Does Transactional Memory Keep Its Promises? Results from an Empirical Study.

Victor Pankratius
University of Karlsruhe
76131 Karlsruhe, Germany
pankratius@ipd.uka.de

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

Frank Otto
University of Karlsruhe
76131 Karlsruhe, Germany
otto@ipd.uka.de

ABSTRACT

Transactional Memory (TM) promises to simplify parallel programming by replacing locks with atomic transactions. This is the first paper to assess the value proposition of TM based on a comparative case study with real programmers. Twelve students, working in teams of two, wrote a parallel desktop search engine in C/C++ during a fifteen week lab. Three randomly chosen study groups (TM teams) competed for the best performance using Intel's Software Transactional Memory compiler and Pthreads, while three control groups (locks teams) competed using just Pthreads.

The study provides empirical evidence that both supports the TM value proposition and at the same time points to problems with TM. The winning TM team's program performed better than that of the winning locks team, and the TM winners were the first to have a prototype parallel search engine, four weeks earlier than the locks winners. Compared to the locks teams, the TM teams spent less than half the time debugging segmentation faults. On the other hand, TM teams had more problems tuning performance because TM performance was hard to predict.

The study also provides insights into general difficulties programmers have with parallel programming. Some insights are technical; for example, some students wrongly assumed that it is safe to read shared variables outside critical sections, resulting in data races in the winning teams' programs. Other insights are non-technical, including psychological ones; for example, the TM teams were less afraid of using parallel constructs, yet two of the teams procrastinated parallelization.

Based on our insights, we elaborate on future research directions. We suggest refinements to TM constructs and tools. We also sketch how to automate similar case studies and propose a methodology for automating the evaluation of new language features for parallelism.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.0 [**Software Engineering**]: General

Technical Report 2009-12
Institute for Program Structures and Data Organization (IPD)
University of Karlsruhe, Germany
September 2, 2009

Keywords

Transactional Memory, Pthreads, concurrency, synchronization, language design, multicore software engineering

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 [12], Fortress [4], Chapel [13], and Clojure [1], all provide transactions in-lieu of locks as the primary concurrency control mechanism. Other research systems have extended existing languages such as C++ [30], Java [3], Haskell [19], and ML [31] 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 of TM versus locks focused on small, mostly numerical programs or microbenchmarks 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 missed to shed light on the issues encountered when parallelizing with TM from scratch [47].

There is no way around systematic empirical studies to validate the TM approach and to give an objective answer to whether TM applies to complex real-world programs (including non-numerical ones) and whether TM performs well enough for such applications. We need to extend the evaluation to a portfolio of advantages and disadvantages that includes software engineering aspects.

In this paper, we present a comparative case study that aims to validate TM's main value proposition to make parallel programming easier. We aim not to study parallel programming experts, but instead learn in great detail from individual graduate-level student programmers tasked with developing a parallel desktop search engine under realistic time pressures. The study randomly assigned twelve graduate students to six teams. Three of the teams had to use locks, while the other three teams had to use TM language constructs provided by the Intel C++ Software Transac-

tional Memory (STM) compiler [30] – one of the most advanced STM compilers built on top of Intel's C++ compiler. The study was organized as a competition so that the locks team and TM team that produced the fastest program on a dual quad-processor machine and implemented the most requested features won.

This paper makes the following contributions: (1) it is the first study of its kind to compare locks to TM programming by observing how several teams wrote a realistic application from scratch over an extended period of time; (2) it systematically collects a combination of quantitative and qualitative data to compare performance, hours spent on various development phases, code metrics, ease of code understanding, as well as subjective or psychological issues during implementation; (3) it shows that TM is indeed a valuable approach for parallel programming; (4) it builds up a chain of evidence to falsify the opinions that TM does not help building real-world parallel applications; (5) it shows that most of the shortcomings of TM are due to the immaturity of the TM tool chain, notably tools for debugging and performance tuning. At the same time the paper shows that TM does not solve all concurrency control problems, and thus is not a silver bullet.

Using this study as an example, we also elaborate on a longer term vision of empirically validated language design that brings programmers and language designers into a loop of empirical studies. Such feedback is essential for making the right tradeoffs in the design of parallel constructs, which can then be defined to avoid common mistakes and reduce the cognitive mismatch between intuition and implementation. Moreover, such an approach reduces the risks of providing the wrong hardware support or maintaining large code bases written in inappropriate languages.

The paper is organized as follows. Section 2 presents an overview of the programming models used in this study. Section 3 describes the case study design. Section 4 shows the experience of each student prior to the study. Section 5 provides an overview of the key case study results. The sections that follow, provide more details: Section 6 discusses performance results; Section 7 summarizes weekly interviews showing how teams made progress; Section 8 complements interview data with insights from a post-project questionnaire; Sections 9 and 10 discuss code metrics and insights from code inspections of the submitted parallel search engines; and Section 11 illustrates the effort in person hours spent by each team on pre-defined categories of tasks. The validity of the results is discussed in Section 12. Section 13 presents related work. Based on the insights of these study, we outline future research directions in Section 14 and provide a conclusion in Section 15.

2. THE PARALLEL PROGRAMMING MODELS IN THIS STUDY

Most programmers today use shared-memory parallel programming techniques to programm 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 [10, 23]) 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 Pthreads - synchronizing with locks

Pthreads is a standardized, platform independent API for parallel programming in C and C++ [10, 23]. 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 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 [17] 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 and atomicity violations. Large transactions can hurt scalability and performance. Programmers may have 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 optimizations can in turn introduce data races in which a variable is accessed concurrently both inside and outside a transaction. They can also introduce atomicity violations, exposing a broken invariant to other threads.

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 paper, we use 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++ [30]. 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's the programmer's responsibility to make sure that such functions can be called inside transactions without instrumentation.

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. CASE STUDY DESIGN

This study aims to compare parallel programming with Transactional Memory to parallel programming with locks for a non-trivial parallel program with realistic features, created over an extended period of time by different programming teams. As subjects, the study uses twelve full-time graduate students in a software engineering lab. The students all had Bachelor-equivalent degrees in computer science and were pursuing Master's degrees in computer science. The study compares the code, performance, errors, and programming effort of a desktop search engine program. In addition, we study how students use the features of the programming models, how they make progress, and which technical or non-technical problems they encounter. By comparing the results, we provide new insights, help assess the real advantages and disadvantages of Transactional Memory, and derive empirically grounded directions for future research.

We followed the recommendations made in [44, 45] in the manner in which we planned the case study collected data, introduced randomization on how subjects are assigned to teams or programming methods, and triangulated data from multiple sources of evidence. This is the first case study on TM to do so in a systematic way.

Two instructors conducted this study as part of a multicore software engineering lab at the University of Karlsruhe, Germany. The lab consisted of a teaching part and a project part. The prerequisites of the lab already filtered out students with inappropriate skills. Twelve students who met the prerequisites and registered for the class were admitted to the study. The students agreed to the usage of their results for research purposes. We assigned a unique number to each student, which we use in the presentation of the data (e.g., experience profiles and questionnaire answers).

3.1 Choice of application

The subjects were given the task of writing a parallel desktop search engine from scratch in C or C++. We chose this application for several reasons: First, the application is useful in everyday life so we can easily describe its functionality and measure its performance on commodity multicore computers. Second, it can take advantage of parallelism in several ways; for example, indexing and querying can execute concurrently to improve responsiveness, and at the same, they can each take advantage of multiple cores to improve their performance. Third, it exemplifies a non-trivial application; dealing with parallel programming on a larger scale and over a longer period of time makes it more likely to reveal existing software engineering problems with parallelism. Finally, in contrast to existing Transactional Memory evaluations (described in Section 13), it does not concentrate entirely on numerical computation.

We conducted a feasibility study prior to the lab to ensure that this application can be parallelized and implemented, given the environment, tools, and amount of time.

3.2 The competition

To simulate a real-world industry scenario, we structured the project as a competition between the teams to build an error-free search engine with the best indexing and query performance. 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. As an award, we promised the winning teams an official certificate of achievement. Section 6 presents the file benchmark, the multicore machine measurement setup, and the performance of the submitted search engines. As later sections show, the competition led to different approaches, data structures, and source codes as the teams had no incentive to collaborate with each other.

Each team had a single-socket four-core Intel machine and shared access to a dual-socket eight-core machine, which was used for benchmarking (Section 6.1). The teams coordinated access for performance tests on the eight-core machine, thus eliminating workload interference.

3.3 Desktop search engine requirements

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 (i.e., not a-z, 0-9, ä, ö, ü, or ß) 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"). It must be allowed to execute one query while indexing is still in progress, but it is not required that more than one query at a time works in parallel. It was up to the students to decide whether to parallelize each query; the number of query threads was not required to be configurable from the command line, but the students 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 for search queries.

Input/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; the sum is 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. The program (or the programs for indexing and search, if implemented separately) should be callable from the command line. For indexing, a parameter must be defined to set the number of indexing threads. This may, but need not be implemented in a similar way for search.

3.4 Schedule

Figure 1 shows the lab schedule. The lab lasted one fifteen week semester during which the whole class met once a week. In the first class session, we randomly created teams of two students to allow students to socialize and get to know each other prior to the project work. We also told the students that they would later program a parallel desktop search engine in C or C++ (without disclosing detailed requirements), and we gave them two survey articles [46, 26] about the basics of and state-of-the-art in text indexing and retrieval.

To make sure every student starts with same baseline knowledge prior to the project, the first three classes taught the theory of parallel programming and the basics of parallel programming in C/C++ using Pthreads and Transactional Memory. All students learned to use Intel's Software Transactional Memory C Compiler v. 2.0. The classes also covered the tools for debugging, profiling, and version control. At the end of each class, the students received pointers to documentation containing additional information.

During the first three weeks of the lab, each student had to

Case study schedule

Teaching part (same for all students, individual assignments)	
Oct 20, 2008	• **Start of lab** • Random creation of teams of two students • Distribution of surveys on text search engine technology • Teaching session 1 (parallel programming, Pthreads) • Distribution of assignment sheet 1 (Pthreads)
Oct 27, 2008	• Discussion of assignment sheet 1 • Teaching session 2 (tools, concepts, demos: programming and debugging with Eclipse CDT, profiling with gprof/kprof, valgrind, callgrind, cachegrind, version control with Subversion) • Distribution of assignment sheet 2 (debugging and profiling)
Nov 03, 2008	• Discussion of assignment sheet 2 • Teaching session 3 (concepts of Transactional Memory, Intel STM Compiler) • Distribution of assignment sheet 3 (Transactional Memory)
Nov 10, 2008	• Discussion of assignment sheet 3 • **Start of project** • Introduction of project, discussion of requirements • Random assignment of teams to use locks or TM

	Practical part (teamwork in a locks team or TM team)	
2	Nov 17, 2008	Meeting and work in class, team interviews
3	Nov 24, 2008	Meeting and work in class, team interviews
4	Dec 01, 2008	Meeting and work in class, team interviews
5	Dec 08, 2008	Meeting and work in class, team interviews
6	Dec 15, 2008	Meeting and work in class, team interviews
7	Dec 22, 2008	Meeting and work in class, team interviews
8	Jan 12, 2009	Meeting and work in class, team interviews
9	Jan 19, 2009	Initial deadline for project submission Meeting and work in class, team interviews
10	Jan 26, 2009	Deadline for project submission; team interviews
	Feb 02, 2009	• **End of project** • Team presentations
	Feb 09, 2009	• Disclosure of winning teams • Final discussions • **End of lab**

Figure 1: Laboratory and project schedule.

solve individually practical exercises in each area. This was an opportunity to teach the necessary C/C++ programming language skills as well. All students passed all assignments, showing that they all reached a comparable level of experience prior to the project. Class discussions intended to bring all students to the same level.

In the fourth week, we introduced the detailed requirements for the search engine (Section 3.3), and we randomly assigned three teams to use only Pthreads and assigned the other three teams to use Pthreads and TM. The TM teams needed Pthreads because Intel's STM compiler did not have native language constructs for thread creation. The TM teams could thus use a superset of language constructs, including locks, semaphores, and condition variables.

The project lasted ten weeks not counting a two week vacation (Christmas and New Year's Eve). Students worked at home and during weekly meetings in class. We extended the initial submission deadline by one week due to requests from all teams. In the last two weeks, all teams presented their approach, and the class discussed the final results. The instructors carried out the performance measurements determining the winning teams.

3.5 Sources of evidence

Throughout this study, we followed the recommendations of [44, 45, 34] and used several sources of qualitative and quantitative evidence:

- Project-related diaries that all teams had to write. The teams used the 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.
- A final report about each team's search engine. Each team had to write a final report that put together the information from their diaries, as well as explaining their architecture, implementation, and performance results.
- Slides of the team presentations after completion of the project.
- A time report sheet capturing effort on a daily basis. All teams logged their hours on this sheet, split according to predefined task categories.
- Notes from the weekly interviews. The instructors kept detailed notes from semi-structured interviews [34] conducted during weekly class meetings.
- A post-project questionnaire, filled out individually by each student.
- The source code produced by each team.
- The subversion repository that all teams were required to use. We used the code revisions, check-in dates, and log messages to study the code development over time.
- Personal observations of instructors.

4. TEAM EXPERIENCE PRIOR TO STUDY

Figure 2 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 2 has less experience than team 3 or team 6. Most teams have Java experience but little C++ experience most likely because of the University curriculum.

Additional data not shown in Figure 2 shows that almost all of the students completed a software engineering course prior to this study, except for student 4 (team 2), and students 7 and 8 (both in team 4). All teams except team 6 had one member with course experience on parallelism. Half of the students took a course on parallelism; in particular, the following students: 1 and 2 (both in team 1), 4 (team 2), 6 (team 3), 7 (team 4), and 9 (team 5).

Students in two of the locks teams (team 1 and 5) and two of the TM teams (team 3 and 6) had industry experience. In team 1, student 1 had one year of experience with the Windows DirectSound library and with programming graphical environments using Motif and Qt under Linux. Both members in team 5 had industry experience: student 9 worked six months in Web development, and her team-mate (student 10) worked for an undisclosed amount of time as a programmer for several companies. Among the TM teams, team 3 had student 5 with five years of support experience for Microsoft server technologies. He also developed an Xbox game on .NET and worked as a freelance consultant for C#. Also in team 3, student 6 worked part-time for seven years as a system administrator and programmer. Finally, both members in team 6 worked for companies: Student 11 worked for five months as a Java programmer and for ten months as a Web programmer. His team-mate (student 12) worked for four years as a software developer.

