation.

3 Case Study Design

This section presents an overview of the project, programming scenario, and team experience. We also discuss the sources used to collect empirical evidence.

3.1 The Project

Every team developed a desktop search engine based on the following indexing and search requirements. 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.

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.

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.

3.2 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 teams had access to the multicore machine that was finally used by the instructors to benchmark each submission.

Before the study started, we gave the same parallel programming training to all students covering common parallel programming issues such as race conditions, deadlocks, as well as performance optimization, reductions, and other topics. 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.

The TM teams were allowed to use only 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 students were allowed to do so.

3.3 Team Experience

All students had Bachelor-equivalent degrees in computer science and were pursuing Master's degrees in computer science at KIT. 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.

Figure 1 shows the experience profile of all teams prior to the study. Each axis shows the years of experience with programming languages, libraries, parallel programming approaches, tools, and operating systems. The "Semester" axis shows the number of semesters the student has been enrolled in the university since high school. We also collected proficiency data, but this data did not appear to provide any more insight than the experience data, so we omit it.

It is valuable for a case study to observe and compare the performance of teams that have a wide variety of experiences. The profiles show that some teams have less overall experience than others; for example, team TM1 has less experience than team TM2 or team TM3. Additional data not shown in Fig. 1 is presented in [25].

3.4 Collection of Evidence

Throughout this study, we followed the recommendations of [30, 39, 40] 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 6 analyzes these times reports. (3) Notes from weekly (semi-structured) interviews [30] 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 [25]. (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.

4 Code Metrics

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

4.1 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 Sect. 5 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.

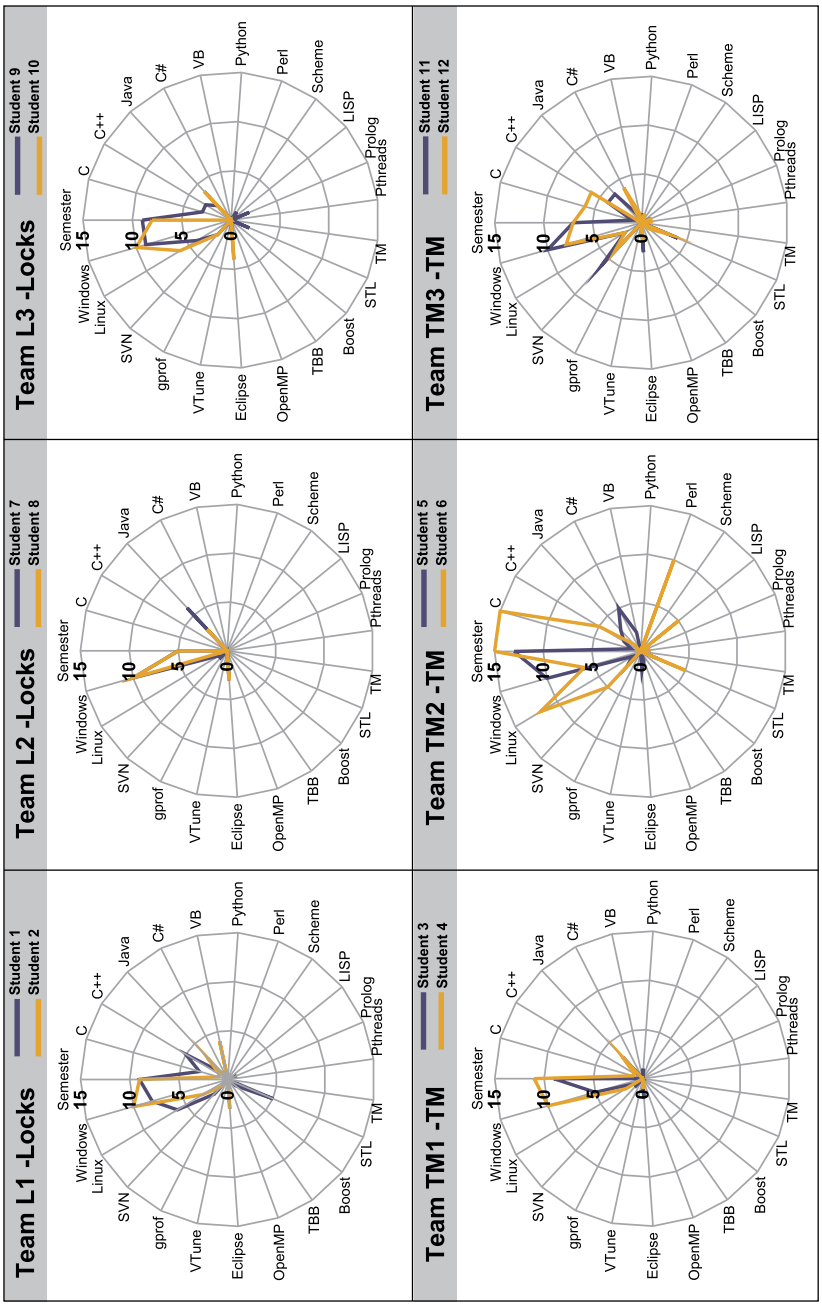

Fig. 1 Team experience prior to study. Each *chart* represents a team, and each *line* represents years of experience for a student

Table 1 Progress summary from locks teams interviews

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.

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 Sect. 5, 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

Table 2 Progress summary from TM teams interviews

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.

	Locks teams			TM teams		
	L1	L2	L3	TM1	TM2	TM3
Total Lines of Code (LOC, excluding comments, blank lines)	2014	2285	2182	1501	2131	3052
	avg: 2160 stddev: 137			avg: 2228 stddev: 780		
LOC pthread*	157 8%	261 11%	120 5%	17 1%	23 1%	12 0%
LOC tm_*	0	0	0	36 2%	22 1%	139 5%
LOC with parallel constructs (pthread* + tm_*)	157 8%	261 11%	120 5%	53 4%	45 2%	151 5%
	avg: 179 stddev: 73			avg: 83 stddev: 59		
Total effort in person 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		
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
- min;max;stddev of LOC/critical section	1; 78; 12.7	1; 14; 3.7	1; 45; 11.9		85; 85; -	3; 6; 2.1
- #crit. sections containing function calls	25	18	29		1	
- total #function calls from critical sections	68	56	119		38	
- #crit. sections 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
- min;max;stddev of LOC/atomic sect.				1; 14; 4.4	1; 21; 5.3	1; 32; 8.2
- #atomic sections containing function calls				6	3	16
- total #function calls from atomic sections				11	5	41
- #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

Fig. 2 Code metrics for the parallel desktop search engines of all teams

incorrectly in a racy fashion to optimize performance instead of being used for failure atomicity.

