Is Transactional Programming Actually Easier?

Christopher J. Rossbach and Owen S. Hofmann and Emmett Witchel
University of Texas at Austin
{rossbach,osh,witchel}@cs.utexas.edu

Abstract

Chip multi-processors (CMPs) have become ubiquitous, while tools that ease concurrent programming have not. The promise of increased performance for all applications through ever more parallel hardware requires good tools for concurrent programming, especially for average programmers. Transactional memory (TM) has enjoyed recent interest as a tool that can help programmers program concurrently.

The transactional memory (TM) research community is heavily invested in the claim that programming with transactional memory is easier than alternatives (like locks), but evidence for or against the veracity of this claim is scant. In this paper, we describe a user-study in which 237 undergraduate students in an operating systems course implement the same programs using coarse and fine-grain locks, monitors, and transactions. We surveyed the students after the assignment, and examined their code to determine the types and frequency of programming errors for each synchronization technique. Inexperienced programmers found baroque syntax a barrier to entry for transactional programming. On average, subjective evaluation showed that students found transactions harder to use than coarse-grain locks, but slightly easier to use than fine-grained locks. Detailed examination of synchronization errors in the students' code tells a rather different story. Overwhelmingly, the number and types of programming errors the students made was much lower for transactions than for locks. On a similar programming problem, over 70% of students made errors with fine-grained locking, while less than 10% made errors with transactions.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: [Concurrent Programming]

General Terms Design, Performance

Keywords Transactional Memory, Optimistic Concurrency, Synchronization

1. Introduction

The increasing ubiquity of chip multiprocessors has resulted in a high availability of parallel hardware resources. However, while parallel computing resources have become commonplace, concurrent programs have not; concurrent programming remains a challenging endeavor, even for experienced programmers. Transactional memory (TM) has enjoyed considerable research attention precisely because it promises to make the development of concurrent software easier. Transactional memory (TM) researchers position TM as an enabling technology for concurrent programming for the "average" programmer.

Transactional memory allows the programmer to delimit regions of code that must execute atomically and in isolation. It promises the performance of fine-grain locking with the code simplicity of coarse-grain locking. In contrast to locks, which use mutual exclusion to serialize access to critical sections, TM is typically implemented using optimistic concurrency techniques, allowing critical sections to proceed in parallel. Because this technique dramatically reduces serialization when dynamic read-write and write-write sharing is rare, it can translate directly to improved performance without additional effort from the programmer. Moreover, because transactions eliminate many of the pitfalls commonly associated with locks (e.g. deadlock, convoys, poor composability), transactional programming is touted as being easier than lock based programming.

Evaluating the ease of transactional programming relative to locks is largely uncharted territory. Naturally, the question of whether transactions are easier to use than locks is qualitative. Moreover, since transactional memory is still a nascent technology, the only available transactional programs are research benchmarks, and the population of programmers familiar with both transactional memory and locks for synchronization is vanishingly small.

To address the absence of evidence, we developed a concurrent programming project for students of an undergraduate Operating Systems course at *The University of Texas at Austin*, in which students were required to implement the same concurrent program using coarse and fine-grained locks, monitors, and transactions. We surveyed students about the relative ease of transactional programming as well as their investment of development effort using each synchronization technique. Additionally, we examined students' solutions in detail to characterize and classify the types and frequency of programming errors students made with each programming technique.

This paper makes the following contributions:

- A project and design for collecting data relevant to the question of the relative ease of programming with different synchronization primitives.
- Data from 237 student surveys and 1323 parallel programs that constitute the largest-scale (to our knowledge) empirical data relevant to the question of whether transactions are, in fact, easier to use than locks.
- A taxonomy of synchronization errors made with different synchronization techniques, and a characterization of the frequency with which such errors occur in student programs.

2. Sync-gallery

In this section, we describe sync-gallery, the Java programming project we assigned to students in an undergraduate operating sys-

PPoPP'10 January 9–14, 2010, Bangalore, India.