Looking at each student's experience, one would predict that team 3 would win the competition among the TM teams. Yet this team did not win and actually ended up underestimating the complexities of implementing the queries. They are also the ones that procrastinated parallelization until the very end.

5. KEY CASE STUDY RESULTS

The study shows in the given setting that TM was indeed applicable to a more complex, non-numerical program. TM program performance was good compared to the locks teams, but still an important issue. TM was combined in one case with semaphores for producer-consumer coordination, and in another case 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. The results showed that TM needs better mechanisms for coordination and better handle I/O, and that a combination of TM with locks is promising.

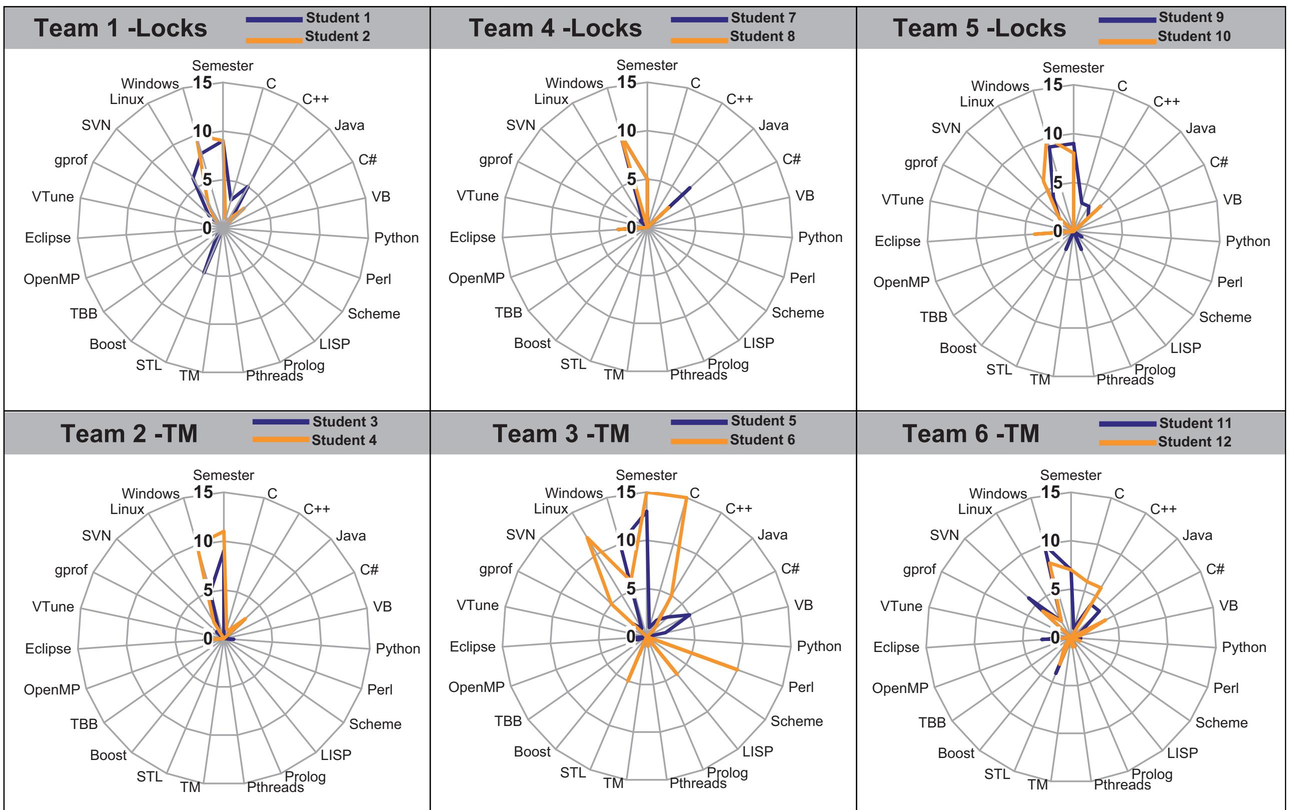


Figure 2: Team experience prior to study. Each chart represents a team, and each line represents years of experience for a student.

The first team to have an acceptably working parallel search engine was the winning TM team at the beginning of the fifth project week; the locks winners had similar functionality in the ninth week. By the end of the project neither winning team had completed all queries. Locks team 1 was the only team to have implemented all queries, but they had the worst indexing performance. Compared to the locks winners, the TM winners had three fewer queries, but were faster on indexing and nine of their queries.

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 Intel compiler experts. Although all teams implemented similar program functionality, all TM teams used 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. Most teams, and in particular the winning 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. The winning TM team 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 to make 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 did not produce a working program; parallel programming remains difficult.

6. PERFORMANCE MEASUREMENTS

As shown in Figures 3 and 4, our indexing and query performance measurements of the submitted search engines contrast the literature that overgeneralizes that Software Transactional Memory performs poorly [11]. The team with the best indexing performance was a TM team. For nine out of eighteen queries, the best results were achieved by a TM team.

Team 6 (TM) had the best indexing performance of all teams, completing our benchmark in 178 seconds. Compared to the fastest locks team on indexing (team 4) that finished in 532 seconds, TM was three times faster. For 9 out of 18 types of queries, TM teams had the best execution times; they were 13%–95% lower than the fastest locks team. The locks teams outperformed the TM teams on the other queries by orders of magnitude.

Except for the most inexperienced team 2 (TM), all teams had executable parallel programs at the final deadline. No search engine was perfect, however, as all implementations had either missing or incorrectly working queries (which were not counted in the competition). The number of working queries was the only significant difference in feature completeness, and we use it as a metric for comparison. The

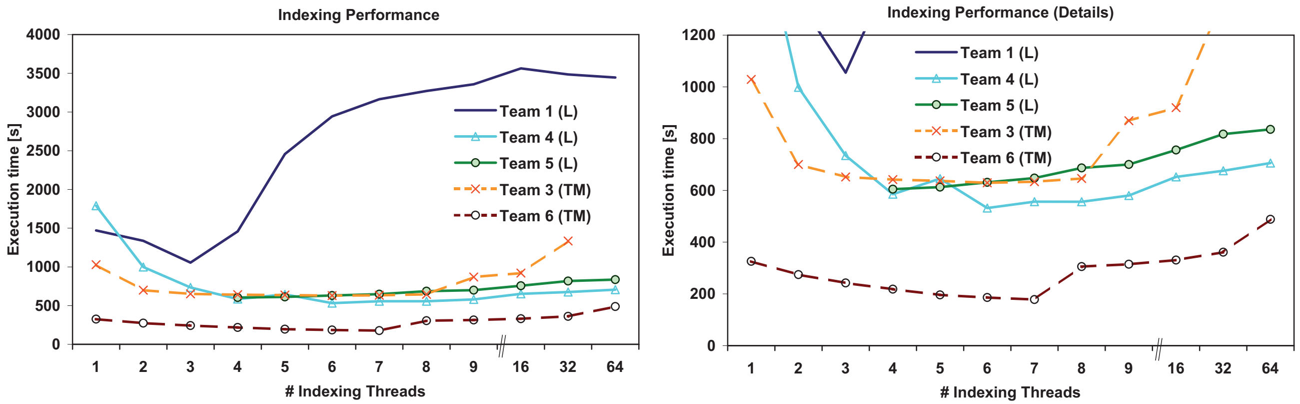


Figure 3: Indexing performance depending on the number of indexing threads.

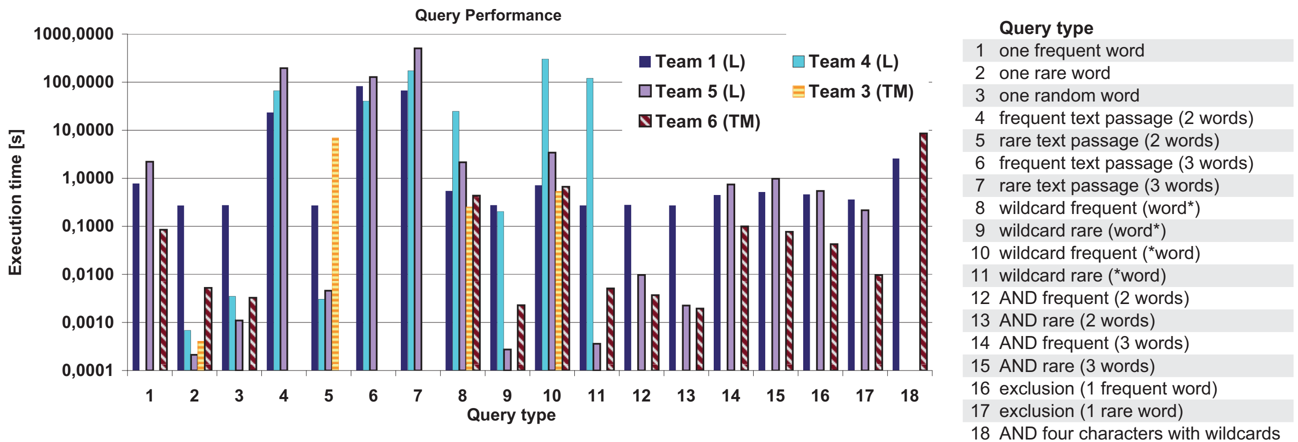


Figure 4: Query performance for different types of queries, excluding queries that were not implemented or that produced incorrect results.

locks teams had more feature complete parallel programs: out of 18 queries, they implemented 18 (team 1), 17 (team 5), 10 (team 4), while TM teams implemented 14 (team 6), 4 (team 3), and 0 (team 2).

6.1 Measurement methodology

Performance measurements were done 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. All source codes were compiled with Intel's C compiler, using STM extensions for Transactional Memory 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. In the following graphs, each point represents the average of five measurements. Only results of correctly working features are shown. To ensure comparability, all measurements were carried out by instructors.

Figure 5 shows the input file set used to benchmark performance in the competition. The benchmark consists of directories containing a diverse selection of ASCII text files. It includes the Calgary Text Compression Corpus (which is used to evaluate compression programs [42]), one big text file, four larger files, and many small files.

The program of team 2 had memory consumption problems and did not work with the specified file benchmark. The team was too inexperienced to fix the relevant before the deadline. Consequently, they were excluded from the competition.

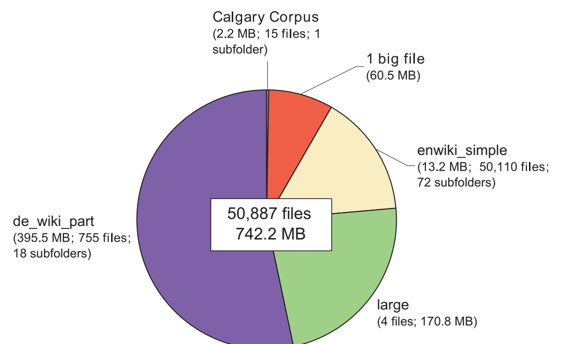


Figure 5: Benchmark used to evaluate the desktop search engines.

6.2 Indexing performance

Figure 3 shows the execution times for indexing; the measurements for the last three numbers of indexing threads on the x axis have a logarithmic scale in order to show the general behavior of execution times for larger numbers of threads. All teams provided a configurable command line parameter to specify the number of indexing threads, and only this parameter was varied when measuring performance. Some teams also had other command line parameters, for which they provided fixed values they thought were best. No queries were executed during performance measurements for indexing, and the machine was used exclusively.

Team 1 (locks) has the worst indexing performance. The best execution time of 1055s is reached at 3 threads. Indexing performance does not scale well with more threads. As the variance in execution time skews the graphs of the other teams, we present a more detailed chart on the right of Figure 3.

Team 6 (TM) achieves the best performance. Although their performance appears to be flat on the left of Figure 3, a typical bath tub form is revealed on the right. Already with single thread execution, their program is the fastest at 326s. Their execution time continues to improve with additional threads until 7 threads (178s), which is the point at which the total number of indexer threads plus the one crawler thread equals the total number of cores. This is the kind of scalable behavior that is generally desirable in parallel programming. At seven threads, they are faster by a factor of 1.8 over a single thread. We remark that reducing the execution time of a fast program can be more difficult than reducing the execution time of a very slow program (such as that of team 1) due to the impact of additional parallel overhead that has to pay off.

The performance graph of team 3 (TM) has a similar shape as that of team 6. However, it is shifted towards a less favorable execution time. The program did not seem to work with 64 indexer threads. This team achieves the best execution time of 629s with 6 threads.

Team 5's (locks) program did not work with less than 4 threads. The best execution time of 605s is reached with 4 threads, but using more threads harms performance. The performance of team 5's program therefore does not scale with an increasing number of threads.

Team 4's (locks) program has the best indexing performance of all locks teams, achieving the best time of 532s with 6 threads. This team is the only locks team whose indexing performance scales with the number of threads. As we see in later code inspections, this team used over 1600 locks.

We make an important observation for indexing performance: the best locks team (4) is just 97s faster than the worst TM team (3), and the difference between the best locks team and best TM team (6) is 354s.

6.3 Query performance

Figure 4 compares the execution times of 18 different queries, each testing a particular type of query. Query threads are not configurable from the command line because of the underlying strategies, which differ widely; for example, some teams did not parallelize queries while others derived the number of threads from the number of words in a query. We only show results for queries that were implemented and produced correct results. Out of 18 queries this is true for the following number of queries: team 1 (18), team 5 (17), team 6 (14), team 4 (10), team 3(4), team 2 (0).

Team 1 (locks) is the only team to have all queries working, but often with bad performance. As will be explained later, they spawn too many threads and have bottlenecks due to locks. Team 4 (locks) has the worst performance of all teams for wildcard queries. Team 5 (locks) has the worst query result of all teams for rare text passage search with three words, but at the same time the best performance of all teams for searching one rare word.

Team 6 (TM) has 14 queries working, often with better performance than some of the locks teams. Team 3 (TM) has just 4. They reported in an interview that they underestimated the complexity to implement queries and ran out of time; however, in three cases they are faster than team 6 and faster than most locks teams.

6.4 Team rankings

The competition aimed to motivate the teams. At the start of the competition, the teams knew their ranking criteria and had the input data set. The ranking criteria used a scoring model that assigned an equal weight to indexing and querying performance. We did this to make sure that the teams optimized the performance of both indexing and querying, rather than optimizing one feature at the expense of the other. The ranking also excludes results from incorrectly implemented queries. Team 2 is excluded because the program did not run on the benchmark. We use the inverse of execution times to obtain a score in which higher numbers are better.