Figure 2 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.

4.2 Usage of Parallel Constructs

Locks and TM teams clearly differ in how many lines of code contain parallel constructs (Fig. 2). 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 Sect. 5, 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 [4].

4.3 Comparison of Critical Sections

The critical sections differ for locks teams and TM teams. Figure 3 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.

5 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

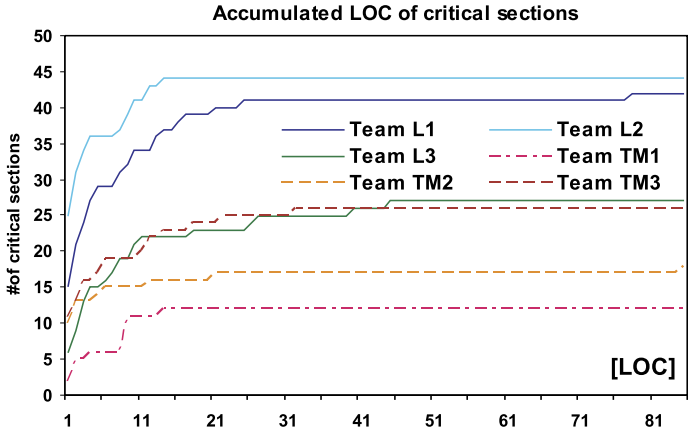

Fig. 3 Code distribution in critical sections

anonymized 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.

5.1 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 [4].

5.2 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 Fig. 2). 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.

5.2.1 Team L1

Architecture and Data Structures A single crawler thread traverses the directories and parses each file to generate tuples into a single shared work queue. Each tuple consists of a word to be indexed, its file, and its file position. A pool of worker threads take tuples from the queue one-at-a-time and concurrently update a shared index data structure.

The index data structure consists 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. Each word contains a map of documents and the list of document positions in which that word appears. To speed up queries with wildcards at the beginning, a second array of sub-indices holds a map storing all words ending with a particular character.

Team L1 designed the queries to work in parallel and to work concurrently with the indexing threads. Each query spawns a new thread, which in turn spawns a child thread for each word in the query—this seems legitimate, but thread creation overhead might be a problem. Each child thread traverses the index independently. The initial query thread waits by joining on its children and combines the results. For multi-word queries, this approach forks a thread per word.

Synchronization Team 1 used locks to protect three areas of their program: (1) a lock to protect the work queue plus two condition variables for producer-consumer synchronization on the queue. (2) several locks protecting code that displays information; and (3) 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 (27×2) locks

protecting the index structure. The number of dynamic lock instances is therefore greater than in Fig. 2. Inserting into the index requires acquiring 2 locks on the sub-indexes. A look-up in the index requires acquiring a lock on the index or a lock on the reverse index if the query contains a wildcard.

To avoid deadlocks, the team specified an order for acquiring locks in these lock arrays, documented as comments in a header file:

```
/************************
 * !!! MUTEX ORDER !!!
 *   To prevent any deadlock, mutexes
 *   have to be locked in the following order:
 * - mMtxFileIndex
 * - mMtxKeywordIndexInverted[0]
 * - mMtxKeywordIndexInverted[1]
 * - ...
 * - mMtxKeywordIndex[0]
 * - mMtxKeywordIndex[1]
 * - ...
 * - mMtxDifferentKeywords
 *********************/
```

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. Some pieces of code, including comments, are reused in many places.

Despite the use of per-character locks for the index, team 1 had the worst indexing performance (Fig. 5). Their indexing performance gets worse as the number of worker threads increases beyond 3 threads. Team L1 described experimental results in their final report that point to the single work queue as a potential performance bottleneck. The report also mentions that they had performance problems with their initial indexing structure, which did not have sub-indexes. The team found out that the execution time of queries depended on the frequency of the terms, so locks protecting the per-letter indices might not have been appropriate for more frequent letters. Nevertheless, the team did not re-design the indexing data structure to take advantage of these insights.

Ease of Code Understanding Team L1's code is visually pleasing, with verbose comments, although there is a mismatch between the comments in the code and documentation that they submitted. Nevertheless, team 1's code is difficult to understand. It is difficult to reason that the code has no data races. Their code has many locks, and documentation lacks details on which variables are shared, or on how locks are associated with variables.

Indexing is mixed with querying; some undocumented parameters with unfortunate names (e.g., "bool yes") are used to steer the process, and in addition have different meanings in different places. Their submission also contained some dead code.

Problems with Usage of Language Constructs This team used many locks and threads, which increased the complexity of their parallel code. They knew that their queue design would be a performance bottleneck, but the code seemed to be difficult to modify later on.

5.2.2 Team L2

Architecture and Data Structures A crawler thread traverses directories and inserts file names into one or more work queues. Each queue is implemented to have a set of worker threads that take file names from the queue, parse the files, and concurrently update a shared index data structure. When a queue is empty, its threads move to work on another non-empty queue. In addition to the number of indexing threads, the team introduced various other command-line parameters to simplify performance experiments. They finally fixed the number of threads per queue to one thread per queue; other parameters and their values that were used for performance measurements (see Fig. 5) are explained next.

To balance the load, each queue tracks the sum of file sizes indexed so far. The crawler thread first checks if a file is above a certain predefined file size threshold; if so, it assigns the file to the queue with the lowest sum so far. Otherwise, the crawler thread assigns the file in a round-robin fashion to queues. Team L2's rationale was that for small files, they did not want to incur additional overhead for a more complex choice of queues. They finally fixed the value threshold for small files to 500 KB.