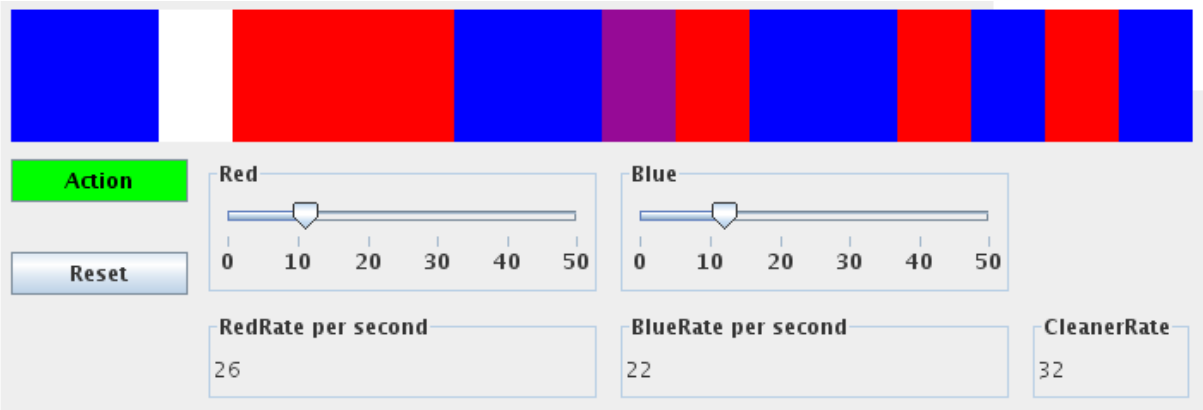


Figure 1. A screen-shot of sync-gallery, the program undergraduate OS students were asked to implement. In the figure the colored boxes represent 16 shooting lanes in a gallery populated by shooters, or *rogues*. A red or blue box represents a box in which a rogue has shot either a red or blue paint ball. A white box represents a box in which no shooting has yet taken place. A purple box indicates a line in which both a red and blue shot have occurred, indicating a race condition in the program. Sliders control the rate at which shooting and cleaning threads perform their work.

tems course. The project is designed to familiarize students with concurrent programming in general, and with techniques and idioms for using a variety of synchronization primitives to manage data structure consistency. Figure 1 shows a screen shot from the sync-gallery program.

The project asks students to consider the metaphor of a shooting gallery, with a fixed number of lanes in which *rogues* (shooters) can shoot in individual lanes. Being pacifists, we insist that shooters in this gallery use red or blue paint balls rather than bullets. Targets are white, so that lanes will change color when a rogue has shot in one. Paint is messy, necessitating *cleaners* to clean the gallery when all lanes have been shot. Rogues and cleaners are implemented as threads that must check the state of one or more lanes in the gallery to decide whether it is safe to carry out their work. For rogues, this work amounts to shooting at some number of randomly chosen lanes. Cleaners must return the gallery to its initial state with all lanes white. The students must use various synchronization primitives to enforce a number of program invariants:

1. **Only one rogue may shoot in a given lane at a time.**
2. **Rogues may only shoot in a lane if it is white.**
3. **Cleaners should only clean when all lanes have been shot (are non-white).**
4. **Only one thread can be engaged in the process of cleaning at any given time.**

If a student writes code for a rogue that fails to respect the first two invariants, the lane can be shot with both red and blue, and will therefore turn purple, giving the student instant visual feedback that a race condition exists in the program. If the code fails to respect to the second two invariants, no visual feedback is given (indeed these invariants can only be checked by inspection of the code in the current implementation).

We ask the students to implement 9 different versions of rogues (Java classes) that are instructive for different approaches to synchronization. Table 1 summarizes the rogue variations. Gaining exclusive access to one or two lanes of the gallery in order to test the lane's state and then modify it corresponds directly to the real-world programming task of locking some number of resources in order to test and modify them safely in the presence of concurrent threads.

2.1 Locking

We ask the students to synchronize rogue and cleaner threads in the sync-gallery using locks to teach them about coarse and fine-grain locking. To ensure that students write code that explicitly performs locking and unlocking operations, we require them to use the Java `ReentrantLock` class and do not allow use of the `synchronized` keyword. In locking rogue variations, cleaners do not use dedicated threads; the rogue that colors the last white lane in the gallery is responsible for becoming a cleaner and subsequently cleaning all lanes. There are four variations on this rogue type: **Coarse, Fine, Coarse2** and **Fine2**. In the coarse implementation, students are allowed to use a single global lock which is acquired before attempting to shoot or clean. In the fine-grain implementation, we require the students to implement individual locks for each lane. The Coarse2 and Fine2 variations require the same mapping of locks to objects in the gallery as their counterparts above, but introduce the additional stipulation that rogues must acquire access to and shoot at two random lanes rather than one. The variation illustrates that fine-grain locking requires a lock-ordering discipline to avoid deadlock, while a single coarse lock does not. Naturally, the use of fine grain lane locks complicates the enforcement of invariants 3 and 4 above.

2.2 Monitor implementations

Two variations of the program require the students to use condition variables along with signal/wait implement both fine and coarse locking versions of the rogue programs. The monitor variations introduce dedicated threads for cleaners: shooters and cleaners must use condition variables to coordinate shooting and cleaning phases. In the coarse version (**CoarseCleaner**), students use a single global lock, while the fine-grain version (**FineCleaner**) requires per-lane locks.

2.3 Transactions

Finally, the students are asked to implement 3 TM-based variants of the rogues that implement the same specification as their corresponding locking variations, but use transactional memory for synchronization instead of locks. The most basic TM-based rogue, **TM**, is analogous to the Coarse and Fine versions: rogue and cleaner threads are not distinct, and shooters need shoot only one lane, while the **TM2** variation requires that rogues shoot at two lanes rather than one. In the **TMCleaner**, rogues and cleaners have

```
TMInt y = new TMInt(0);
TMInt x = new TMInt(10);
Callable c = new Callable<Void> {
  public Void call() {
    // txnl code
    y.setValue(x.getValue() * 2);
    return null;
  }
}
Thread.doIt(c);
```

```
TMInt y = new TMInt(0);
TMInt x = new TMInt(10);
Transaction tx = new Transaction(id);
boolean done = false;
while(!done) {
  try {
    tx.BeginTransaction();
    // txnl code
    y.setValue(x.getValue() * 2);
    done = tx.CommitTransaction();
  } catch(AbortException e) {
    tx.AbortTransaction();
    done = false;
  }
}
```

```
int y = 0;
int x = 10;
atomic {
  y = x * 2;
}
```

Figure 2. Examples of (left) DSTM2 concrete syntax, (middle) JDASTM concrete syntax, and (right, for comparison) ideal atomic keyword syntax. Year 1 of the study used DSTM2, while years 2 and 3 used JDASTM.

dedicated threads. Students can rely on the TM subsystem to detect conflicts and restart transactions to enforce all invariants, so no condition synchronization is required.