Ranks for indexing: We multiply the inverse of the minimum execution time of every team (according to the graph in Figure 3) by 0.5. The team with the highest score is ranked first.

Indexing	**Locks**			**TM**		
Rank	1	2	3	1	2	3
Team	4	5	1	6	3	–

Ranks for queries: For all implemented queries that produced correct results, we multiply the inverse of their execution times by 0.5/18 and sum up. The team with the highest score is ranked first.

Queries	**Locks**			**TM**		
Rank	1	2	3	1	2	3
Team	5	4	1	3	6	–

Final ranks: Because of differing orders of magnitude, we multiply the scores of indexing and query to obtain the final score. This approach intentionally favors a team that might have fewer but faster queries than other teams.

Final ranks	**Locks**			**TM**		
Rank	1	2	3	1	2	3
Team	5	4	1	6	3	–

Who won the competition?.

According to the final ranking, team 5 (locks) and team 6 (TM) won the competition.

7. INTERVIEW RESULTS

During the weekly class meetings, the instructors interviewed each team for fifteen minutes. Tables 1 and 2 summarize the interviews starting on December 1, the fourth week of the project. We omit irrelevant information from earlier weeks in which no significant progress was made and mainly focus on parallelism issues.

These interviews allow us to track the progress of each team and give us insights into what the students perceived as the most important issues. The interviews were semi-structured [34], asking open-ended questions about current status, problems, and plans, without requiring any particular format in the response. The students were free to present any issues they felt were relevant. The instructors were careful not to influence design decisions or plans; for example, we avoided giving feedback or judging their comments, instead encouraging students to experiment and find out by themselves. In the last two class meetings, after the project ended, the class as a whole openly discussed and analyzed the different approaches.

Team 6, the TM winners, were the first team to demonstrate parallel indexing and querying on the benchmark data. In contrast, team 5 (the winning locks team) demonstrated similar functionality four weeks later. The locks teams began parallelization earlier than the TM teams; two of the TM teams delayed parallelization and instead focused on sequential programming.

During the interviews the teams also described what they read. Team 4 (locks) read library-related documentations as they intended to reuse code. Team 5 (locks) researched the literature to find an appropriate index data structure and studied Burst Tries [20]. All TM teams read the Transactional Memory related documentation of Intel's STM compiler [24]. Team 6 read survey papers [46, 26] and studied chapters from an information retrieval book [8]. Team 3 read library-related documentations for a B-tree library, but they eventually decided to build their own.

7.1 Interviews with locks teams

In the fourth week, team 4 was the first among all the teams to elaborate their thoughts on parallelization. By contrast, the parallelization strategies of team 1 were more vague, but they started smaller experiments in the fifth week to find out how to place locks in the indexing data structure. Team 5 did not have a parallel indexer until the sixth week; they spent a lot of time discussing the choice of data structures.

In the eighth week, one week before the original deadline, all locks teams had parallel implementations, but none of them could show a full demonstration. Team 1 did not have a file crawler, team 4 was debugging unexpected behaviors in their program, and team 5 had not tried out their program on the benchmark data. All teams had problems with performance, missing features, or bugs, which they tried to address as much as possible, but ran out of time before the original deadline of week nine.

In the ninth week, the original deadline of the project, all teams had running search engines, but two of them appeared experimental: team 1's program was unstable, and team 4's had segmentation faults. Team 5 focused on performance testing, but in the following week they found a bug.

By the end of the project in the tenth week (January 26), team 1 had run out of time and skipped performance tests, team 4 was not finished with performance tests, and team 5 had discovered a concurrency bug that they were trying to fix before submission.

7.2 Interviews with Transactional Memory teams

By the fourth week, team 6 was only team to have thought about parallelization. In the beginning of the fifth week, team 6 was the first among all six teams in the study to show an advanced demo of a parallel search engine. Having attacked parallelization early, they had also overcome their initial difficulties on how and where to apply TM constructs. In contrast, in the fifth week, team 2 and team 3 had no clear ideas on how to parallelize.

In the eighth week, one week before the original deadline, team 2 had just a serial program, team 3 had an incomplete parallel indexer and no queries working, while team 6 had a full-fledged working demo.

In the ninth week, the original deadline of the project, team 2 was still incomplete, team 3 had a running, but buggy parallel program with bad performance, and team 6 fixed many bugs in their search engine.

By the end of the project in the tenth week (January 26), team 2's program failed on the final benchmark, team 3 had parallel indexing and queries working with reasonable performance, and team 6 had an even more improved search engine.

Team 2 and team 3 procrastinated parallelization due to various reasons. The main reason for team 2 was their lack of experience. Both students were hesitant and insecure, especially during implementation. Team 3 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 tools to simplify these tasks.

8. QUESTIONNAIRE RESULTS

Each student individually answered a questionnaire after the end of the project. The questionnaire captures in a structured way student ratings on different aspects of working strategy, design, implementation, testing and debugging, and the usage of parallel constructs. Figures 19 and 20 in Appendix C show the questions and answers.

We interpret the response of each student individually rather than a roll-up of the responses, as there is not enough data to make conclusions from aggregates. This allows us to put each student response in the context of other data from that same student. We highlight only the data that clearly indicates a pattern.

8.1 Work strategy

Looking at Figure 19 (a), all students understood the assignment and rated it easy to understand with average complexity (i.e., neither easy nor complex). All students found it difficult to create a working plan at the beginning of the project, except for TM team 3 and one student in locks

Day (proj. week)	**Locks teams interview results**
Dec 1 (4)	• **Team 1:** The team discussed the index design and the placement of locks, but did not have any code running yet. • **Team 4:** 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. • **Team 5:** The team did not have a running program yet. The team discussed indexing strategies and data structures choices, but had no code running.
Dec 8 (5)	• **Team 1:** 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. • **Team 4:** The team implemented a rudimentary parallel indexer. • **Team 5:** 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 competition benchmark.
Dec 15 (6)	• **Team 1:** 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 because the responsible team-mate did not finish. The team was evaluating the index structure based on the prototype. • **Team 4:** Parallel indexing worked. Queries could be executed while indexing was in progress. • **Team 5:** Parallel indexing worked. Queries were implemented in a rudimentary way.
Dec 22 (7)	• **Team 1:** Parallel indexing and parallel queries still worked with the simulated file crawler. They were working on query result ranking but were not finished yet. • **Team 4:** Both students mentioned that they were used to Java programming in the past. They were missing comparable approaches in C, such as code structuring with packages or having classes in independent files. They had a lot of code in their header files, not in the corresponding .c files. Despite knowing how to do it right, they didn't make any attempts to change their code. • **Team 5:** The team showed how they used the Linux system monitor for performance testing and debugging. There was not much other progress to see.
Jan 12 (8)	• **Team 1:** The team had finished all components except the file crawler, but they hadn't tested it yet on the real benchmark, because the team member responsible for the file crawler hadn't finished. • **Team 4:** Although parallel indexing and search seemed to had worked in the past, the team suddenly found out that they had problems compiling their code on other machines. • **Team 5:** Parallel indexing and parallel search worked, but the team did performance tests only on a subset of the competition benchmark.
Jan 19 (9)	• **Team 1:** The team finished implementing the file crawler, and parallel indexing and parallel queries worked. Their version was sometimes unstable, and needed more testing and debugging. • **Team 4:** Parallel indexing parallel queries worked. The team fixed segmentation faults and did performance tests. • **Team 5:** Parallel indexing and queries worked. The team continued with performance testing.
Jan 26 (10)	• **Team 1:** The team still had not tested performance on the given benchmark. • **Team 4:** The team was about to test performance on the given benchmark. • **Team 5:** The team was about to fix a bug with the file statistics update of their indexer.

Table 1: Summary of locks teams interviews.

team 1, who said it was easy. In every team, one student replied to have done most of the work, while his team mate said the opposite; these responses matched the instructor's perception.

In all teams, one student perceived the planning horizon differently than the team mate: In all of the locks teams the student that did most of the work felt that decisions were made spontaneously, without planning ahead. This is also true for TM team 3. All of the students in the other TM teams planned one to seven days ahead. All teams perceived the same time pressure except during the testing phase, in which the TM teams felt more pressure than the locks teams.

8.2 Design

Looking at Figure 19 (b), all teams thought that in the design process it was not difficult to identify the places for parallelization. Students who did most of the work in the winning locks and winning TM team thought it was unimportant to know during the design phase whether they would use locks or TM in the implementation, whereas their team mates thought it was very important. Interestingly, both members of team 2, the most inexperienced team in the study, felt that this information was very important.

8.3 Implementation

Looking at Figure 19 (c), all teams felt it was important to get to the first executable parallel program quickly. Team 6, the TM winners, were the only team in which both students agreed that it was very important to get quickly to the first executable parallel program. This matches the way in which they actually implemented features. Similarly for team 4, the locks team that started earliest on parallelization. Team 2 also agreed it was important to get to the first executable parallel program quickly, yet they procrastinated parallelization. Interestingly, team 5 (the winning locks team) ranked the importance of early parallelization lower than the other teams, consistent with the interview results showing that early parallelization was not their priority.

The team mates who did less of the work in the winning teams (locks and TM) were the only ones to report using parallel constructs from the first lines of code; all other students

Day (proj. week)	**TM teams interview results**
Dec 1 (4)	• **Team 2:** The team discussed design alternatives. • **Team 3:** The team was about to implement their index data structure and planned to have an executable version in the next 1–2 weeks. • **Team 6:** A rudimentary indexer worked, but they did no performance tests. They had problems understanding TM constructs and how to apply them in their code. The design of their search engine was not yet clear, but it seemed they had worked heavily on their piece of paper.
Dec 8 (5)	• **Team 2:** The team was about to test a first sequential indexer. So far, they had not thought about how to make it work in parallel or how to use TM. • **Team 3:** The team had two sequential modules of their program but no thoughts so far on how to parallelize with TM. The main work was on index reading and writing operations. The team planned to have a first parallel version at the end of December (eventually, they missed this deadline). • **Team 6:** Parallel indexing worked in an acceptable way and the team was already doing performance tests on the final benchmark. This was the first team of all to achieve this level in the fifth week of the project. Problems with segmentation faults appeared. As no appropriate thread-safe libraries were available, they decided to implement a lot of low-level functions by themselves.
Dec 15 (6)	• **Team 2:** 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. • **Team 3:** The team's entire code was still sequential. Neither of them had thought of parallelism or transactions yet. • **Team 6:** The team's parallel indexing worked. A rudimentary query could execute while indexing was in progress.
Dec 22 (7)	• **Team 2:** The team had made some unsuccessful parallelization attempts. The team had problems interpreting compiler errors, due to their lack of experience. They tested their program with just one of the files from the benchmark. They had memory leaks they couldn't find. • **Team 3:** 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 (see Figure 4). The STM compiler crashed due to a known bug when statistics were turned on. • **Team 6:** The team finished implementing their thread-safe library functions. Both students mentioned that they occasionally forgot to enclose code by atomic blocks, but that they fixed these errors.
Jan 12 (8)	• **Team 2:** Indexing worked just in serial mode. The team had procrastinated much of the parallelization work. The few parallelization attempts were superficial. They got compiler warnings that several functions did not have the *tm_callable* attribute. The memory leak was still there. Only one word could be used in a query. • **Team 3:** The team had not yet finished parallel indexing. No performance tests had yet been done. Queries did not work yet. • **Team 6:** 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. The students show that they used advanced tuning concepts by deriving performance-relevant parameters from the number of indexing threads.
Jan 19 (9)	• **Team 2:** The team reported that parallel indexing and queries were almost finished. Queries allowed just the inclusion or exclusion of one word, however. A segmentation fault was fixed. • **Team 3:** 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. • **Team 6:** The team fixed a segmentation fault and many bugs.
Jan 26 (10)	• **Team 2:** The team's indexing did not work for the competition benchmark, due to the memory leak they did not fix. Turning on compiler optimizations caused segmentation faults, which was a bug in the compiler. • **Team 3:** 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. • **Team 6:** 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: Summary of interviews with TM teams.

report employing parallel constructs no later than after implementing a few features. The predominant implementation strategy was to first implement a few features and then add parallel constructs.

Except for the inexperienced team 2, who had to rethink their design once, the TM teams did not have to rethink their design during implementation. By contrast, the winning locks team (team 5), had to rethink their design between one and three times.

The students who did most of the work in TM teams 3 and 6 perceived that in comparison to the performance of the first executable parallel program (which performed as expected) the performance of the final parallel program was below expectations. This reflects the problems of understanding the performance of atomic blocks and tuning TM programs, which they reported in their interviews, and

points to the need for better performance analysis tools for TM.

According to the students, tools were hardly used. All students used the Eclipse [15] environment. In addition, TM teams's students 4 and 5 used the memcheck program from the valgrind tool suite [40]; student 6 used graphviz [16], and student 12 just made an unspecific remark to having used profilers. The locks team's students 9 and 10 used the Linux system monitor for performance tuning.

8.3.1 Psychological issues

Figure 19 (d) shows the questions aimed to uncover psychological problems of parallel programming. In particular, the questions address cognitive issues related to program understanding of parallel programs, fear of applying parallel constructs, procrastination of parallelization work, and backtracking to earlier stages of the project.

TM teams found it more difficult than locks teams to keep track what their parallel program was doing. This may be due to the lack of TM debugging tools and shows that programmers indeed need additional aid for TM program understanding and debugging.

Team 6, the TM winners, never felt afraid of destroying a working version of their parallel program using parallel constructs. Except for student 5 in TM team 3 who reported not doing most of the work in his team, all students in the TM teams were not afraid of destroying their program by using additional parallel constructs. By contrast, the locks teams had a wider range of replies, implying that they were more afraid than the TM teams. The two students (2 and 5) that were most afraid of destroying their programs by using additional parallel constructs were the ones who did not do most of the work.

Team 5, the winning locks team, said they never tried to postpone parallelization work, though they were not the first locks team to have a parallel program; by contrast, the winning TM team tried to postpone parallelization work more often than all locks teams, yet were the first to have a working parallel prototype.

TM team 2, one of the most inexperienced teams, was the only team whose members both report that they had to backtrack 2–3 times to earlier stages of their project and start over.

8.4 Testing and debugging

Figure 20 (a) presents the questionnaire results on testing and debugging. From this figure, we highlight only the data that clearly indicates a pattern.