Another file size threshold specifies where the crawler thread inserts a file name in a queue. Below this threshold, indexing threads dequeue files from the front and the crawler thread adds a new file name at the end of a queue. Above this threshold, the crawler thread appends at the front. The team explained that they wanted to achieve an early indexing of big files with this strategy, and fixed this threshold to 1000 KB after experimenting with different values.

In the inverted index data structure, 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 = "abcdefghijklmnopqrstuvwxyz0123456789"
//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.

Like the prior team, team L2 designed queries to work in parallel and to work concurrently with the indexing threads. A main query thread takes user input and forks off new threads that query the index in parallel. These threads are created per word for every part of the query and store partial results in temporary buffers. When all query threads are finished, combiner threads are started in parallel to aggregate the buffers and produce the final result.

Synchronization Each sub-index has a lock, so team L2 has over 1600 locks. Similar to team 1, the number of dynamic lock instances is greater than in Fig. 2. An indexer thread has to acquire at least a lock on one of the queues to read a file name, and two locks on a sub-index.

For each class that might be shared, they define a member field called *access_mutex*. They want to emulate per-object monitors, such as those found in Java. As seen in Figure 1, this team indeed has a Java background and no C++ background. This programming pattern reflects Team L2's Java background.

In Fig. 2 they have more unlock operations compared to locks because in a few cases, they don't use locking in a block-structured manner, but perform unlock in *then* and *else* branches.

Team L2 used condition variables for producer-consumer synchronization of their queues.

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. The following example illustrates multiple races due to unprotected read accesses to the *jobs* and *empty* member fields:

```
//jobs.h
class Jobs {
 public:
  int size();
  void add(StringFile file,...);
 private:
  std::deque<StringFile> jobs;
  bool empty;
  pthread_mutex_t access_mutex;
...
//jobs.cc
...
void Jobs::add(StringFile file,...) {
  pthread_mutex_lock(&access_mutex);
    ... jobs.push_back(file);
    ... empty = false;
        pthread_cond_broadcast(&wait_condition);
  pthread_mutex_unlock(&access_mutex)
};
...
int Jobs::size() {return jobs.size();} //unprotected read
bool isEmpty() {return empty;} //unprotected read
}
```

In another case, they return a pointer to an object contained within the inverted index whose updates are guarded by a lock, but the accesses performed through the returned

pointer are not guarded by that same lock. This causes a race between the indexer and the query processor.

Ease of Code Understanding Their complex parallelization scheme was not easy to understand from the code. Many parameters are not expressive or lack appropriate comments. A lot of information had to be inferred from the more general documentation. They create many threads, often dedicated to different types of work, interacting with different queues. It is hard to say if everything works as they intended.

Problems with Usage of Language Constructs Some source code comments suggest that they tried to compare a C++ object—and not a pointer to an object—to NULL. Moreover, they put a lot of code in their header files, despite being taught not to do so.

5.2.3 Team L3

Architecture and Data Structures Team L3 has a pool of indexing threads each of which has a queue containing words and document positions to index. A crawler thread traverses directories, parses the files, and inserts the words and document positions in a round robin fashion into the queue of each indexing thread. Indexing threads consume the words in their queue and update the index data structure. Team L3 uses condition variables for producer-consumer synchronization of the work queues.

Compared to teams L1 and L2, both of whom used maps for their index data structures, this team read more research papers to find a suitable indexing structure. They finally used a BurstTrie based on [15], which is a more complex tree-based data structure than the map-based data structures of teams L1 and L2. Part of their time spent on reading was reading [15]. In addition, like team L1, they have a reverse index (also a BurstTrie) to speed up queries with wildcards at the beginning.

Team L3 also designed queries to work in parallel and to work concurrently with the indexing threads. They spawn a sub-query thread for each word in the query.

Synchronization They have 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 1, 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
 ...
}
```

Ease of Code Understanding Many of their source code comments help; header files have detailed comments for method parameters. There are also many useless comments (e.g., many one-line methods having the comment “algorithm: trivial”).

Locking is difficult to understand because the lock and unlock operations are not used block-wise in several parts of the program.

Problems with Usage of Language Constructs Team L3 did not use locks in a block-structured way, which made their use of locks difficult to understand and verify by inspection. The locking protocol is also not well-documented.

5.3 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 transaction and then checking it inside, similar to incorrect implementations of the double-checked locking pattern [4].

5.3.1 Team TM1

Architecture and Data Structures A crawler thread traverses directories and builds up a list of files to be indexed, sorted by file size. There is a single shared work queue between the crawler thread and the indexer threads. Several indexing threads go through the documents and build up the index. The crawler thread runs concurrently with the indexing threads.

They use 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.

Queries containing several words use one thread per word. They didn't finish other types of searches. Wildcard searches are partly implemented and are intended to generate several threads that search the matching sub-indexes in parallel.

They did not try out their program on the benchmark given in the lab (742 MB), but rather on two small sets of files (21 MB with 8000 files, 120 MB with 214 files). Their submitted version consumed too much memory and crashed. It was too late when they discovered this problem. They were finally excluded from the competition.

Synchronization Team TM1's code has clear data races that could make the program crash. In the following code, they traverse a linked list starting from the sorted start node without the proper synchronization:

```
//called by crawler thread
void FileIndex::add_File(string filename, int size) {
 sortedFileNode* newNode = new sortedFileNode(...);
 sortedFileNode* tempNode;
 if (sortedstartNode == NULL) {...}
 else {
  tempNode = sortedstartNode;
  while(tempNode->get_next() != NULL &&
        tempNode->get_next()->get_size() > size)
       {tempNode = tempNode->get_next();}
   __tm_atomic {
      newNode->set_next(tempNode->get_next());
      tempNode->set_next(newNode);
   }
}
...
```

In their documentation, they mention that they tried to design the program in a way that reduced transactional conflicts. They also mentioned that TM was easy to use and that it and helped avoid many sources of errors. Yet their program crashed during benchmarking and clearly contained 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;
}}}}
```

Ease of Code Understanding Functions in header files had comments (e.g., for explaining the meaning of constants). Team TM1 were rather naïve and inexperienced programmers. Despite being taught not to do so, large portions of code were contained in header files.