2.4 Transactional Memory Support

Our ideal TM system would support atomic blocks in the Java language, allowing students to write transactional code of the form:

```
void shoot() {
  atomic {
    Lane l = getLane(rand());
    if(l.getColor() == WHITE)
      l.shoot(this.color);
  }
}
```

No such tool is yet available; implementing compiler support for atomic blocks, or use of a a source-to-source compiler such as spoon [2] were considered out-of-scope for the project. Instead we used a TM library.

Using a TM library means that students are forced to deal directly with the concrete syntax of our TM implementation, and must manage read and write barriers explicitly. We assigned the lab to 5 classes over 3 semesters during 3 different school years. During the first year both classes used DSTM2 [15]. For the second and third years, all classes used JDASTM [29].

The concrete syntax has a direct impact on ease of programming, as seen in Figure 2, which provides examples for the same task using DSTM2, JDASTM, and for reference, with language support using the `atomic` keyword. While the version with the `atomic` keyword is quite simple, both the DSTM2 and JDASTM examples pepper the actual data structure manipulation with code that explicitly manages transactions. We replaced DSTM2 in the second year because we felt that JDASTM syntax was somewhat less baroque and did not require students to deal directly with programming constructs like generics. Also, DSTM2 binds transactional execution to specialized thread classes. However, both DSTM2 and JDASTM require explicit read and write barrier calls for transactional reads and writes.

3. Methodology

Students completed the sync-gallery program as a programming assignment as part of several operating systems classes at *The University of Texas at Austin*. In total, 237 students completed the assignment, spanning five sections in classes from three different semesters, over three years of the course. The first year of the course included 84 students, while the second and third included 101 and 53 respectively. We provided an implementation of the shooting gallery, and asked students to write the rogue classes described in the previous sections, respecting the given invariants.

We asked students to record the amount of time they spent designing, coding, and debugging each programming task (rogue). We use the amount of time spent on each task as a measure of the difficulty that task presented to the students. This data is presented in Section 4.1. After completing the assignment, students rated their familiarity with concurrent programming concepts prior to the assignment. Students then rated their experience with the various tasks, ranking synchronization methods with respect to ease of development, debugging, and reasoning (Section 4.2).

While grading the assignment, we recorded the type and frequency of synchronization errors students made. These are the errors still present in the student's final version of the code. We use

Rogue name	Technique	R/C Threads	Additional Requirements
Coarse	Single global lock	not distinct	
Coarse2	Single global lock	not distinct	rogues shoot at 2 random lanes
CoarseCleaner	Single global lock, conditions	distinct	conditions, wait/notify
Fine	Per lane locks	not distinct	
Fine2	Per lane locks	not distinct	rogues shoot at 2 random lanes
FineCleaner	Per lane locks, conditions	distinct	conditions, wait/notify
TM	TM	not distinct	
TM2	TM	not distinct	rogues shoot at 2 random lanes
TMCleaner	TM	distinct	

Table 1. The nine different rogue implementations required for the sync-gallery project. The technique column indicates what synchronization technique was required. The R/C Threads column indicates whether coordination was required between dedicated rogue and cleaner threads or not. A value of "distinct" means that rogue and cleaner instances run in their own thread, while a value of "not distinct" means that the last rogue to shoot an empty (white) lane is responsible for cleaning the gallery.

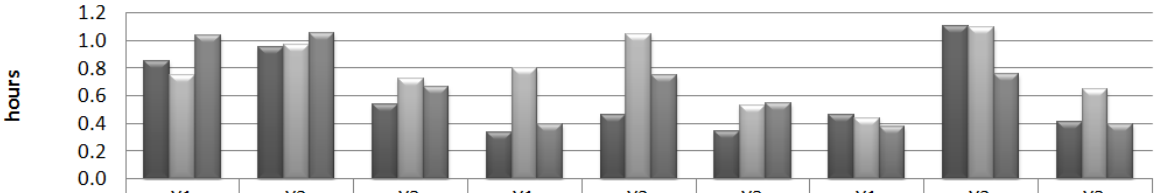
the frequency with which students made errors as another metric of the difficulty of various synchronization constructs.

To prevent experience with the assignment as a whole from influencing the difficulty of each task, we asked students to complete the tasks in different orders. In each group of rogues (single-lane, two-lane, and separate cleaner thread), students completed the coarse-grained lock version first. Students then either completed the fine-grained or TM version second, depending on their assigned group. We asked students to randomly assign themselves to groups based on hashes of their name. Due to an error, nearly twice as many students were assigned to the group completing the fine-grained version first. However, there were no significant differences in programming time between the two groups, suggesting that the order in which students implemented the tasks did not affect the difficulty of each task.