The TM teams report more frequently than the locks teams running into errors that they could not diagnose; this matches earlier interview observations. TM teams report experiencing fewer deadlocks than the locks teams. The TM teams in general report difficulty understanding the ordering of parallel events in their program. Both members of TM team 3, one of the most experienced teams in the study, said it was very difficult to understand such ordering during debugging, yet they do not report running into any significant ordering errors. This is more evidence that we need tools for TM program understanding and debugging. Compared to the locks teams, the TM teams felt that they had many segmentation faults and unexplainable crashes. Later objective data in Section 11 shows that the TM teams spent less effort than the locks teams on fixing segmentation faults. The interviews show that at least one of the TM teams (team 3) ran into a known STM compiler bug that caused segmentation faults; it's hard to say whether all segmentation faults are due to the compiler bug or due to other bugs in their programs.

8.5 Parallel constructs

Figure 20 (c) shows that only TM team 3 used the *tm_pure* function annotation. Later code inspections show that one usage of *tm_pure* was for a declaration of *printf* so that they could debug the program. Yet more evidence that we need better debugging tools for TM.

Student 4, who did the most of the work in team 2, is the only student that reports using *tm_abort*. This matches later code metrics; however, later code inspections show that this team used *tm_abort* not to recover from error as it was intended to be used but rather in a misguided attempt to optimize performance.

9. CODE METRICS

On average, all teams produced about the same amount of code with comparable productivity. The metrics 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 10 reinforce this observation. Although locks and TM teams programmed parallel search engines with similar functionality, TM teams used fewer critical sections that often had fewer lines of code than the critical sections of locks teams.

Only TM team 3 used the *tm_pure* construct, and only TM team 2 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 implement a racy double-checking pattern [6].

9.1 Code size and productivity

Figure 6 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.

In contrast to the standard deviation of 137 LOC for locks teams, TM teams have a higher standard deviation of 780 LOC which can be explained by the fact that team 2 had incomplete code (1551 LOC less than the winning TM team 6) that did not work on the final benchmark. On the other hand, team 6 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.

We put programming effort in relation to lines of code to calculate the parallel programming productivity of each team. Such data is rare for parallel programs, especially for those written from scratch. In this study, the average productivity of locks teams and TM teams is about the same, which is 0.1 hours (i.e., 6 minutes) to produce one line of code. The TM winners are below average in person hours per line of code; therefore, they are more productive than average. The TM winners were more productive than the locks winners; TM team 6 needed 0.05 hours/LOC (i.e., 3 minutes/LOC), while locks team 5 needed 0.1 hours/LOC (i.e., 6 minutes/LOC). Another interesting observation is that locks team 1 had a productivity of 0.07 hours/LOC

	Locks teams			TM teams		
	Team 1	**Team 4**	**Team 5**	**Team 2**	**Team 3**	**Team 6**
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

Figure 6: Code metrics for the parallel desktop search engines of all teams.

(i.e., 4 minutes/LOC) that was closest to the TM winners, yet team 1 had the worst performance of all teams and team 6 the best.

9.2 Use of parallel constructs

Locks and TM teams clearly differ in how many lines of code contain parallel constructs. Between 5% and 11% of the locks team code had parallel constructs (179 LOC on average). By contrast, between 2% and 5% of TM team code had parallel constructs (83 LOC on average). The locks winners' code had the least usage parallel constructs from all locks teams (120 LOC; 5%), and the TM winner's code the highest of all TM teams (151 LOC; 5%).

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 10, team 3 used one lock to protect a large critical section containing I/O, and team 6 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 that this nesting was not really necessary for the TM teams. Two of the locks teams (teams 1 and 5) had one lexically nested critical section. TM team 3 had two location with nested atomic blocks, and TM team 6 had one.

The special-purpose TM constructs offered by Intel's compiler were used very differently. Team 3 used the *tm_callable* annotation in 2 lines of code, but team 6 used it in 115 lines. Team 3 were the only team that used the *tm_pure* annotation. Only TM team 2 used the *tm_abort* construct, but later code inspections show that they used it mostly in a misguided attempt to optimize performance.

9.3 Critical sections

The critical sections differ for locks teams and TM teams. We statically approximated a lower bound of the length of critical sections by manually counting the LOC enclosed by lock/unlock operations, semaphore wait/post operations, or *atomic* blocks, and excluding comments and blank lines. Information from code inspections will explain 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.

The average size of a lock-protected critical section varies for locks teams between 3.1 and 9.2 LOC, and between 3.5 and 6.4 LOC for the atomic blocks of TM teams. Critical sections of locks teams contain between 56 and 119 function calls, compared to a range of 5 to 41 function calls for TM teams. As an exception, TM team 3 has one lock-protected critical section of 85 LOC, and TM team 6 has several semaphore-protected sections ranging between 3 and 6 LOC.

Figure 7 shows manually collected details on how many critical sections each team had and the cumulative lines of code. We see, for example, that team 4 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.

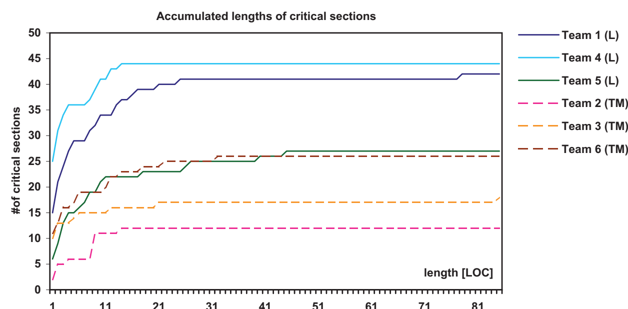


Figure 7: Code distribution in critical sections.

Most critical sections are short. TM teams have fewer critical sections than locks teams. The locks teams have many critical sections with just one line of code, which could have been easily expressed as atomic blocks. Locks teams 1 and 4 have more critical sections than teams 3, 5, and 6, although all teams implement similar functionality. Interestingly, the cumulative critical section size distributions of team 5 (the locks winners) and team 6 (the TM winners) are quite similar.

10. CODE INSPECTIONS

Code inspection allows us to analyze the use of constructs, the kinds of parallel programming mistakes, and code and bug patterns. The first two authors and the STM compiler developers at Intel inspected each team's code in detail, but in in anonymized form. For each team, we summarize important aspects of the architecture, major data structures, synchronization, ease of code understanding, and problems.

10.1 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 4's program the crawler thread generated work at the granularity of files, while in team 1's and 5'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 5 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 4 used 1600 locks, team 5 used 80 locks, and team 1 used 54 locks. Team 4'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 4 and 5 have cases where unlocking is performed in both *then* or *else* statements due to a function return from the middle of a critical section.

Some locks teams used the high number of locks to compensate their insecurity when writing complex parallel programs. Team 4, for example, emulated the Java synchronized constructs. 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, thus they had races. Only team 1 had critical sections protecting a single shared variable read.

10.1.1 Code of team 1

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 1 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 this 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 x 2) locks protecting the index structure. The number of dynamic lock instances is therefore greater than in Figure 6. 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 (Figure 3). Their indexing performance gets worse as the number of worker threads increases beyond 3 threads. Team 1 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 1'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.

10.1.2 Code of team 4

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. 3) 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 4'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 500KB.

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 1000KB 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 = "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.

Like the prior team, team 4 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 4 has over 1600 locks. Similar to team 1, the number of dynamic lock instances is greater than in Figure 6. 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 [22], such as those found in Java. As seen in Figure 2, this team indeed has a Java background and no C++ background. This programming pattern reflects Team 4's Java background.

In Figure 6 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 4 used condition variables for producer-consumer synchronization of their queues.

Team 4'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. In about 500 lines of code, they use the "std::" prefix instead of using name spaces.

10.1.3 Code of team 5

Architecture and data structures: Team 5 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 5 uses condition variables for producer-consumer synchronization of the work queues.

Compared to teams 1 and 4, 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 [20], which is a more complex tree-based data structure than the map-based data structures of teams 1 and 4. Part of their time spent on reading was reading [20]. In addition, like team 1, they have a reverse index (also a BurstTrie) to speed up queries with wildcards at the beginning.

Team 5 also designed queries to work in parallel and to work in 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 5 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.

10.2 Code inspections for Transactional Memory teams

Like some of the locks teams, TM teams 2 and 6 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 2, none of the TM teams parallelized queries. Unlike all of the teams, TM team 3 uses a persistent index on disk and runs queries as a separate program that reads the on-disk index.

The code inspections show that realistic TM programs need to perform producer-consumer synchronization. Team 6 used a semaphore; team 3 avoided producer-consumer synchronization because each indexing thread performed part of the crawling. Team 2 did not consider producer-consumer synchronization because an indexer thread exits once it de-

tects 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 3 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 believed that unprotected reading of shared state is safe. Most teams systematically tried to optimize transactions by first checking a condition outside a transactions and then checking it inside, similar to the flawed double-checked locking pattern [6].

10.2.1 Code of team 2

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 threads 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, Figure 5), but rather on two small sets of files (21MB with 8000 files, 120MB 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 2'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 2 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.

10.2.2 Code of team 3

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 [26]. 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 3 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.

10.2.3 Code of team 6

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 [8], 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 trie 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 [6]. 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 (STM) 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.

11. PROGRAMMING EFFORT

This section analyzes the time spent by each team on different tasks during the project. We compare team effort on different tasks, and we illustrate the hours spent on a certain task category over time. We discuss the most significant patterns based on individual team comparisons and draw conclusions given the context of the other data we have. In general, the effort data backs up previous interview and questionnaire results. Figure 9 and Figures 11 – 18 of the Appendix present the complete effort data.

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 8 shows the categories of tasks that were tracked, and Figure 9 presents the data in terms of person-hours spent by each team per category. Categories 1, 2, and 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 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; for example, team 6 wrote a script to analyze the size of the SVN repository, and this effort was logged in this category.

Overall, the TM teams spent less total effort than the locks teams. In particular, TM teams spent 28 hours less on

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

Figure 8: Effort categories for which each team logged their hours.

reading documentation, 80 hours less on implementation, and 14 hours less on debugging than the locks teams. Comparing effort values at both ends of the range, results look favorable for TM. Out of all teams, the winner TM team 6 spent the least total effort on the project, spending 10 hours less than locks team 1, who spent the least effort of locks teams and had the worst performance of all teams. TM team 3, who spent the most effort out of the TM teams, spent 73 hours less effort than locks team 4, who spent the most effort out of the locks team. The winning TM team 6 needed 67 hours less and had better performance than the winning locks team 5.

11.1 Parallelization

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. Compared to the locks teams, TM teams 3 and 6 (who had working search engines) spent a higher percentage of their implementation time on writing sequential code. In particular, the winning TM team spent 42 hours (76%) of implementation time on writing sequential code, whereas the locks winners spent 35 hours (49%). Being inexperienced, TM team 2 was the only team to spend 8 more hours (11% of implementation time) on implementing parallel code than on implementing sequential code. According to Figure 14, the TM teams spent in total about half as much time as the locks teams on writing 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 1, 1 day; team 4, 13 days; and team 5, 19 days. By contrast, TM teams have larger time lags: team 6, 19 days; team 3, 23 days; and team 2, 29 days.

The effort data generally backs up several of our observations related to parallelization from the interviews. First, the larger time lags for TM teams 2 and 3 back up our observations that these teams procrastinated parallelization. Second, Figures 14–4.1 and 14–4.2 back up our observation that TM team 6, who were also the first to have a working parallel version, started parallelization after locks teams 1 and 4. Figure 14–4.2 reveals that all teams did not parallelize from the first line of code. The locks teams 1 and 4 were the first to start parallelizing, whereas the TM teams 2 and 3 were the last to start parallelization.

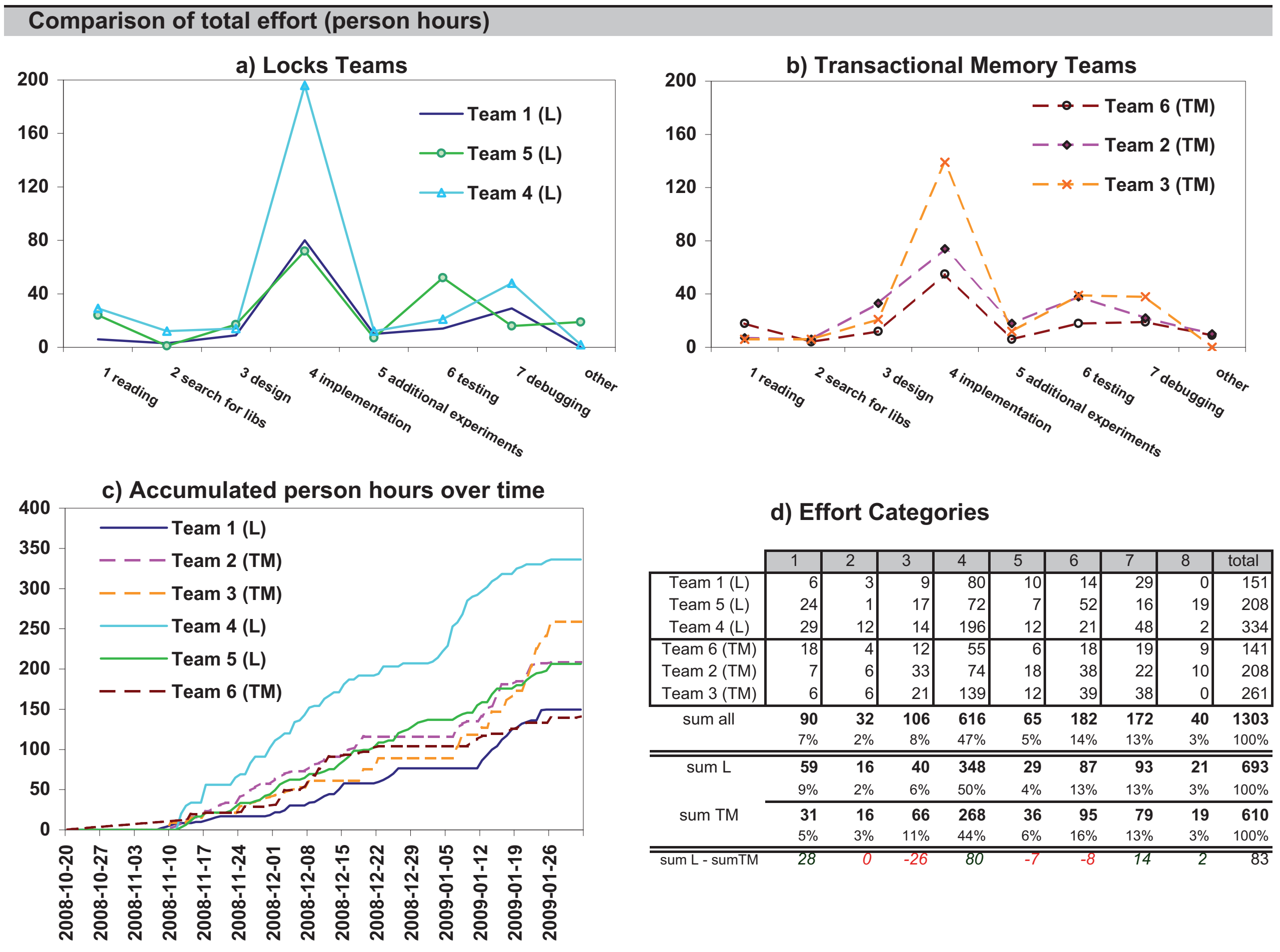
11.2 Performance tuning

The collected data on refactoring, performance experiments on additional programs, and performance tests on the search engine shows that TM teams had more problems with performance tuning than the locks teams, supporting earlier observations that the performance of TM is difficult to understand and predict. The data also suggests, however, that getting good performance using locks requires more refactoring effort than TM.