Problems with Usage of Language Constructs They did not use TM function attributes in header files, but only in definitions in the .c files.

5.3.2 Team TM2

Architecture and Data Structures Unlike all other teams, this team 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, although not required.

This team used a modified B-tree according to the approach described in [19]. They looked at B-Tree implementation of scalingweb.com, but considered it too general and having too many functions. Because they were unsure how long an adaptation would take, they developed their own data structure in C. They used C except for querying, where they used C++.

Queries are not parallelized; however, they do run concurrently with the indexer, as was required. A second program performs the querying and uses the on-disk index. Queries are single-threaded and run in a separate program from the indexer threads; therefore, they did not use any synchronization in the queries.

Synchronization They create a background thread to print statistics periodically. They have short atomic blocks that mainly update the B-tree and the statistics concurrently.

Surprisingly, they use the *tm_callable* annotation only twice (for functions accessing the B-Tree) one of which was even unnecessary.

They use the *tm_pure* construct for a string comparison function (c and header file); this function is used in one place to compare a given word with a word in the B-tree. Since both words are immutable, this use of *tm_pure* is correct. Another usage of *tm_pure* was for annotating a custom *fprintf* function that was used throughout the program to store debugging messages in a file.

They used a global Pthreads mutex lock in an I/O function that returns the next text file to parse. The lock is not used in a block-structured manner; unlocks are performed in two different locations. The critical section beginning with the lock operation and ending with one of the unlock operations has 85 LOC and is the longest in their program. By contrast, their longest atomic block has 21 LOC.

Team TM2 also assumed that reading shared variables without protection is safe. This could be a reason for their program to crash:

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

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;
 }
}
...
```

Ease of code understanding Their code has almost no comments. They have a "compact" style of C programming, due to one of the team members being an experienced C programmer. Their atomic blocks are easy to understand. Part of their functionality implementing their indexing was difficult to understand, even by the experienced code reviewers.

Problems with Usage of Language Constructs This team misunderstood the purpose of nested transactions. They used statically nested atomic blocks in two places where they put updates of statistics into their own nested transactions. The inner atomic just updates statistics and has no abort statement, which means that they did not use nested transactions for failure atomicity.

5.3.3 Team TM3

Architecture and Data Structures A crawler thread 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 [6], which is a tree-like data structure with nodes representing shared prefixes of index terms. The shared prefix structure is also advantageous for wildcard searches. To speed up wildcard queries, they add into the tree the reverse of an indexed word, and put a pointer from the last character node of the reversed word to the last character node of the indexed word.

Queries are not parallelized, and querying uses just one thread.

Synchronization 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 smaller transactions back-to-back instead of few big transactions, to optimize performance.

Their indexer code has a race, as it uses a variant of double-checked locking [4]. They are checking outside a transaction if their stack of files is empty, and perform a *pop* operation inside that 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);
 ...
```

They used TM attributes in header files on the declarations of functions. They were the only team to use TM attributes on template functions. They also have *assert* statements inside transactions as well as *printf* constructs for debugging.

Ease of Code Understanding Despite very few comments, their code is quite readable. They have just one nested atomic section, but it's unclear why they employed it.

Problems with Usage of Language Constructs There are *ifdef DEBUG* blocks with *cout* operations inside transactions to print debugging messages. The compiler statistics and support for debugging were not sufficient.

If a Standard Template Library (STL) for TM was available, they would not have to write their own atomic dictionary or vector data structure. Thus, they could have been even more productive.

6 Programming Effort

Throughout the project, each team had to fill out a form tracking how many hours they spent per day on a certain task category. Figure 6 in the Appendix shows the categories of tasks that were tracked, and Fig. 4 presents the data in terms of person-hours spent by each team per category. The categories "read documentation" (1), "search for libraries" (2), and "experiments" (5) factor out effort that might otherwise be counted as "implementation" (category 4). This increases the validity of the numbers reported for implementation, as they are not mixed with other tasks. The "other tasks" category (8) consists of tasks that do not fit into the defined categories and are not considered to be interesting enough to be split up for the study. Figures 7–13 in the Appendix present detailed effort measurements.

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 Fig. 5.

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 Fig. 4 and the interview data.

6.1 Interview Summaries

6.1.1 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

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

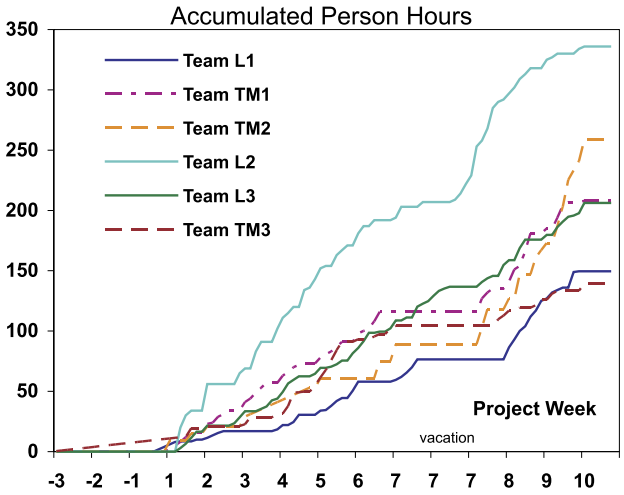

Fig. 4 Total effort of all teams in person hours

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.