3.1 Limitations

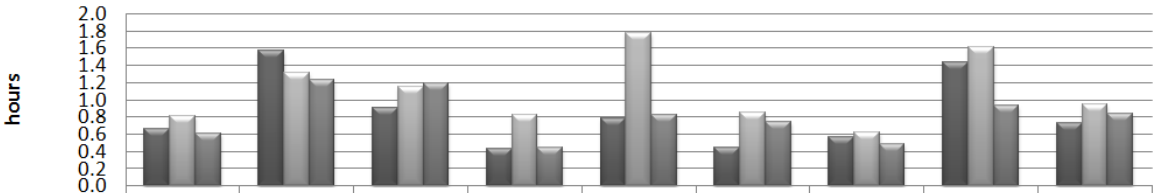
Perhaps the most important limitation of the study is the much greater availability of documentation and tutorial information about locking than about transactions. The novelty of transactional memory made it more difficult both to teach and learn. Lectures about locking drew on a larger body of understanding that has existed for a longer time. It is unlikely that students from year one influenced students from year two given the difference in concrete syntax between the two courses. Students from year two could have influenced those from year three, since the syntax remained the same from year two to year three.

Design



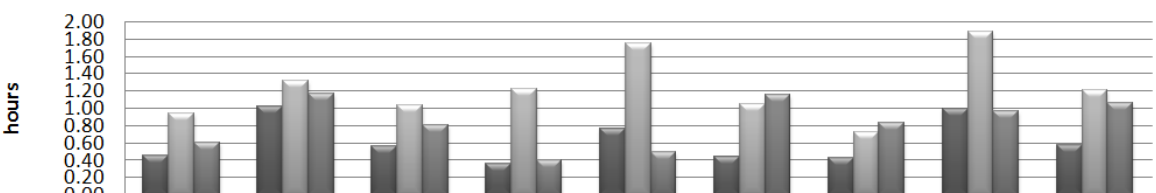
	Coarse			Fine			Cleaner		
	Y1	Y2	Y3	Y1	Y2	Y3	Y1	Y2	Y3
locks	0.85	0.95	0.54	0.34	0.46	0.35	0.47	1.10	0.41
conditions	0.75	0.97	0.72	0.80	1.04	0.53	0.44	1.09	0.64
TM	1.04	1.05	0.67	0.39	0.75	0.55	0.38	0.76	0.39

Code



	Coarse			Fine			Cleaner		
	Y1	Y2	Y3	Y1	Y2	Y3	Y1	Y2	Y3
locks	0.66	1.57	0.90	0.43	0.78	0.44	0.56	1.44	0.72
conditions	0.81	1.31	1.15	0.82	1.78	0.85	0.62	1.61	0.94
TM	0.61	1.23	1.19	0.44	0.83	0.74	0.48	0.93	0.84

Debug



	Coarse			Fine			Cleaner		
	Y1	Y2	Y3	Y1	Y2	Y3	Y1	Y2	Y3
locks	0.45	1.03	0.56	0.36	0.77	0.44	0.43	0.99	0.57
conditions	0.94	1.32	1.03	1.23	1.75	1.04	0.72	1.88	1.21
TM	0.61	1.17	0.81	0.40	0.49	1.15	0.83	0.96	1.06

Figure 3. Average design, coding, and debugging time spent for analogous rogue variations.

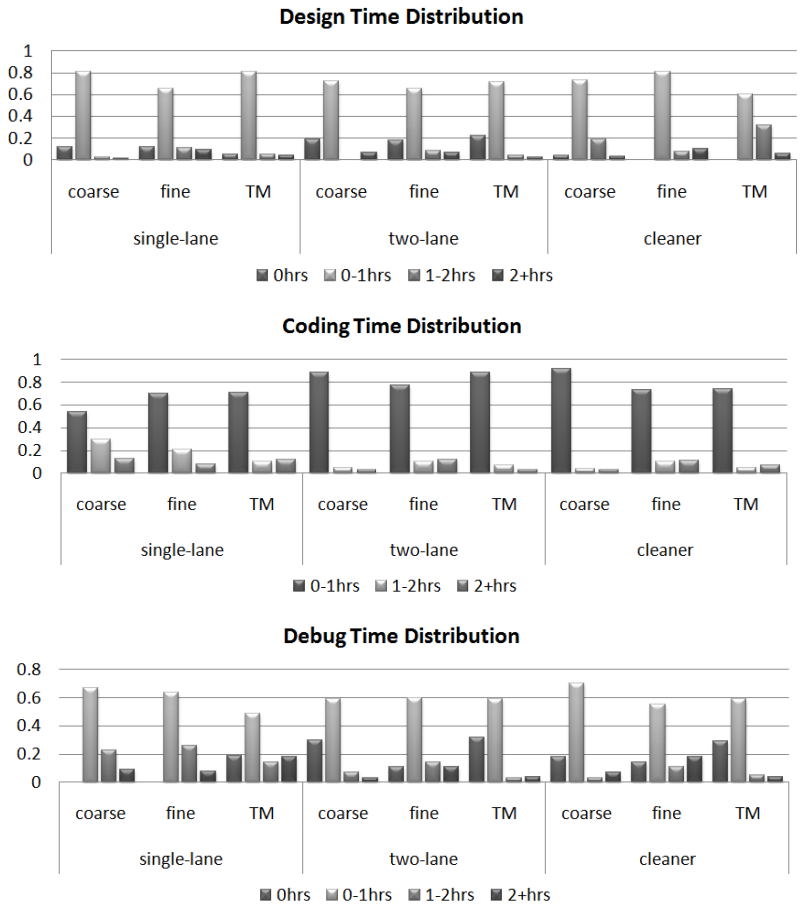


Figure 4. Distributions for the amount of time students spent coding and debugging, for all rogue variations.