Late into the project, the TM teams had to experiment with performance and restructure their programs to deal with performance problems. Figure 14-4.3 shows that the refactoring effort increased for all TM teams by the end of the project. Team 3 mentioned during the interviews that they had to split up large transactions into smaller ones, pointing to a late restructuring problem for TM programs. This team had a sharp increase of 14 hours spent on performance testing of their search engine in the last weeks of the project, as shown in Figure 16-6.2. 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, as shown in Figure 15-5.3.

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.

The effort data suggests that getting good performance using locks requires more refactoring effort than TM. The two locks teams (5 and 4) that had good performance spent



	1	2	3	4	5	6	7	8	total
Team 1 (L)	6	3	9	80	10	14	29	0	151
Team 5 (L)	24	1	17	72	7	52	16	19	208
Team 4 (L)	29	12	14	196	12	21	48	2	334
Team 6 (TM)	18	4	12	55	6	18	19	9	141
Team 2 (TM)	7	6	33	74	18	38	22	10	208
Team 3 (TM)	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 9: Total effort of all teams in person hours.

the most absolute time of all teams on refactoring (27 and 22 hours, respectively). Although the winning locks team 5 spent the least amount of time (5 hours) of all teams on parallel code, they also spent the largest amount of time on refactoring, supporting questionnaire data in which they report the highest number of times they had to fundamentally rethink their design. Out of the locks teams, team 1 was the only team to also have a sharp increase in performance testing effort by the end of the project, but this was due to the late delivery of the file crawler by one of the team members.

11.3 Debugging

According to Figure 17, 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. 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. Locks team 5 and TM team 3 spent the least effort (4 hours) on debugging segmentation faults. According to Figure 6, team 5 had the lowest number of lines of code with parallel constructs among the locks teams (120 LOC; 5% of the code); similarly, team 3 had the lowest number of lines of code with parallel constructs among the TM teams (45 LOC; 2% of the code). By contrast, locks team 4 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 4 had the most extensive usage of condition variables, and team 5 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.

TM team 3 was the only team to report a significant amount of time (16 hours; 24% of debugging time) spent on debugging due to problems with TM constructs. This can be traced back to their interviews and their final report, in which they mention running into a bug in the STM compiler (crashes when statistics are turned on). This teams also complained that turning compiler optimizations on caused the compiler to crash.

12. CASE STUDY VALIDITY

In general, a case study aims to provide detailed insight of a case being studied in a real environment, has a broad scope, and is particularly useful in areas where empirical results are scarce [34]. Our case study helps explore and find out what the important issues are when programmers use TM or locks in the realistic environment of a larger project. We collect data to explain why or how something was observed.

In this study, we aimed to understand how well TM applies to parallel programming compared to locks. To model a realistic software development environment, we did not mandate specific data structures, algorithms, or tools. This lead to a diverse set of solutions, which is good because it shows a wide range of possibilities of what works and what does not. Our observations, however, apply only to this case study [39]; future experiments should confirm the general validity of our observations using controlled environments.

12.1 How validity is constructed for this study

Why can the reader trust the results of this study? We employed several techniques to create internal validity. We used multiple sources of evidence and different types of data that on the one hand could be triangulated to isolate the factors of interest and reduce bias, and on the other hand broadened the view of the study. In addition, we employed randomization in two places: once when creating the student teams, and once when assigning the programming model to the teams.

We collected all data in a planned and consistent way from reliable sources. The code was indeed produced by the students in the respective teams; this was obvious from the interviews, coding style, or problem reports. The reported programming effort was checked for plausibility by looking at the interview data, logs of the subversion repository, and the code. The first two authors and experienced Intel developers participated in code inspections to assess the quality of the search engine code. The student answers in the post-project questionnaire were not biased by their teammates, as each student answered the questions individually. The answers are honest, as they correspond with the student's experience profiles and the team mate's responses (e.g., who did most of the work in the team).

12.2 Threats to validity

Of course, a case study like this has limitations. It is easy to disprove general statements even with a small number of subjects, but difficult to prove general statements.

The study uses just one type of application. It is possible that results differ for other applications.

Despite a pre-selection prior to the lab, student experience varied. The teaching phase of the lab aimed to bring all students to a similar parallel programming level prior to the project. We provide detailed experience profiles of all students prior to the project in Figure 2; when appropriate, this data was used to explain certain observations throughout the study.

The nature of the case study is to have a realistic, but not entirely controlled environment. Thus, other unknown factors could have influenced the results. We tried to compensate by triangulating data from several sources.

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 [47]. 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.

TM team 2 may distort the aggregate values for code metrics and effort because they had an incomplete program and were inexperienced. We therefore made conclusions based on comparisons of individual teams and used aggregated data only as additional information for the reader where appropriate. Due to their inexperience, team 2 produced fewer lines of code yet spent more effort on certain tasks than all other teams – for example, they spent significantly more time on design and experimentation than the other teams. For effort comparison, locks teams and TM teams could be paired as follows: teams 6 and 5 (because they were the winners), teams 3 and 4 (who both spent most effort), and team 2 and 1 (who did not have a good program – either not working or working with bad performance). Even if we had eliminated the pair of teams 1 and 2 from the effort comparison, the aggregated implementation effort would still significantly differ by 55 hours between TM teams and locks teams; debugging would still differ by 7 hours. The effort spent on design would only differ by 2 hours, becoming insignificant. This is because team 2 tried to compensate their inexperience by spending more time on conceptual work; team 2 spent most time of all teams on design (33 hours), which is 21 hours more than team 6 (the TM winners) and 12 hours more than TM team 3.

13. 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 [21, 28], software [37, 3, 7, 24, 30], or a hybrid of the two [14, 25]. 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 [32, 47] – for example, the Stanford ParalleL Applications for SHared memory (SPLASH-2) [43], the PARSEC benchmark suite [9], or SPEC OMP [38]. In addition, TM-specific benchmark suites have been developed, such as the Stanford Transactional Applications for Multi-Processing (STAMP) [27]. All of these benchmarks consist mostly of numerical applications.

Various case studies have assessed the performance of non-trivial applications using TM. Example applications include Lee's algorithm for circuit routing [5], the Linux sendmail application [35], among others [36, 18, 41]. These studies did not pay attention to software engineering aspects of Transactional Memory. Rossbach et al. [33] looked at errors in the programs of undergraduate students from different classes over 2 semesters. The students wrote a synthetic application in Java using TM APIs and locks. The results of [33] support our observations that TM leads to safer parallel programs and that implementing fine-grained locking is difficult and error-prone.

14. FUTURE RESEARCH DIRECTIONS

Grounded on the empirical results of this case study, we discuss future research directions and make suggestions how to address some of identified problems.

14.1 Improvements for Transactional Memory

Our results provide empirical feedback for TM language designers. This study also shows that we need better tools for program understanding, performance monitoring, and debugging of TM-based programs.

Libraries.

Team 6, the TM winners, devoted a lot of effort to implement low-level functionality that they would have expected from a library; this also becomes apparent in their code size, which is the largest of all teams. We need better thread-safe library support for C++ based TM programs to provide functionality similar to the C++ Standard Template Library. Such TM libraries must already include TM compiler annotations and be tested with TM.

TM performance tuning.

The performance of TM was not well-understood, which means that it was difficult for the TM teams to predict how their TM programs would perform. We need better techniques to analyze TM performance and improve the performance of atomic code blocks. We also need performance models for TM to predict the performance of TM-based programs.

Refactoring transaction code.

Team 3 had to refactor part of their transaction code late into the project because of bad performance. We therefore need tools to help programmers break atomic blocks into finer-grained atomic blocks or avoid data conflicts between atomic blocks of different threads. We also need code analysis tools that provide developers with supportive information when refactoring atomic blocks in order to gain better performance. Such analysis tools could be combined with performance prediction models for TM.

Design patterns for TM.

Races were common for most teams. Even the winning TM team had races. Patterns and anti-patterns for TM-based programs could help avoid races. For example, an anti-pattern would provide an example were the assumption that a concurrent read operation does not have to be protected by a transaction is not safe. In addition, patterns and anti-patterns could help avoid TM performance problems.

TM debugging.

The results of the questionnaires, interviews, and effort data clearly indicate that we need better debugging tools for TM. The winning team had races that were detected only by code inspection; no tool support was available for TM. Adding logging constructs to capture debugging messages in transactions was a missing feature that would have been useful for the students. A capture and replay approach [29] for transactions could be a useful extension for debugging TM-based code.

Improvements for TM language design.

Based on the code inspection results, we believe that type systems that avoid low-level races would be useful. Such type systems would ensure that a transactional piece of code is accessed only within transactions.

Our results show that only one team used the *tm_pure* construct, and used it in only one place to optimize string copy for immutable strings. It was hard for the code inspectors to verify the correctness of this usage, because it was not clear initially that the string arguments were immutable. We don't think that this feature should be exposed to average programmers. Learning from this particular case, we recommend a declarative construct for expressing immutability that the compiler can automatically verify and optimize.

No team had exception-safe code, even the TM teams who could have used the *tm_abort* construct. Under time pressure, programmers avoided explicit implementation of exception safety. We recommend therefore providing automatic exception-safety for atomic blocks.

Two TM teams successfully combined the usage of TM with semaphore or lock constructs from the Pthreads library for producer-consumer coordination or I/O, respectively. This worked very well and naturally; the code was simple, and it seemed appropriate. Although TM programming models have proposed the *retry* construct [19] for such type of coordination, it is not well-integrated into the C/C++ language models because it interacts poorly with irrevocable actions. Thus, this study shows that for producer-consumer type of coordination we need to either allow transactions to interoperate with semaphores, or we need to introduce some other mechanisms into the language.

Controlled Experiments.

An important insight of this study is that TM and locks can be used in a complementary way; they need not be considered as alternatives. To better quantify these benefits, we need experiments with programmers in environments that are more strictly controlled than in this case study.

We observed that the TM teams spent less overall time on implementation and debugging than the locks teams. In our realistic environment, this result could be influenced by factors other than the choice of locks or TM. A controlled experiment with more programmers must verify if we can draw a statistically significant conclusion that programming with TM leads to less implementation and debugging effort in comparison to locks.

14.2 Empirical language engineering

The success of a language depends not only on the efficiency of the compiler implementation, but also on its adoption by a programmer community. Without early feedback on language design, there is a high risk that a new language will not be adopted, despite existing efficient compiler implementations. We argue that programming language design – for sequential or parallel languages – must be put into a quality assurance feedback loop with empirical studies to detect diverging programmer expectations and language designer assumptions. This systematic process provides early feedback on how to improve a language (e.g., by detecting cognitive mismatches between programmer's intuitions and language construct semantics). The case study in this paper represents one cycle in such a quality assurance loop.

By collecting and comparing objective data and subjective programmer feedback, we can better understand the programmer's thought process and better explain why mistakes are made. In our study, we collected empirical data manually, which was very tedious. For future evaluations, however, most of the data collection can be automated. Such studies could be thus conducted more frequently and with a larger number of programmers.

Objective data can be collected automatically by modifying software development environments such as Eclipse [15]. For example, such collected data would include code or pieces of code using certain language constructs, code metrics, versioning information from repositories, compiler errors and warnings, or the steps a programmer performed in a debugger. To capture the hours spent on particular programming tasks, versioning repositories could be extended to present the programmers with a form to complete after each check-in.

The collection of subjective data can be automated by integrating user feedback mechanisms into software development environments. For example, users could annotate language constructs, pieces of code, compiler errors, or parts of the call graph. Small questionnaires could pop up as a context menu on demand or when a certain language construct is used for the first time, asking to rate how useful it was in that particular context (the collected data would include part of the code to reconstruct the context).

The collected objective and subjective data can be stored in a central repository for further analysis. Future research could address how to analyze this repository (e.g., with data mining techniques) in order to automatically detect patterns that lead to insights for better language design.

Although this approach would speed up the language design process, we are aware that it could raise privacy, copyright, or security issues; we believe, however, that these issues are negligible in many contexts. For example, most of the collected data can be anonymized before it is transmitted to a central repository. Pieces of code or aggregated metrics might suffice language designers to get an impression of the usage of a language construct. Finally, the collection of data can be restricted to a group of programmers (e.g., beta testers) who agreed to the terms and conditions of data collection.

15. CONCLUSION

The evidence in this case study of a realistic programming scenario supports Transactional Memory's promises. This is the first study to provide insights from variety of data, including application performance, code quality, code metrics, effort, and subjective programmer impressions. The winning TM team spent less overall effort compared to the winning locks team and had better performance. All locks teams tried to scale by using many fine-grained locks, yet it was difficult for them to achieve scalable performance. Only one locks team using over 1600 locks had a scalable parallel program.

The study shows that TM and locks were successfully employed in a complementary way; they need not be considered as alternatives excluding each other. The evidence also shows that to realize fully the benefits of TM in C++ we need language refinements supporting condition synchronization, and we need debugging and performance tuning tool support. For most TM teams, it was difficult to understand the behavior of their TM program.