6.1.2 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

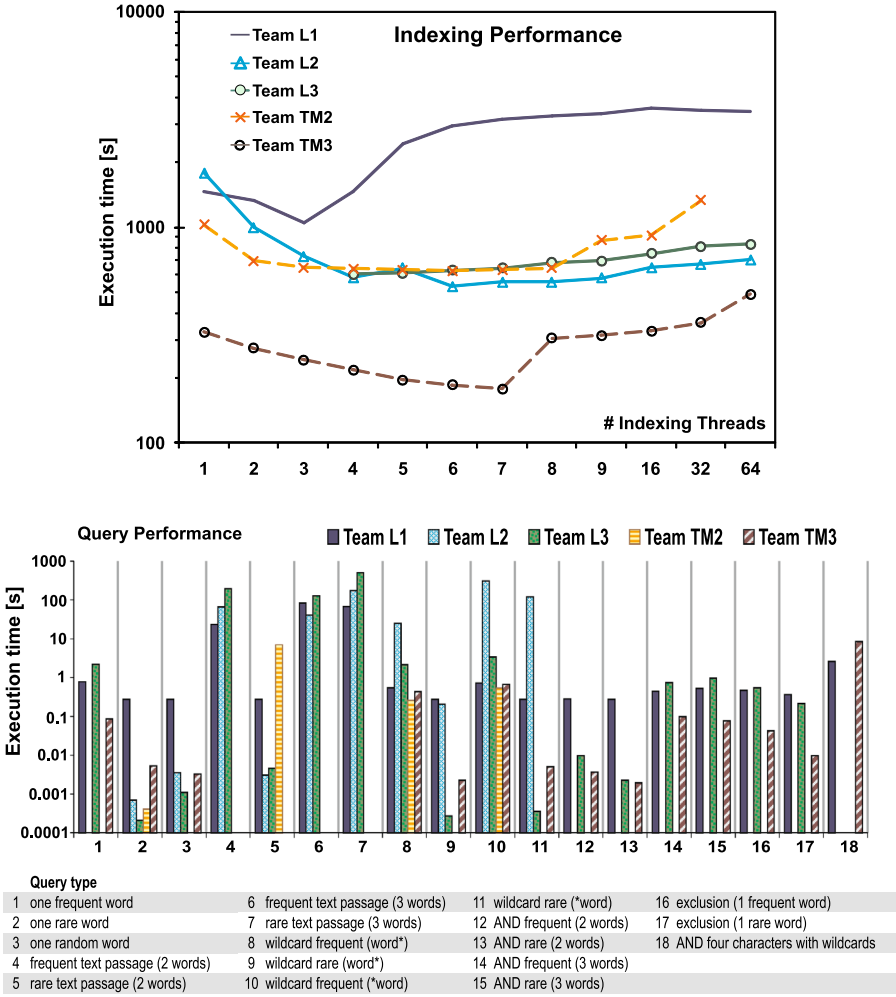

Fig. 5 Indexing and query performance. The *right graphs* excludes for each team the queries that are not implemented or not executing correctly

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. 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.

6.2 Parallelization

TM teams spent in total about half as much time as the locks teams on writing parallel code (see [25] 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.

6.3 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.

6.4 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 [25]. 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 [25]. According to Fig. 2, 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 [25].

7 Performance Measurements

Figure 5 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.

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 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 [25].

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 Fig. 5). 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 Fig. 4 if the locks teams had tested their programs in a fashion comparable to the TM teams.

8 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 show 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. While TM implementations have used locks under the covers (e.g., the STM runtime used in the Intel STM compiler [22]), researchers have devoted little effort to programming models that provide both locks and atomic blocks with clear semantics. Our practical application provides evidence that we need to expand the theory of combining transactions and locks [10, 37]. Intel's recent Specification of Transactional Language Constructs for C++ [1] 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.

9 Threats to Validity

A case study provides detailed insights on one case being studied [30]. 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 [41]. 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.

10 Related Work

Empirical studies for parallel programming with TM are scarce. This is supported by a comprehensive overview of the TM literature [35]. Various Transactional Memory implementations have been proposed based on hardware [16, 21], software [2, 5, 17, 22, 33], or a hybrid of the two [11, 18]. 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 [28, 41]—for example, the Stanford ParalleL Applications for SHared memory (SPLASH-2) [38], the PARSEC benchmark suite [7], or SPEC OMP [34]. In addition, TM-specific benchmark suites have been developed, such as the Stanford Transactional Applications for Multi-Processing (STAMP) [20]. 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 [3], the Linux sendmail application [31], among others [13, 32, 36]). These studies did not pay attention to software engineering aspects of TM.

Rossbach et al. [29] looked at errors in the programs of a larger number of undergraduate students from different classes, but in a much less rigorous study than we present. For example, [29] had unequal groups of students who employed STM implementations that changed over time. By contrast, our study ensures continuity and the graduate students use an advanced industrial-strength C++ STM compiler to create a complex application.

11 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. To make this search easier, programming language researchers must develop and employ more sophisticated usability evaluation techniques for language constructs [23, 26].

Acknowledgements We thank KIT and the Excellence Initiative for supporting the first author during this study. We also thank Frank Otto for helping with lab organization, and Matthias Dempe and Nikolay Petkov for assisting with performance measurements. For code inspections, we appreciate the feedback of the Intel STM team, Ravi Narayanaswamy, Yang Ni, Tatiana Shpeisman, Xinmin Tian, and Adam Welc. Further feedback was provided by Chris Vick at Sun Labs.

Appendix

1. Read documentation
(e.g., program or library documentation, papers, help files, Web pages)

2. Search for suitable libraries

3. Conceptual development and design

4. Implementation
- **4.1 Implementation of mostly sequential code**
  (i.e., the thought process is mostly sequential, implementation activities do not take parallelism into account)
- **4.2 Implementation of mostly parallel code**
  (e.g., using Pthreads or TM constructs; the thought process is centered on parallelism, interactions among threads or transactions, etc.)
- **4.3 Refactoring**
  (i.e., transformation or reorganization of a program without changing its behavior)
- **4.4 Other implementation tasks**

5. Experiments
(this category captures the hours spent on experiments that are independent of the search engine, in order to gain additional knowledge or understand general performance issues. For example, this may include the implementation of small test programs.)
- **5.1 Trying out parallelization constructs**
- **5.2 Trying out library calls**
- **5.3 Performance experiments**
- **5.4 Other experiments**