Another important limitation is the lack of compiler and language support for TM in general, and the lack of support for the `atomic` keyword specifically. Because of this, programmers must directly insert read and write barriers and write exception handling code to manage retrying on conflicts. This yields a concrete syntax for transactions that is a a barrier to ease of understanding and use (see §4.2).

4. Evaluation

We examined development time, user experiences, and programming errors to determine the difficulty of programming with various synchronization primitives. In general, we found that a single coarse-grained lock had similar complexity to transactions. Both of these primitives were less difficult, caused fewer errors, and had better student responses than fine-grained locking.

4.1 Development time

Figure 3 shows the average time students spent designing, coding and debugging with each synchronization primitive, for all three years of the study. To characterize the diversity of time investment the students reported, Figure 4 shows the distribution of times students spent on coding and debugging. Figure 4 shows only data from year two: distributions for years one and three are similar.

On average, transactional memory required more development time than coarse locks, but less than what was required for fine-grain locks and condition synchronization. Coloring two lanes or using condition synchronization are more complex synchronization tasks than using coarse-grained locks. Debugging time increases more than design or coding time for these complex teaks. (Figure 3).

We evaluate the statistical significance of differences in development time in Table 2. Using a Wilcoxon signed-rank test, we evaluate the alternative hypothesis on each pair of synchronization tasks that the row task required less time than the column task. Pairs for which the signed-rank test reports a p-value of $< .05$ are considered statistically significant, indicating that the row task required less time than the column. If the p-value is greater than .05, the difference in time for the tasks is not statistically significant or the row task required more time than the column task. Results for the different class years are separated due to differences in the TM part of the assignment(Section 2.4).

We found that students took more time on their initial task, because they were familiarizing themselves with the assignment. Except for fine-grain locks, later versions of similar synchronization primitives took less time than earlier, e.g. the Coarse2 task took less time than the Coarse task. In addition, condition synchronization is difficult. For both rogues with less complex synchronization (Coarse and TM), adding condition synchronization increases the time required for development. For fine-grain locking, students simply replace one complex problem with a second, and so do not require significant additional time.

In years one and two, we found that coarse locks and transactions required less time than fine-grain locks on the more complex

two-lane assignments. This echoes the promise of transactions, removing the coding and debugging complexity of fine-grain locking and lock ordering when more than one lock is required. The same trend was not observable in year three.

4.2 User experience

To gain insight into the students' perceptions about the relative ease of using different synchronization techniques we asked the students to respond to a survey after completing the sync-gallery project. The survey ends with 6 questions asking students to rank their favorite technique with respect to ease of development, debugging, reasoning about, and so on.

A version of the complete survey can be viewed at [3].

In student opinions, we found that the more baroque syntax of the DSTM2 system was a barrier to entry for new transactional programmers. Figure 5 shows student responses to questions about syntax and ease of thinking about different transactional primitives. In the first class year (which used DSTM2), students found transactions more difficult to think about and had syntax more difficult than that of fine-grain locks. In the second year, when the TM implementation had a less cumbersome syntax, student opinions aligned with our other findings: TM ranked behind coarse locks, but ahead of fine-grain. For all three years, other questions on ease of design and implementation ranked TM ahead of fine-grain locks.

4.3 Synchronization Error Characterization

We examined the solutions from all three years in detail to classify the types of synchronization errors students made along with their frequency. This involved both a thorough reading of every student's final solutions and automated testing. While the students' subjective evaluation of the ease of transactional programming does not clearly indicate that transactional programming is easier, the types and frequency of programming errors does.

While the diversity of different errors we found present in the students' programs far exceeded our expectations, we found that all errors fit within the taxonomy described below.

1. **Lock ordering (lock-ord)**. In fine-grain locking solutions, a program failed to use a lock ordering discipline to acquire locks, admitting the possibility of deadlock.
2. **Checking conditions outside a critical section (lock-cond)**. This type of error occurs when code checks a program invariant with no locks held, and subsequently acts on that invariant after acquiring locks. This was the most common error in sync-gallery, and usually occurred when students would check whether to clean the gallery with no locks held, subsequently acquiring lane locks and proceeding to clean. The result is a violation of invariant 4 (§2). This type of error may be more common because no visual feedback is given when it is violated (unlike races for shooting lanes, which can result in purple lanes).
3. **Forgotten synchronization (lock-forgot)**. This class of errors includes all cases where the programmer forgot to acquire locks, or simply did not realize that a particular region would require mutual exclusion to be correct.
4. **Exotic use of locks (lock-exotic)**. This error category is a catch-all for synchronization errors made with locks for which