This case study showed that even with TM, parallel programming remains difficult, so the quest for new parallel programming language features must continue. We need, however, systematic approaches to evaluate parallel programming proposals and make designs converge quickly to the programmer's needs. This paper presented one such approach using empirical studies. For future studies, we propose to automate data collection and analysis, which were done manually in this study. We call this new approach empirical language engineering.

Acknowledgements

We would like to thank the students in our multicore lab for their enthusiasm and the Excellence Initiative at the University of Karlsruhe, Germany, for the financial support. We thank Matthias Dempe and Nikolay Petkov for assisting us with the performance measurements. Many thanks to Intel for providing us with licenses for the STM compiler. At Intel, we acknowledge the support of the STM team, in particular, Ravi Narayanaswamy, Yang Ni, Tatiana Shpeisman, Xinmin Tian, and Adam Welc for their feedback during code inspections. We also thank Chris Vick at Sun Microsystems Labs for feedback.

Feedback

We are happy to receive feedback. Please send your feedback to `pankratius@ipd.uka.de`

Biographies

Dr. Victor Pankratius heads the young investigator "Software Engineering for Multicore Systems" group at the University of Karlsruhe, Germany. He serves as the elected chairman of the "Software Engineering for parallel Systems" international working group in the German Computer Science Society. Dr. Pankratius' current research concentrates on how to make parallel programming easier for the average programmer. His work on multicore software engineering covers a range of research topics including empirical studies, auto-tuning, parallel programming models, language design, and debugging. Dr. Pankratius received the Intel Leadership Award and holds a PhD with distinction from the University of Karlsruhe, Germany.

Ali-Reza Adl-Tabatabai, PhD is a Senior Principal Engineer in Intel's Programming Systems Lab. He leads a team of researchers working on compilers and scalable runtimes for future Intel Architectures. His current research concentrates on language features that make it easier for the mainstream developer to build reliable and scalable parallel programs for future multi-core architectures and on architectural support for those features. Most recently he has worked on transactional memory. Ali holds 24 patents and has published over 30 papers in leading conferences and journals. He received his PhD in Computer Science from Carnegie Mellon University.

Frank Otto is a PhD student at the University of Karlsruhe, Germany. His research focuses on parallel programming languages and the integration of stream computing into general-purpose languages such as Java. He holds a Master's degree in Computer Science from the University of Karlsruhe, Germany.

16. REFERENCES

[1] Clojure. http://clojure.org, last accessed August 2009.

[2] TM bibliography. http://www.cs.wisc.edu/trans-memory/biblio/index.html, last accessed August 2009.

[3] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM.

[4] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification version 1.0. http://research.sun.com/projects/plrg/fortress0707.pdf, 2008.

[5] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Luj̨an, and K. Jarvis. Lee-tm: A non-trivial benchmark suite for transactional memory. In *Algorithms and Architectures for Parallel Processing*, pages 196–207, 2008.

[6] D. Bacon et al. The "double-checked locking is broken" declaration. http://www.cs.umd.edu/ pugh/java/memoryModel/DoubleCheckedLocking.html, last accessed August 2009.

[7] W. Baek, C. C. Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun. The opentm transactional application programming interface. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 376–387, Washington, DC, USA, 2007. IEEE Computer Society.

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

[9] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, New York, NY, USA, 2008. ACM.

[10] D. R. Butenhof. *Programming with POSIX Theads*. Addison Wesley, 1997.

[11] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: why is it only a research toy? *Commun. ACM*, 51(11):40–46, 2008.

[12] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[13] Cray Inc. Chapel Specification 0.782. http://chapel.cray.com/spec-0.782.pdf.

[14] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, New York, NY, USA, 2006. ACM.

[15] Eclipse Foundation. Eclipse. http://www.eclipse.org/, last accessed August 2009.

[16] Graphviz. Graph visualization software. http://www.graphviz.org/, last accessed August 2009.

[17] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[18] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: a benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3):315–324, 2007.

[19] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[20] S. Heinz, J. Zobel, and H. E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223, 2002.

[21] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[22] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[23] IEEE. Standard for information technology - portable operating system interface (posix). base definitions. IEEE Std 1003.1, 2004. The Open Group Technical Standard Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. Base Definitions.

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

[25] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 209–220, New York, NY, USA, 2006. ACM.

[26] N. Lester, A. Moffat, and J. Zobel. Efficient online index construction for text databases. *ACM Trans. Database Syst.*, 33(3):1–33, 2008.

[27] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[28] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. Logtm: log-based transactional memory. In *The Twelfth International Symposium on High-Performance Computer Architecture (HPCA)*, pages 254–265, February 2006.

[29] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 22–31, New York, NY, USA, 2007. ACM.

[30] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach,

S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for c/c++. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 195–212, New York, NY, USA, 2008. ACM.

[31] M. F. Ringenburg and D. Grossman. Atomcaml: first-class atomicity via rollback. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 92–104, New York, NY, USA, 2005. ACM.

[32] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel. Txlinux: using and managing hardware transactional memory in an operating system. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 87–102, New York, NY, USA, 2007. ACM.

[33] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional memory programming actually easier? In *Proceedings of the 8th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, Austin, Texas, Junes 2009.

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

[35] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM.

[36] M. Scott, M. Spear, L. Dalessandro, and V. Marathe. Delaunay triangulation with transactions and barriers. In *IEEE 10th International Symposium on Workload Characterization (IISWC 2007).*, pages 107–113, Sept. 2007.

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

[38] Standard Performance Evaluation Corporation. Spec openmp benchmark suite. http://www.spec.org/omp, 2009.

[39] W. F. Tichy. Hints for reviewing empirical work in software engineering. *Empirical Software Engineering*, 5(4):309–312, 2000.

[40] Valgrind team. Valgrind tool suite. http://valgrind.org/, last accessed August 2009.

[41] I. Watson, C. Kirkham, and M. Lujan. A study of a transactional parallel routing algorithm. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 388–398, Washington, DC, USA, 2007. IEEE Computer Society.

[42] I. Witten, T. Bell, and J. Cleary. The calgary corpus. http://www.data-compression.info/Corpora, 1987.

[43] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM.

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

[45] C. Zannier, G. Melnik, and F. Maurer. On the success of empirical studies in the international conference on software engineering. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 341–350, New York, NY, USA, 2006. ACM Press.

[46] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.

[47] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero. Atomic quake: using transactional memory in an interactive multiplayer game server. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 25–34, New York, NY, USA, 2009. ACM.

APPENDIX

A. INFLUENCE OF COMPILERS ON PERFORMANCE

B. EFFORT DIAGRAMS

C. QUESTIONNAIRE

Appendix A. Influence of compilers on performance

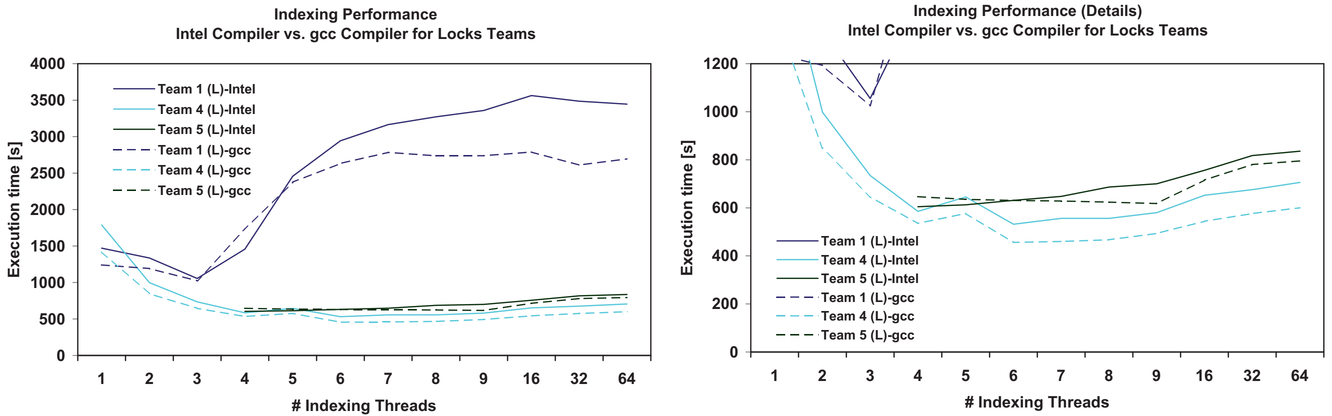
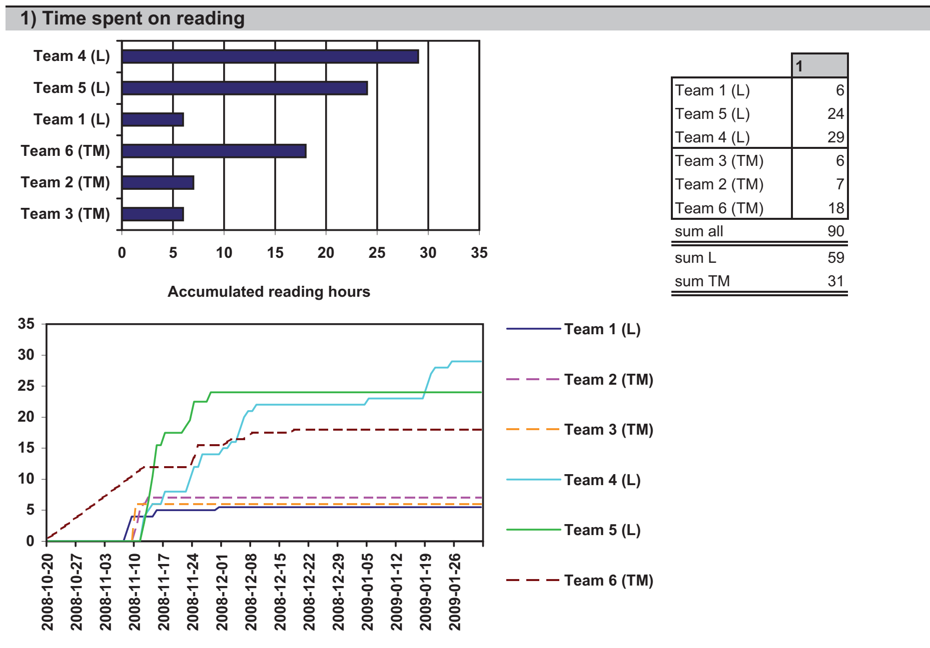


Figure 10: Performance comparison between compilers for indexing. The programs of all locks teams were compiled with Intel's C++/STM compiler prototype v. 2.0 and gcc 4.1.3. They were executed on a Dell eight core machine with 2x Intel Quadcore E5320 QC processors, clocked at 1.86 GHz, using 8 GB RAM, running Ubuntu Linux 2.6.

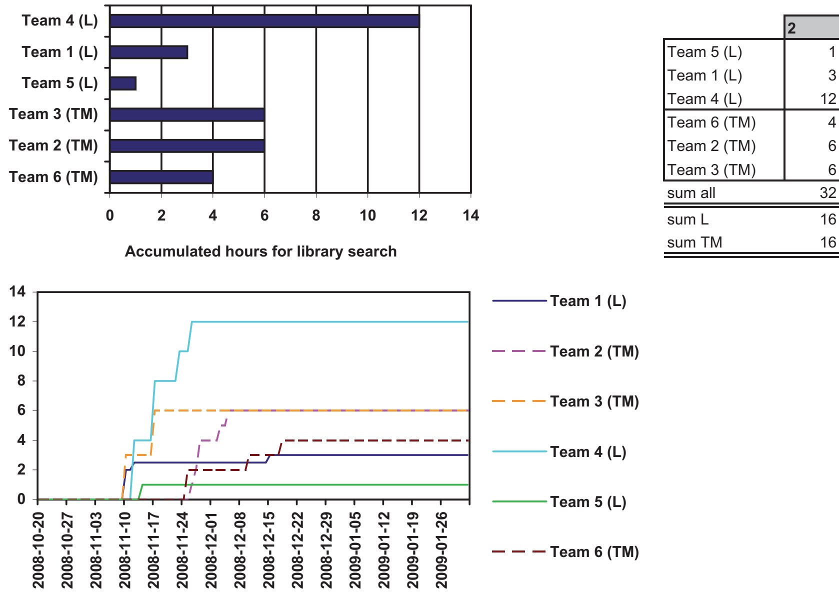
Appendix B. Effort diagrams



	1
Team 1 (L)	6
Team 5 (L)	24
Team 4 (L)	29
Team 3 (TM)	6
Team 2 (TM)	7
Team 6 (TM)	18
sum all	90
sum L	59
sum TM	31

Figure 11: Time in person hours spent on reading.

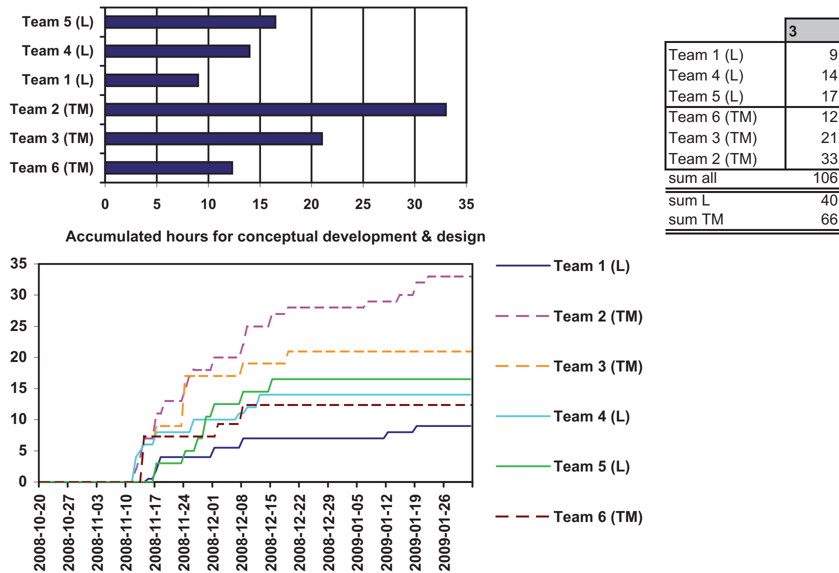
2) Time spent on search for libraries



	2
Team 5 (L)	1
Team 1 (L)	3
Team 4 (L)	12
Team 6 (TM)	4
Team 2 (TM)	6
Team 3 (TM)	6
sum all	32
sum L	16
sum TM	16

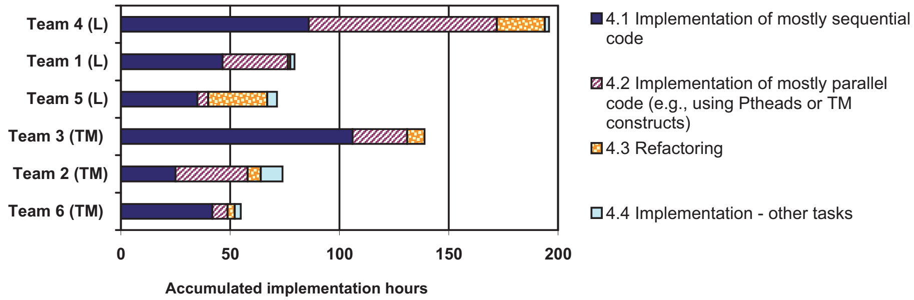
Figure 12: Time in person hours spent on search for libraries.