6. Testing
- **6.1 Functional tests**
  (e.g., does the indexing work correctly? Are query results correct?)
- **6.2 Performance tests**
  (assessing the performance of the desktop search engine itself and tuning its performance parameters. Different from 5.; the object tested here is the search engine, not another program.)
- **6.3 Tests for ensuring correct parallel operation**
  (e.g., detecting incorrect synchronization behavior, race conditions, atomicity violations)
- **6.4 Integration tests**
  (e.g., for employed libraries, code of other team members)
- **6.5 Other tests**

7. Debugging
because of
- **7.1 segmentation faults**
- **7.2 unexpected or ”strange” program behavior**
- **7.3 incorrect input/output**
- **7.4 integration problems between libraries and other search engine code**
- **7.5 integration problems of code produced by different team members**
- **7.6 problems with Pthreads constructs**
- **7.7 problems with TM constructs**
- **7.8 other causes**

8. Other tasks

Fig. 6 Effort categories logged by each team

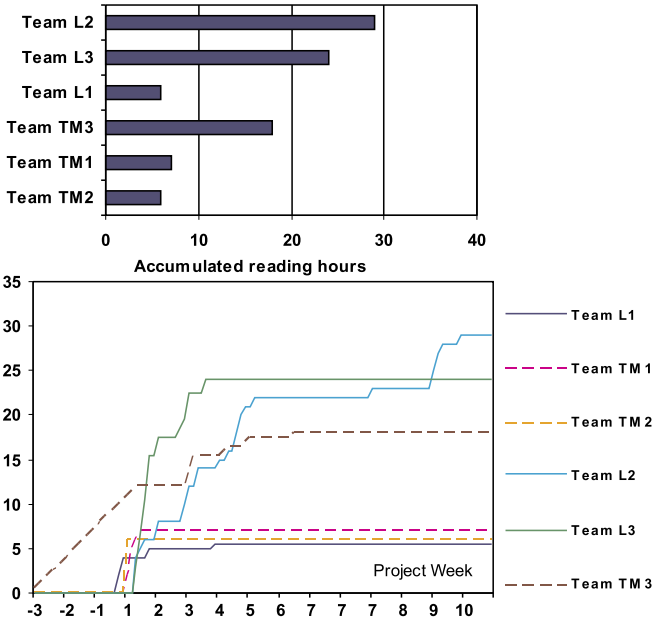

Fig. 7 Effort category 1: Time spent on reading

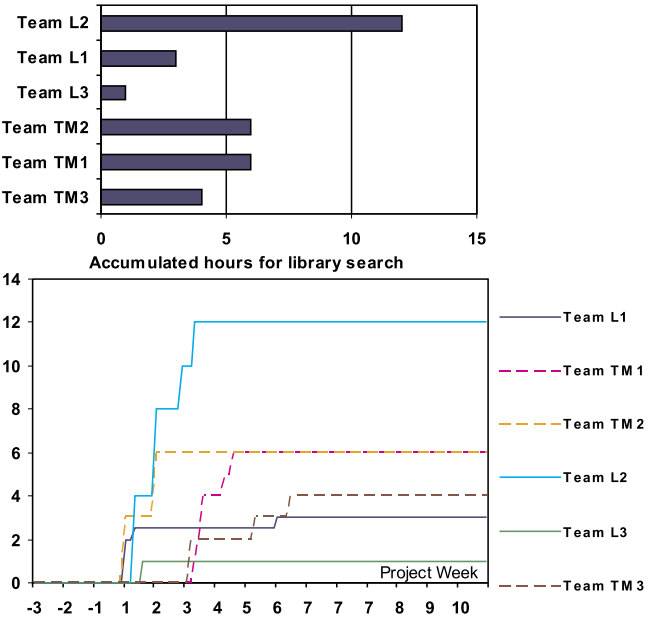

Fig. 8 Effort category 2: Time spent on search for libraries

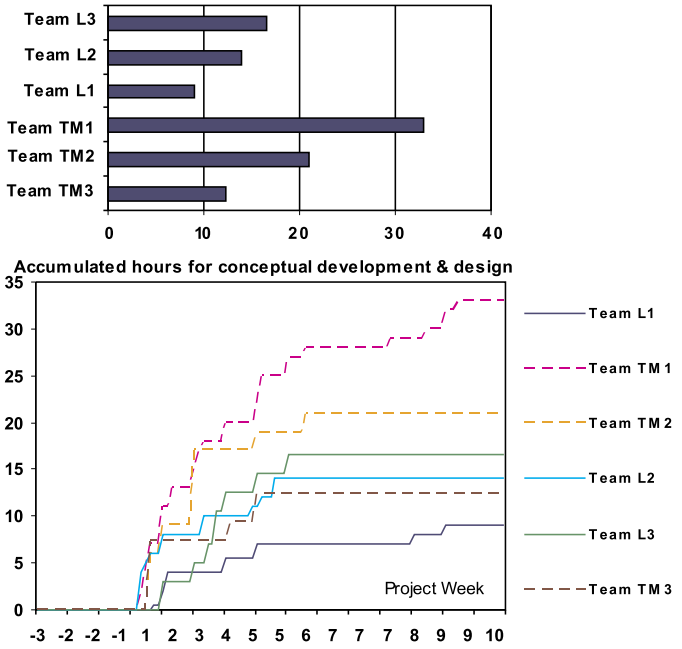

Fig. 9 Effort category 3: Time spent on conceptual development and design

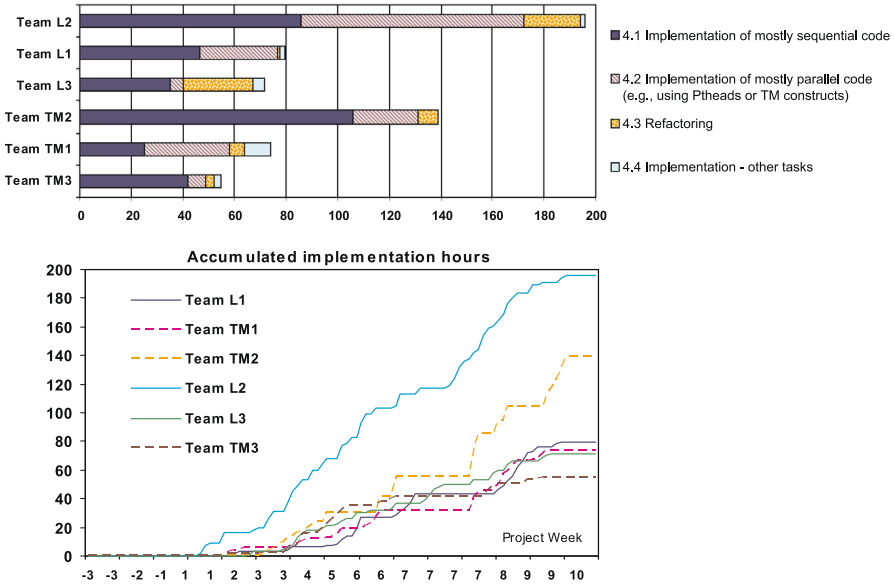

Fig. 10 Effort category 4: Time spent on implementation

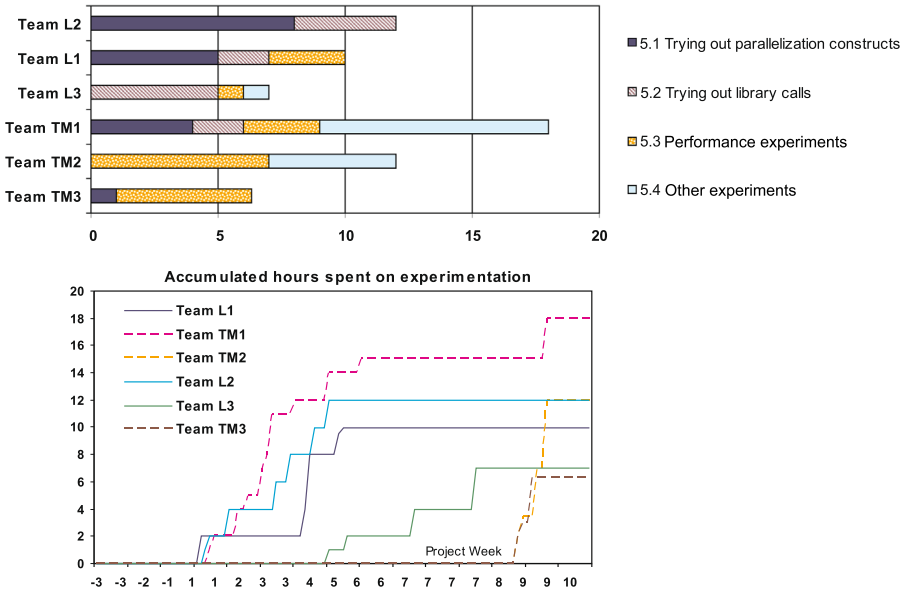

Fig. 11 Effort category 5: Time spent on experimentation

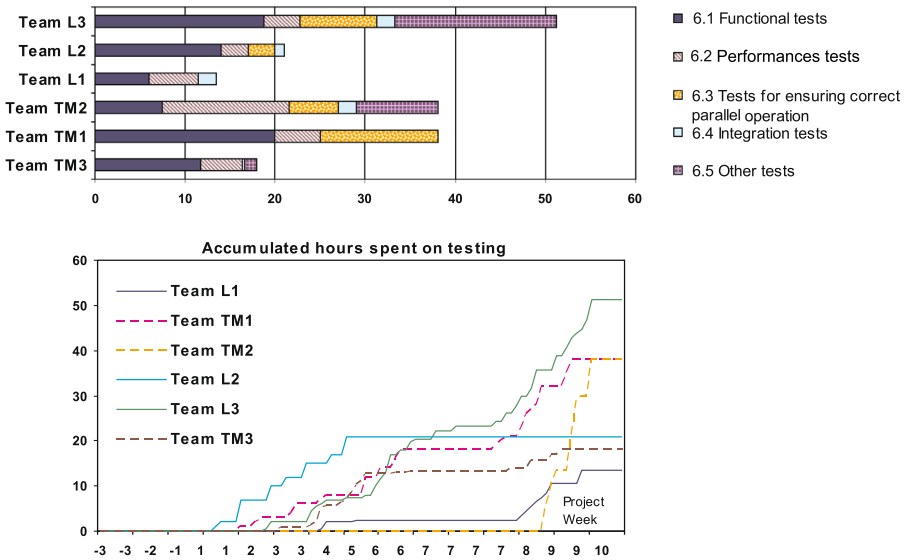

Fig. 12 Effort category 6: Time spent on testing

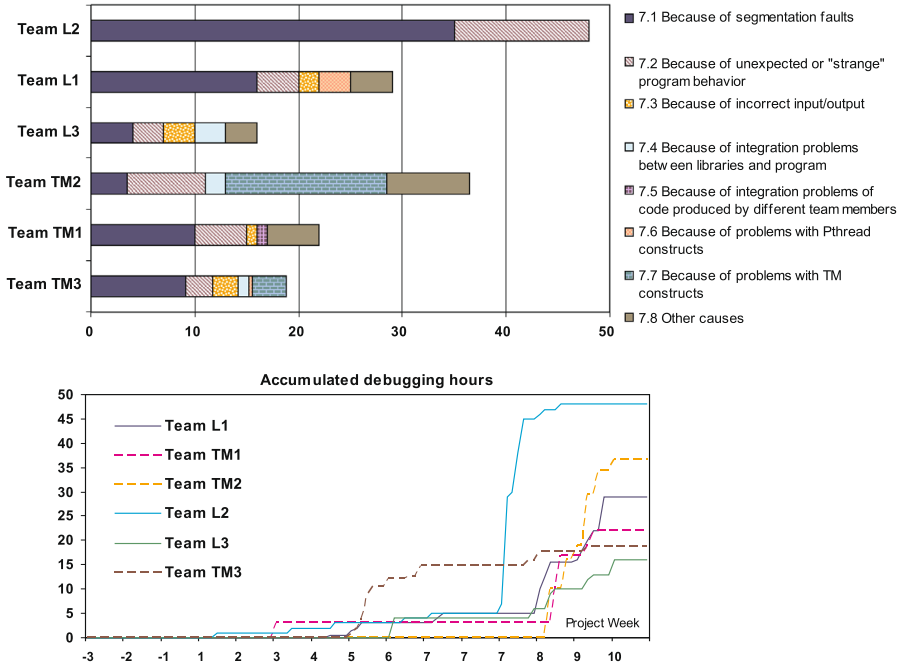

Fig. 13 Effort category 7: Time spent on debugging

References

1. Adl-Tabatabai, A.-R., Shpeisman, T. (eds.): Draft Specification of Transactional Language Constructs for C++ (v.1.0) (2009)
2. Adl-Tabatabai, A.-R., et al.: Compiler and runtime support for efficient software transactional memory. In: Proc. ACM PLDI'06, pp. 26–37 (2006)
3. Ansari, M., et al.: Lee-TM: a non-trivial benchmark suite for transactional memory. In: Algorithms and Architectures for Parallel Processing, pp. 196–207 (2008)