Year 1
Best syntax

Answers	1	2	3	4
Coarse	**69.6%**	17.4%	0%	8.7%
Fine	13.0%	**43.5%**	17.4%	21.7%
TM	8.7%	21.7%	21.7%	**43.5%**
Conditions	0%	21.7%	**52.1%**	21.7%

Easiest to think about

Answers	1	2	3	4
Coarse	**78.2%**	13.0%	4.3%	0%
Fine	4.3%	**39.1%**	34.8%	17.4%
TM	8.7%	21.7%	26.1%	**39.1%**
Conditions	4.3%	21.7%	30.4%	**39.1%**

Year 2
Best syntax

Answers	1	2	3	4
Coarse	**61.6%**	30.1%	1.3%	4.1%
Fine	5.5%	20.5%	**45.2%**	26.0%
TM	26.0%	**31.5%**	19.2%	20.5%
Cond.	5.5%	20.5%	28.8%	**39.7%**

Easiest to think about

Answers	1	2	3	4
Coarse	**80.8%**	13.7%	1.3%	2.7%
Fine	1.3%	**38.4%**	30.1%	28.8%
TM	16.4%	**31.5%**	30.1%	20.5%
Cond.	4.1%	13.7%	**39.7%**	**39.7%**

Year 3
Best syntax

Answers	1	2	3	4
Coarse	**72.2%**	22.2%	2.8%	2.8%
Fine	16.7%	**38.9%**	33.3%	11.1%
TM	8.3%	25%	25%	**41.7%**
Conditions	8.3%	13.9%	36.1%	**41.7%**

Easiest to think about

Answers	1	2	3	4
Coarse	**88.9%**	8.3%	0.0%	0.0%
Fine	0.0%	**44.4%**	33.3%	19.4%
TM	5.6%	27.8%	**33.3%**	30.8%
Conditions	2.8%	22.2%	27.8%	**44.4%**

Figure 5. Selected results from student surveys. Column numbers represent rank order, and entries represent what percentage of students assigned a particular synchronization technique a given rank (e.g. 80.8% of students ranked Coarse locks first in the "Easiest to think about category"). In the first year the assignment was presented, the more complex syntax of DSTM made TM more difficult to think about. In the second year, simpler syntax alleviated this problem.

		Coarse	Fine	TM	Coarse2	Fine2	TM2	CoarseCleaner	FineCleaner	TMCleaner
Coarse	Y1	1.00	**0.03**	**0.02**	1.00	**0.02**	1.00	0.95	0.47	0.73
	Y2	1.00	0.33	0.12	1.00	0.38	1.00	1.00	0.18	1.00
	Y3	1.00	0.06	0.43	1.00	0.17	0.60	0.93	**0.02**	0.61
Fine	Y1	0.97	1.00	0.33	1.00	0.24	1.00	1.00	0.97	0.88
	Y2	0.68	1.00	0.58	1.00	0.51	1.00	1.00	0.40	1.00
	Y3	0.94	1.00	0.66	1.00	0.70	0.99	0.99	0.35	0.94
TM	Y1	0.98	0.68	1.00	1.00	0.13	1.00	1.00	0.98	0.92
	Y2	0.88	0.43	1.00	1.00	0.68	1.00	1.00	0.41	1.00
	Y3	0.57	0.35	1.00	1.00	0.46	0.84	0.96	**0.03**	0.82
Coarse2	Y1	**<0.01**	**<0.01**	**<0.01**	1.00	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**<0.01**
	Y2	**<0.01**	**<0.01**	**<0.01**	1.00	**<0.01**	0.45	**<0.01**	**<0.01**	**<0.01**
	Y3	**<0.01**	**<0.01**	**<0.01**	1.00	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**<0.01**
Fine2	Y1	0.98	0.77	0.87	1.00	1.00	1.00	1.00	1.00	0.98
	Y2	0.62	0.49	0.32	1.00	1.00	1.00	0.99	0.59	1.00
	Y3	0.83	0.31	0.55	1.00	1.00	0.93	0.96	**<0.01**	0.69
TM2	Y1	**<0.01**	**<0.01**	**<0.01**	0.99	**<0.01**	1.00	**0.04**	**<0.01**	**<0.01**
	Y2	**<0.01**	**<0.01**	**<0.01**	0.55	**<0.01**	1.00	**<0.01**	**<0.01**	**<0.01**
	Y3	0.41	**0.02**	0.17	1.00	0.07	1.00	0.73	**<0.01**	0.40
CoarseCleaner	Y1	0.05	**<0.01**	**<0.01**	1.00	**<0.01**	0.96	1.00	**<0.01**	0.08
	Y2	**<0.01**	**<0.01**	**<0.01**	1.00	**<0.01**	1.00	1.00	**<0.01**	0.96
	Y3	0.07	**<0.01**	**0.04**	1.00	**0.05**	0.28	1.00	**<0.01**	0.34
FineCleaner	Y1	0.53	**0.03**	**0.02**	1.00	**<0.01**	1.00	0.99	1.00	0.46
	Y2	0.83	0.60	0.59	1.00	0.42	1.00	1.00	1.00	1.00
	Y3	0.98	0.66	0.97	1.00	0.99	1.00	1.00	1.00	0.99
TMCleaner	Y1	0.28	0.12	0.08	1.00	**0.03**	1.00	0.92	0.55	1.00
	Y2	**<0.01**	**<0.01**	**<0.01**	0.99	**<0.01**	1.00	**0.04**	**<0.01**	1.00
	Y3	0.40	0.06	0.19	1.00	0.32	0.60	0.67	**0.02**	1.00

Table 2. Comparison of time taken to complete programming tasks for all students. The time to complete the task on the row is compared to the time for the task on the column. Each cell contains p-values for a Wilcoxon signed-rank test, testing the hypothesis that the row task took less time than the column task. Entries are considered statistically significant when $p < .05$, meaning that the row task did take less time to complete than the column task, and are marked in bold. Results for the three class years are reported separately, due to differing transactional memory implementations.