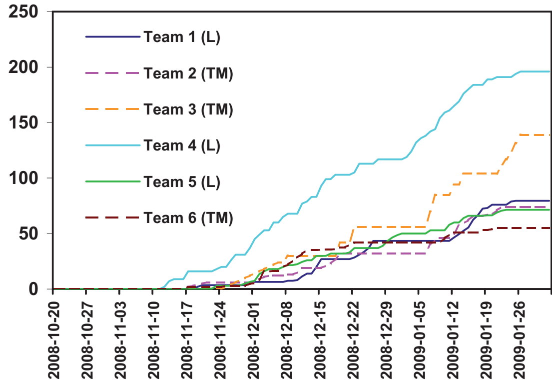
3) Time spent on conceptual development and design



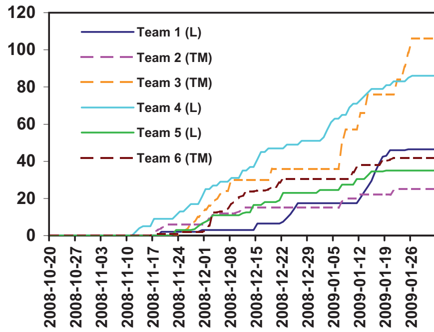
	3
Team 1 (L)	9
Team 4 (L)	14
Team 5 (L)	17
Team 6 (TM)	12
Team 3 (TM)	21
Team 2 (TM)	33
sum all	106
sum L	40
sum TM	66

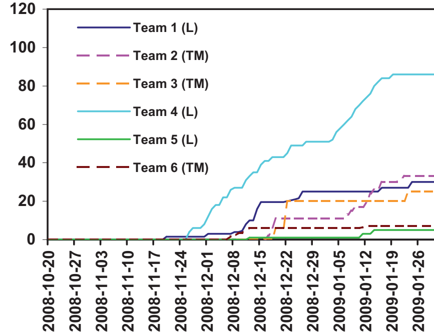
Figure 13: Time in person hours spent on design.





	4.1	4.2	4.3	4.4	
Team 5 (L)	35	5	27	5	72
Team 1 (L)	47	30	1	2	80
Team 4 (L)	86	86	22	2	196
Team 6 (TM)	42	7	3	3	55
Team 2 (TM)	25	33	6	10	74
Team 3 (TM)	106	25	8	0	139
sum all	**341**	**186**	**67**	**22**	**616**
	55%	30%	11%	4%	100%
sum L	**168**	**121**	**50**	**9**	**348**
	48%	35%	14%	3%	100%
sum TM	**173**	**65**	**17**	**13**	**268**
	65%	24%	6%	5%	100%





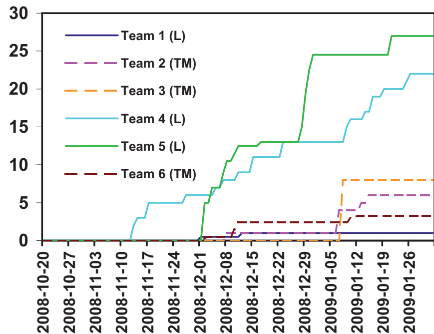
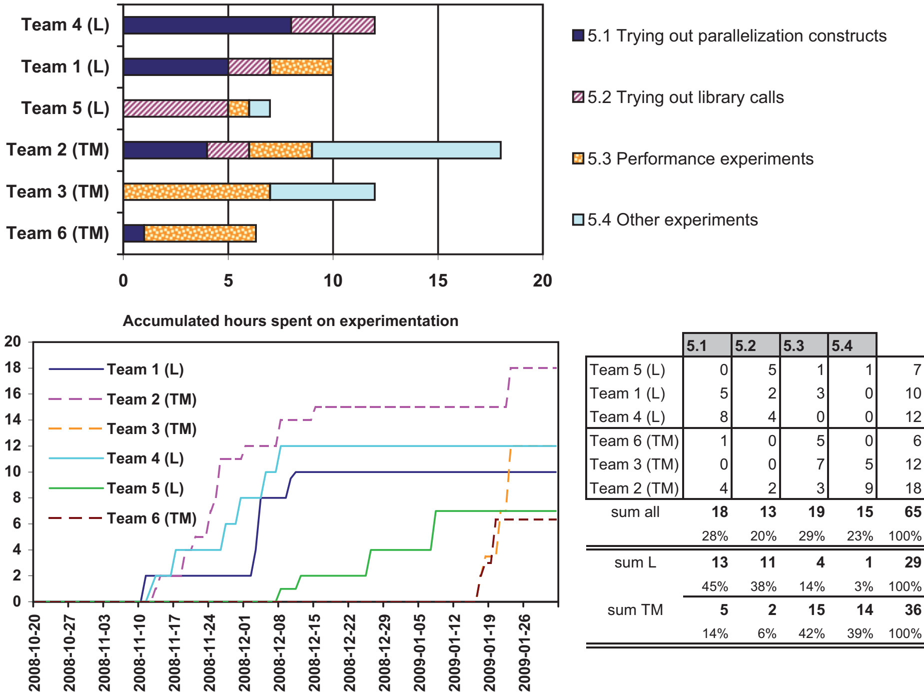
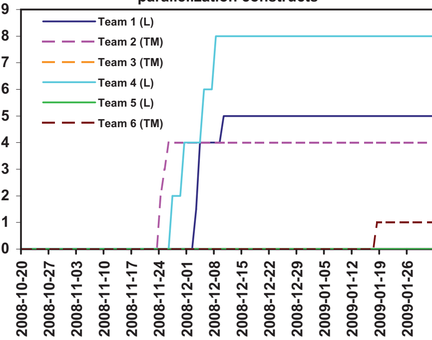
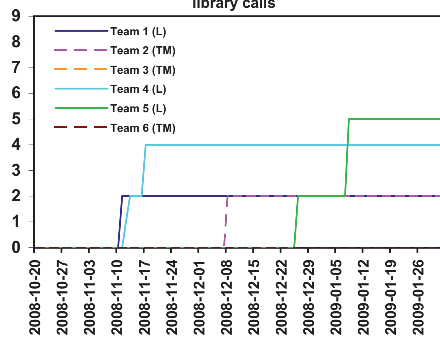


Figure 14: Time in person hours spent on implementation.



	5.1	5.2	5.3	5.4	
Team 5 (L)	0	5	1	1	7
Team 1 (L)	5	2	3	0	10
Team 4 (L)	8	4	0	0	12
Team 6 (TM)	1	0	5	0	6
Team 3 (TM)	0	0	7	5	12
Team 2 (TM)	4	2	3	9	18
sum all	18	13	19	15	65
	28%	20%	29%	23%	100%
sum L	13	11	4	1	29
	45%	38%	14%	3%	100%
sum TM	5	2	15	14	36
	14%	6%	42%	39%	100%





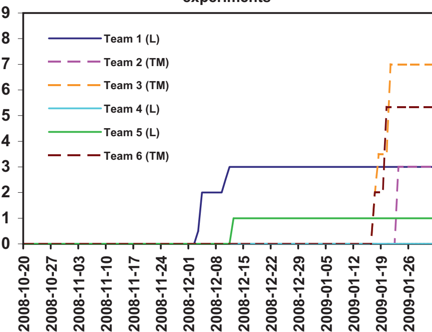
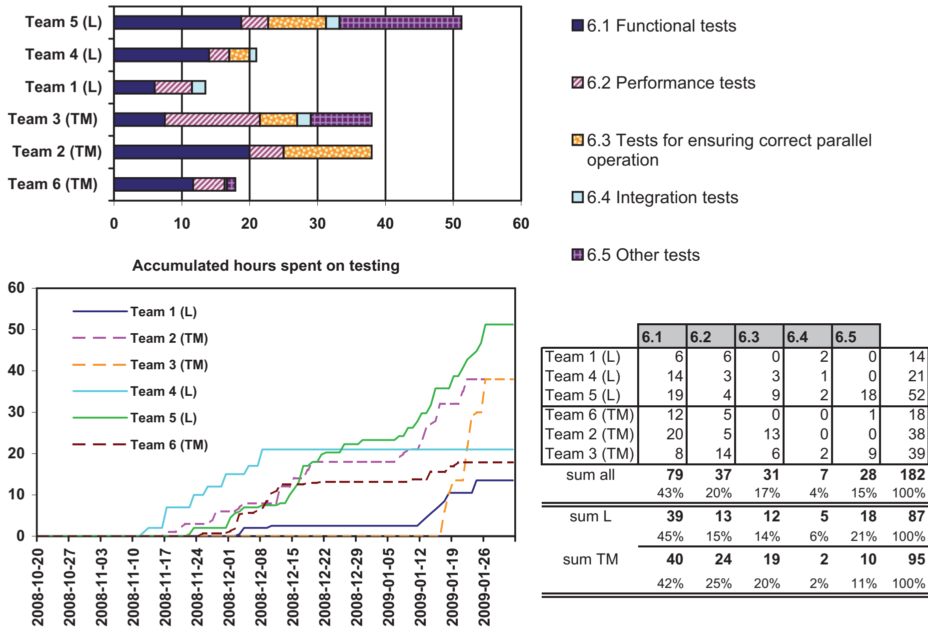
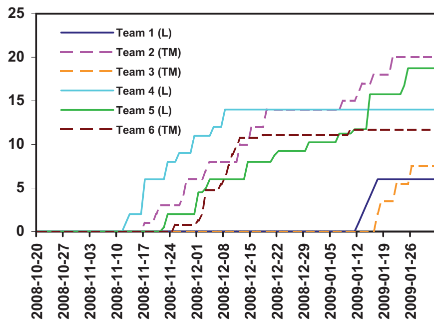
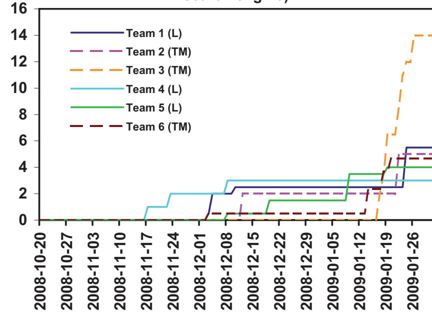


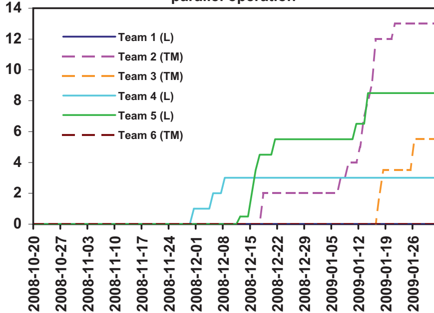
Figure 15: Time in person hours spent on experimentation.



	6.1	6.2	6.3	6.4	6.5	
Team 1 (L)	6	6	0	2	0	14
Team 4 (L)	14	3	3	1	0	21
Team 5 (L)	19	4	9	2	18	52
Team 6 (TM)	12	5	0	0	1	18
Team 2 (TM)	20	5	13	0	0	38
Team 3 (TM)	8	14	6	2	9	39
sum all	79	37	31	7	28	182
	43%	20%	17%	4%	15%	100%
sum L	39	13	12	5	18	87
	45%	15%	14%	6%	21%	100%
sum TM	40	24	19	2	10	95
	42%	25%	20%	2%	11%	100%







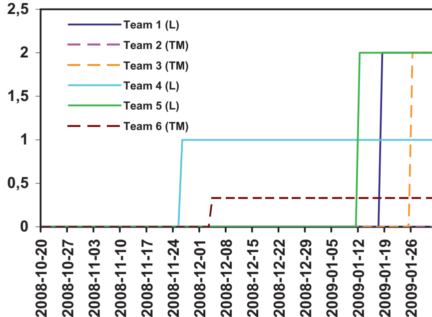
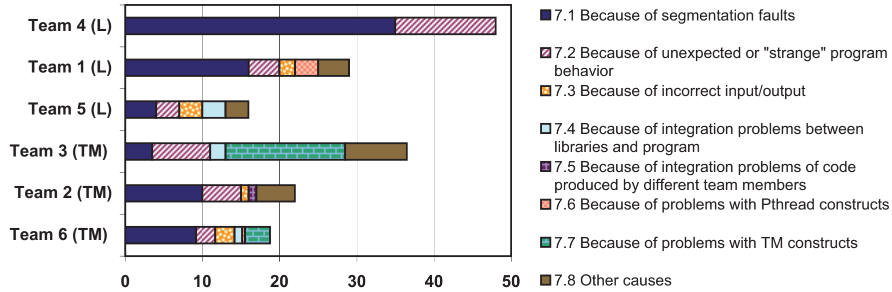
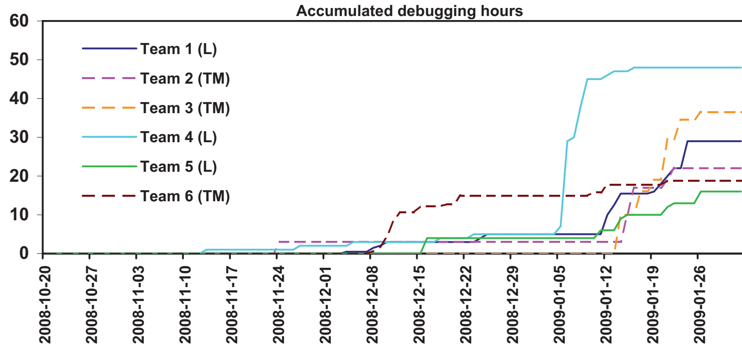


Figure 16: Time in person hours spent on testing.