4. Bacon, D., et al.: The "double-checked locking is broken" declaration. http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html (2011)
5. Baek, W., et al.: The OpenTM transactional application programming interface. In: Proc. ACM PACT'07 (2007)
6. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley, Reading (1999)
7. Bienia, C., et al.: The PARSEC benchmark suite: characterization and architectural implications. In: Proc. ACM PACT'08, pp. 72–81 (2008)
8. Butenhof, D.R.: Programming with POSIX Threads. Addison-Wesley, Reading (1997)
9. Cascaval, C., et al.: Software transactional memory: why is it only a research toy? Commun. ACM **51**(11), 40–46 (2008)
10. Dalessandro, L., et al.: Transactional mutex locks. In: EuroPar (2010)
11. Damron, P., et al.: Hybrid transactional memory. In: Proc. ACM ASPLOS-XII, pp. 336–346 (2006)
12. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Mateo (1993)
13. Guerraoui, R., et al.: STMBench7: a benchmark for software transactional memory. Oper. Syst. Rev. **41**(3), 315–324 (2007)
14. Harris, T., et al.: Composable memory transactions. In: Proc. ACM PPoPP'05, pp. 48–60 (2005)
15. Heinz, S., et al.: Burst tries: a fast, efficient data structure for string keys. ACM Trans. Inf. Syst. **20**(2), 192–223 (2002)

16. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: Proc. ACM ISCA'93, pp. 289–300 (1993)
17. Intel: Intel C++ STM compiler prototype edition 2.0. language extensions and user's guide (2008)
18. Kumar, S., et al.: Hybrid transactional memory. In: Proc. ACM PPoPP'06, pp. 209–220 (2006)
19. Lester, N., et al.: Efficient online index construction for text databases. ACM Trans. Database Syst. **33**(3), 1–33 (2008)
20. Minh, C.C., et al.: STAMP: Stanford transactional applications for multi-processing. In: Proc. IISWC (2008)
21. Moore, K., et al.: LogTM: log-based transactional memory. In: Proc. HPCA'06, pp. 254–265 (2006)
22. Ni, Y., et al.: Design and implementation of transactional constructs for C/C++. In: Proc. ACM OOPSLA (2008)
23. Pankratius, V.: Automated usability evaluation of parallel programming constructs (NIER track). In: Proc. ACM ICSE (2012)