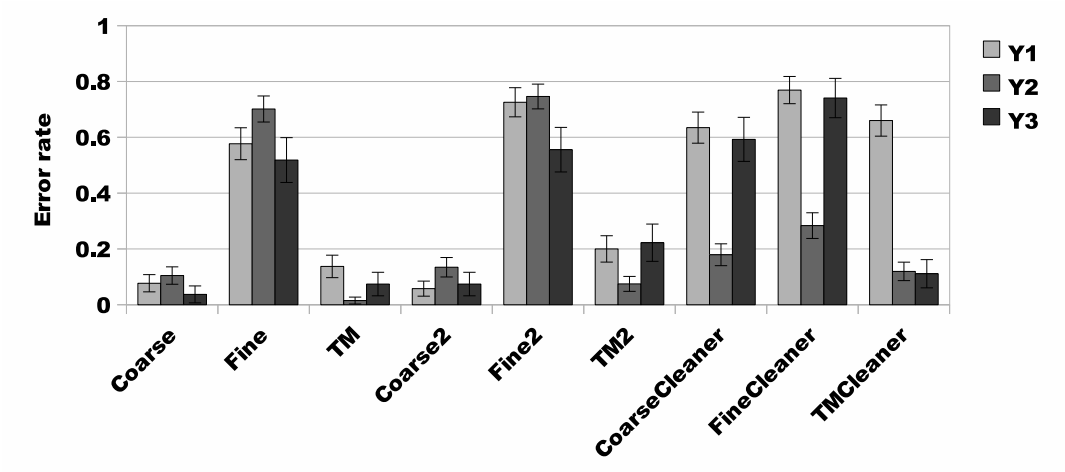


Figure 6. Overall error rates for programming tasks, for all three years of the study. Error bars show a 95% confidence interval on the error rate. Fine-grained locking tasks were more likely to contain errors than coarse-grained or transactional memory (TM).

we were unable to discern the programmer's actual intent. Programs that contribute a data point in this category were characterized by heavy over-use of additional locks.

5. **Exotic use of condition variables (cv-exotic)**. We encountered a good deal of signal/wait usage on condition variables that indicates no clear understanding of what the primitives actually do. The canonical example of this is signaling and waiting on the same condition in the same thread.
6. **Condition variable use errors (cv-use)**. These types of errors indicate a failure to use condition variables properly, but do indicate a certain level of understanding. This class includes use of `if` instead of `while` when checking conditions on a decision

to wait, or failure to check the condition at all before or after waiting.

7. **TM primitive misuse (TM-exotic)**. This class of error includes any misuse of transactional primitives. Technically, this class includes mis-use of the API, but in practice the only errors of this form we saw were failure to call `BeginTransaction` before calling `EndTransaction`. Omission of read/write barriers falls within this class as well, but it is interesting to note that we found no bugs of this form over all three years.
8. **TM ordering (TM-order)**. This class of errors represents attempts by the programmer to follow some sort of locking discipline in the presence of transactions, where they are strictly unnecessary. Such errors do not result in an incorrect program, but do represent a misunderstanding of the primitive.
9. **Checking conditions outside a transaction (TM-cond)**. This class of errors is analogous to the **lock-cond** class. It occurs when a programmer checks a condition outside of a transaction which is then acted on inside the transaction, and the condition is no longer guaranteed to hold during the transaction.
10. **Forgotten TM synchronization (TM-forgot)**. Like the forgotten synchronization class above (lock-forgot), these errors occur when a programmer failed to recognize the need for synchronization and did not use transactions to protect a data structure.

Because different rogue implementations use different synchronization techniques, a different subset of the error classes applies for each rogue. For example, it is not possible for a programmer to create a lock-ordering bug using a single coarse grain lock, so **lock-ord** does not apply for coarse rogues. Table 4 shows explicitly which error classes are applicable for each rogue implementation.