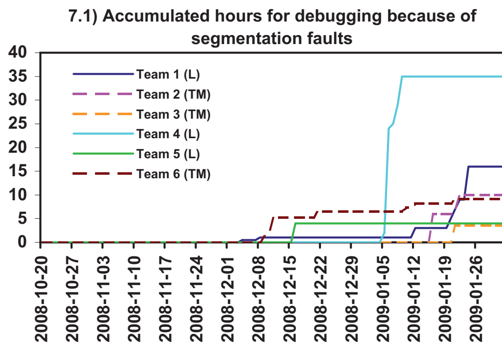
7) Time spent on debugging

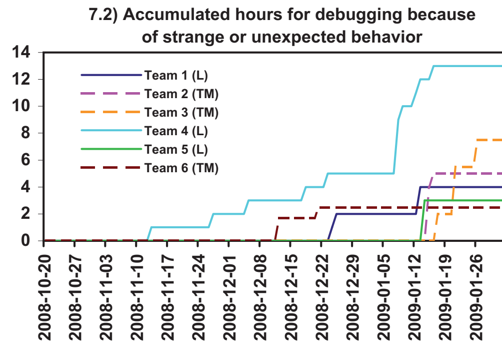


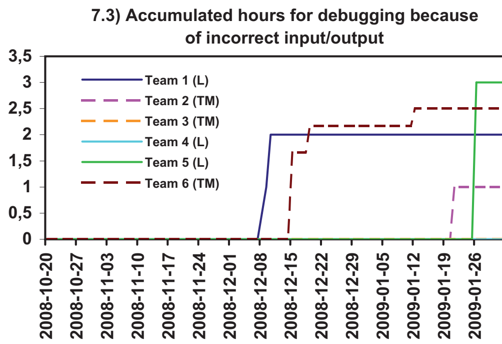


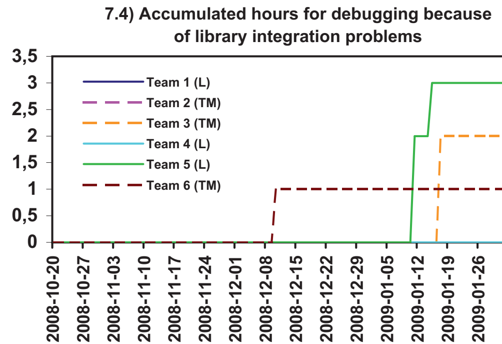
	7.1	7.2	7.3	7.4	7.5	7.6	7.7	7.8	
Team 5 (L)	4	3	3	3	0	0	0	3	16
Team 1 (L)	16	4	2	0	0	3	0	4	29
Team 4 (L)	35	13	0	0	0	0	0	0	48
Team 6 (TM)	9	3	3	1	0	0	3	0	19
Team 2 (TM)	10	5	1	0	1	0	0	5	22
Team 3 (TM)	4	8	0	2	0	0	16	8	38
sum all	**78**	**36**	**9**	**6**	**1**	**3**	**19**	**20**	**172**
	45%	21%	5%	3%	1%	2%	11%	12%	100%
sum L	**55**	**20**	**5**	**3**	**0**	**3**	**0**	**7**	**93**
	59%	22%	5%	3%	0%	3%	0%	8%	100%
sum TM	**23**	**16**	**4**	**3**	**1**	**0**	**19**	**13**	**79**
	29%	20%	5%	4%	1%	0%	24%	16%	100%

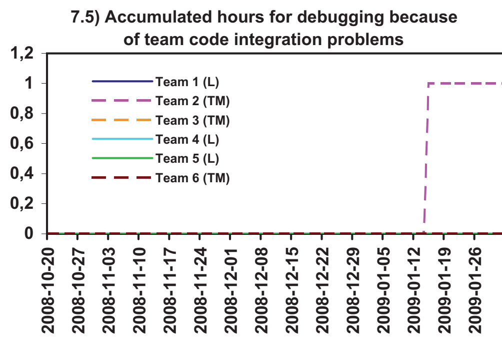
Figure 17: Time in person hours spent on debugging.

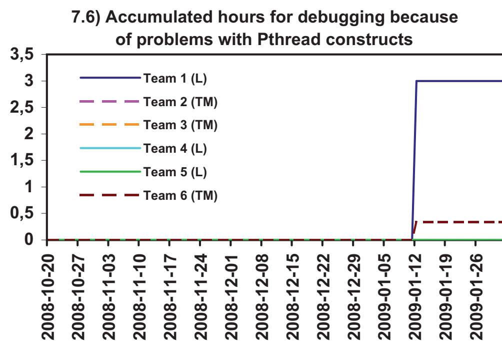












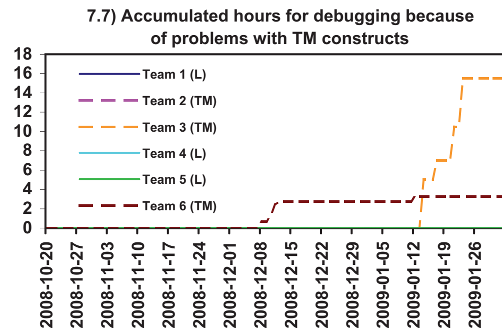


Figure 18: Effort development for subtasks of debugging.

Appendix C. Questionnaire

		Locks Teams						Transactional Memory Teams					
		Team 1		Team 4		Team 5		Team 2		Team 3		Team 6	
		S1	S2	S7	S8	S9	S10	S3	S4	S5	S6	S11	S12
(a) Working strategy													
Q1	**How easy or difficult was the assignment to understand?** (1-very easy; 5-very difficult)	2	2	2	2	1	1	2	2	1	2	2	2
Q2	**How complex do you rate this project?** (1-very easy; 5-very complex)	3	4	3	4	3	3	3	4	3	2	3	2
Q3	**How easy or difficult was it to create a concrete working plan at the beginning of the project?** (1-very easy; 5-very difficult)	1	4	5	3	4	4	3	4	1	2	4	3
Q4	**How did you plan the work in your team?** (1-spontaneous decisions; 2-structured with goals set for next 1-3 days; 3: next 4-7 days; 4: next 8-14 days; 5: >14 days)	1	5	1	3	1	5	2	3	3	1	3	2
Q5	**I had the impression that I did most of the work in my team** (1-not true; 2-partly true; 3-true for many modules; 4-true for most modules; 5-true for all modules)	4	1	5	1	4	1	1	3	1	5	4	1
Q6	**How did you perceive time pressure during the different phases of the project?** (1-no time pressure, could complete all planned tasks by deadline; 2-sometimes short of time, but could still complete planned tasks; 3-could not complete some of planned tasks; 4-could not complete about half of planned tasks; 5-could not complete most of planned tasks by planned deadline)												
	during initiation phase	1	1	4	1		1	1	1	1	1	2	2
	during design phase	1	1	2	1	1	1	1	1	1	1	1	1
	during implementation phase	2	3	3	3	2	5	4	3	3	3	3	3
	during testing phase	2	3	4	3	2	4	3	4	4	3	4	3
Q7	**Roles in your team** (1-no clear role definition; 2-all members worked equally on all tasks; 3-each one was specialist in particular area, general project coordination together; 4-each one was specialist in particular area, but one of us responsible for the general coordination)	4	4	4	2		4	3	3	2	1	4	4
Q8	**Importance of tools for cordination** (rank the following 4 choices using a rank exactly once: 1-most important to 4-most unimportant)												
	Face-to-face Meetings	2	2	3	1		1	1	4	1	2	1	1
	Email	3	1	4	4		2	3	3	2	1	2	2
	Instant Messaging	4	4	2	3		3	4	1	4	4	4	3
	Subversion (SVN)	1	3	1	2		4	2	2	3	3	3	4
(b) Design													
Q1	**In the design process, how easy or difficult was it to identify the places for parallelization?** (1-very easy; 5-very difficult)	2	2	2	3	1	1	3	3	2	2	3	2
Q2	**How easy or difficult was it to modularize the parts that worked in parallel?** (1-very easy; 5-very difficult)	1	3	3	4	2	1	2	3	2	2	3	2
Q3	**How important was it for you to know in the design phase that you would use locks or TM later?** (1-not important; 5-very important)	1	1	4	2	2	5	4	4	1	2	2	5
(c) Implementation													
Q1	**In the implementation process, how easy/difficult was it to identify the places for parallelization?** (1-very easy; 5-very difficult)	2	3	2	3	1	2	2	2	3	2	3	2
Q2	**How important was it for you to get quickly to the first executable parallel program?** (1-not important; 5-very important)	5	3	4	4	3	3	4	4	4	5	5	5
Q3	**When did you first employ parallel constructs?** (1-already in first lines of code; 2-after first reliable compilation and execution; 3-after implementing a few features; 4-after implementing most features; 5-after implementing all features)	3	2	3	3	3	1	2	3	2	3	3	1
Q4	**During implementation, we had to fundamentally rethink our overall design** (1-never; 2-one time; 3: 2-3 times; 4: 4-6 times; 5: >6 times)	1	1	1	2	2	3	2	2	1	1	1	1
Q5	**The performance of our <u>first</u> parallel program was** (1-way below our expectations; 2-somewaht below our expectations; 3-as expected; 4-somewhat above our expectations; 5-way above our expectations)	3	1	3	3		4	1	2	5	3	4	3
Q6	**The performance of our <u>final</u> parallel program was** (1-way below our expectations; 2-somewaht below our expectations; 3-as expected; 4-somewhat above our expectations; 5-way above our expectations)	3	1	3	3		3	2	1	5	1	2	3
(d) Psychological issues during implementation													
Q1	**How easy/difficult was it to keep track of what your parallel program is doing?** (1-very easy; 5-very difficult)	1	3	2	3	3	2	2	4	4	3	2	3
Q2	**I felt uncomfortable when using parallel constructs** (1-never; 5-very often)	1	3	2	3	1	1	2	2	5	2	3	1
Q3	**I was afraid of "destroying" my working program by using additional parallel constructs** (1-never; 5-very often)	1	4	2	3	3	1	1	2	4	2	1	1
Q4	**Our team tried to postpone parallelization work** (1-never; 5-very often)	1	3	2	2	1	1	2	3	3	1	3	2
Q5	**During implementation, we had to backtrack to earlier versions and start over** (1-never; 2-one time; 3: 2-3 times; 4: 4-6 times; 5: >6 times)	1	1	3	1	1	1	3	3	1	1	1	1

Figure 19: Questionnaire results on working strategy, design, implementation, and psychological issues during implementation.

		Locks Teams						Transactional Memory Teams					
		Team 1		Team 4		Team 5		Team 2		Team 3		Team 6	
		S1	S2	S7	S8	S9	S10	S3	S4	S5	S6	S11	S12
(a) Testing and debugging													
Q1	**During implementation I confronted errors that I could not diagnose** (1- never; 2- one time; 3: 3-10 times; 4: 11-20 times; 5: >20 times)	2	3	3	3	3	5	4	3	5	4	4	4
Q2	**During debugging, how easy/difficult was it to understand the ordering of parallel events?** (1- very easy; 5- very difficult)	3	5	5	3	3	3	2	4	5	5	5	3
Q3	**How often did you encounter errors of the following types?** (1- never; 5- very frequently)												
	Race conditions because of forgotten sync constructs	1	3	1	2	1	3	2	2	2	1	1	2
	Race conditions due to other causes	1	3	1	1	2	2	2	2	2	2	1	1
	Ordering errors during parallel execution of threads	1	3	1	1	2	1	1	2	1	1	1	1
	Wrong initializations (e.g., variables, threads, etc.)	2	1	3	2	1	1	2	2	3	4	2	2
	Wrong assumptions about libraries (e.g., not threadsafe)	1	1	1	1	1	1	1	2	3	3	1	1
	Deadlocks	1	4	1	2	1	3	2	1	2	1	1	1
	Memory leaks	1	1	1	1	1	2	3	2	4	2	1	2
	Segmentation faults	3	3	3	3	3	4	4	4	5	3	4	5
	Unexplainable crashes	2	3	3	2	1	5	1	4	5	3	3	4
	Forgotten signal/wait	1	3	2	1	2	2	1	1	1	1	1	1
	Wrong assumptions about atomicity / atomicity violations	1	0	1	1	1	1	1	2	4	3	3	1
(b) Pthreads and Transactional Memory constructs													
Q1	**How easy/difficult was it to use parallel constructs of Pthreads library (TM compiler)?** (1- very easy; 5- very difficult)	2	3	2	2	1	3	2	3	1	3	3	1
Q2	**How easy/difficult were the constructs of Pthreads (TM) to understand?** (1- very easy; 5- very difficult)	1	3	3	2	1	3	2	2	2	3	3	1
Q3	**How easy/difficult was the submitted parallel program to understand?** (1- very easy; 5- very difficult)	2	3	2	2	2	2	3	4	5	2	3	1
Q4	**Using Pthreads (TM), I had the impression of not advancing fast enough** (1-never; 5- very often)	1	2	2	1	2	4	2	3	4	3	4	3
Q5	**How often did you slip using Pthreads (TM) constructs?** (1- never; 5- very often)	1	4	3	2	2	1	2	4	4	2	3	2
Q6	**How detailed were you able to control parallelization using Phreads (TM)?** (1- impossible; 5- very detailed)	4	5	2	4	4	4	4	4	2	2	3	4
(c) Transactional Memory constructs													
Q1	**How easy/difficult were function annotations to use?** (1- very easy; 5- very difficult)							2	2	2	2	4	3
Q2	**How useful were the statistics generated by the STM compiler?** (1- not useful; 5- very useful)							3	4	1	3	4	3
Q3	**Were you able to use the tm_pure function annotation?** (0- no; 1- yes)							0	0	1	1	0	0
Q4	**Did you use the __tm_abort construct?** (0- no; 1- yes)							0	1	0	0	0	0
Q5	**Did you have to throw C++ exceptions out of atomic blocks?** (0- no; 1- yes)							0	0	0	0	0	1
Q6	**Did you need to perform irreversible I/O operations inside atomic blocks?** (0- no; 1- yes)							1	0	0	0	0	0
Q7	**Did you need to perform condition synchronization?** (0- no; 1- yes)							0	0	0	0	0	1
Q8	**Was the STM compiler well-documented?** (0- no; 1- yes)							0	1	0	1	1	0

Figure 20: Questionnaire results on testing, debugging, and parallel constructs.