24. Pankratius, V., Adl-Tabatabai, A.-R.: A study of transactional memory vs. locks in practice. In: Proc. ACM SPAA (2011)
25. Pankratius, V., et al.: Does transactional memory keep its promises? Results from an empirical study. Technical report, 2009-12, University of Karlsruhe, Germany (2009)
26. Pankratius, V., et al.: OpenMPspy: leveraging quality assurance for parallel software. In: Proc. Euro-Par (2012)
27. Ringenburg, M.F., Grossman, D.: AtomCaml: first-class atomicity via rollback. In: Proc. ACM ICFP (2005)
28. Rossbach, C.J., et al.: Txlinux: using and managing hardware transactional memory in an operating system. In: Proc. ACM SOSP'07, pp. 87–102 (2007)
29. Rossbach, C.J., et al.: Is transactional programming actually easier. In: Proc. ACM PPoPP (2010)
30. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. **14**(2), 131–164 (2009)
31. Saha, B., et al.: McRT-STM: a high performance software transactional memory system for a multi-core runtime. In: Proc. ACM PPoPP'06, pp. 187–197 (2006)
32. Scott, M., et al.: Delaunay triangulation with transactions and barriers. In: Proc. IEEE IISWC (2007)
33. Shavit, N., Touitou, D.: Software transactional memory. Distrib. Comput. **10**(2), 99–116 (1997)
34. Standard Performance Evaluation Corporation: SPEC OpenMP benchmark suite. www.spec.org/omp (2009)
35. TM bibliography. http://www.cs.wisc.edu/trans-memory/biblio/index.html (2011)
36. Watson, I., et al.: A study of a transactional parallel routing algorithm. In: Proc. ACM PACT'07, pp. 388–398 (2007)
37. Welc, A., et al.: Transparently reconciling transactions with locking for Java synchronization. In: Proc. ECOOP (2006)
38. Woo, S.C., et al.: The SPLASH-2 programs: characterization and methodological considerations. In: ACM ISCA (1995)
39. Yin, R.K.: Case Study Research: Design and Methods, 3rd edn. Sage, Thousand Oaks (2002)
40. Zannier, C., et al.: On the success of empirical studies in the international conference on software engineering. In: Proc. ACM ICSE'06, pp. 341–350 (2006)
41. Zyulkyarov, F., et al.: Atomic quake: using transactional memory in an interactive multiplayer game server. In: Proc. ACM PPoPP'09, pp. 25–34 (2009)