Table 3 shows the characterization of synchronization for programs submitted for all three years of the study. Figure 6 shows the overall portion of students that made an error on each programming task. Students were far more likely to make an error on fine-grain synchronization than on coarse or TM. At least 50% of students made at least one error on the Fine and Fine2 portions of the assignment. We believe error rates in year 2 are generally lower because the teaching assistants during this year were more active in helping students with the material and the assignment. In almost all cases, the error rates associated with TM implementations are dramatically lower than those associated with locks or conditions. The notable exception is the TM-based cleaner implementations from Year 1, where more than half of the solutions contained at least one error: the vast majority of errors were of type **TM-cond**. The bulk of these errors were similar, and arose when programmers checked whether lanes should be cleaned outside a transaction, subsequently starting a transaction to perform the cleaning. In years 2 and 3, where transaction support relied on JDASTM rather than DSTM2, this type of error was non-existent.

	coarse			fine			TM		
	single	**two**	**cleaner**	**single**	**two**	**cleaner**	**single**	**two**	**cleaner**
lock-ord				X	X	X			
lock-cond	X	X	X	X	X	X			
lock-forgot	X	X	X	X	X	X			
lock-exotic	X	X	X	X	X	X			
cv-exotic			X			X			
cv-use			X			X			
TM-exotic							X	X	X
TM-order							X	X	X
TM-forgot							X	X	X
TM-cond							X	X	X

Table 4. Indication for which synchronization errors are possible with each rogue assignment. The cell in the table contains an X if it is possible to make the given error using the synchronization primitives prescribed for that rogue implementation. Error types are explained in Section 4.3.

5. Code complexity

Table 5 shows code complexity statistics, and Figure 7 shows relative cyclomatic complexity and code size (in instructions) for all rogue classes (measured by the `cyvis` software complexity visualizer [1]). Data shown are for year 2. The cyclomatic complexity [21] of a fragment of code is the the number of independent control paths through it. Cyclomatic complexity does not *directly* capture important aspects of complexity in a multi-threaded programming environment. However, in this study, the programming tasks remain constant, and the bulk of variation from solution to solution can be attributed to synchronization implementation. Hence, cyclomatic complexity is a reasonable metric for understanding additional differences in code complexity across different synchronization techniques.

The cyclomatic complexity data corroborate what we observed in the students' surveys. In years two and three, solutions based on coarse grain locking have the lowest average cyclomatic complexity, while TM-based solutions have lower complexity than fine-grain and cleaner solutions. For example, in year 2, the single-lane, two-lane, and cleaner rogues show complexity of 2.9, 3.6, and 2.8 respectively. The solutions synchronizing with TM show slightly

		lock-ord	**lock-cond**	**lock-forgot**	**lock-exotic**	**cv-exotic**	**cv-use**	**TM-exotic**	**TM-order**	**TM-forgot**	**TM-cond**
year 1	**occurrences**	23	84	39	16	12	32	0	5	8	41
	opportunities	51	312	312	312	52	52	153	153	149	149
	rate	45.1%	29.6%	12.5%	5.1%	23.1%	61.5%	0.0%	3.7%	5.4%	27.5%
year 2	**occurrences**	11	62	26	0	11	14	5	4	1	0
	opportunities	134	402	402	402	134	134	201	201	201	201
	rate	8.2%	6.5%	15.4%	0%	8.2%	10.5%	2.5%	2.0%	0.5%	0%
year 3	**occurrences**	5	41	5	0	22	12	6	0	3	0
	opportunities	28	168	168	168	56	56	84	84	84	84
	rate	17.9%	24.4%	3.0%	0%	39.3%	21.4%	7.1%	0%	3.6%	0%

Table 3. Synchronization error rates for all three years. The occurrences row indicates the number of programs in which at least one bug of the type indicated by the column header occurred. The **opportunities** row indicates the sample size (the number of programs we examined in which that type of bug could arise: e.g. lock-ordering bugs cannot occur in with a single coarse lock). The **rate** column expresses the percentage of examined programs containing that type of bug. Bug types are explained in Section 4.3.

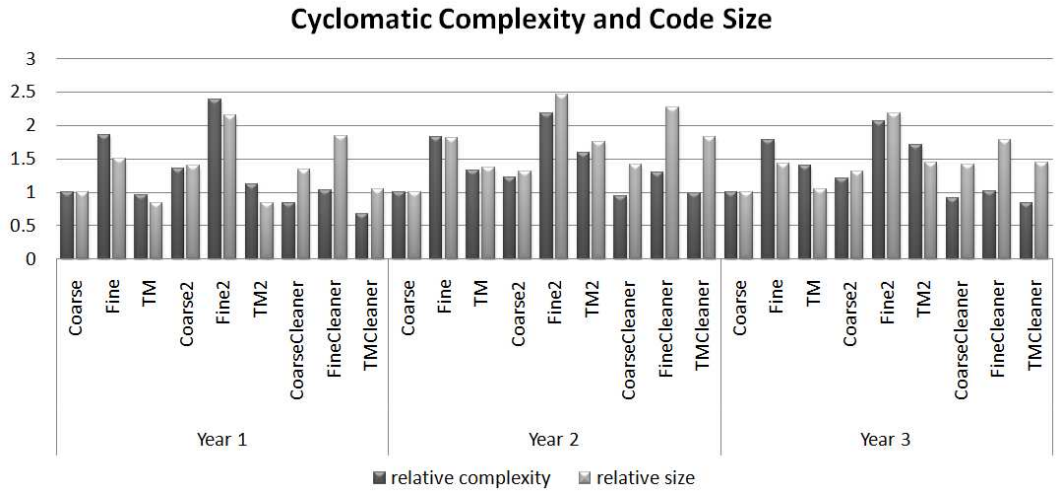


Figure 7. Cyclomatic complexity and code size relative to RogueCoarse for solutions from all years.

year	rogue name	complexity mean	complexity std	size mean	size std
1	Coarse	2.8	0.2	114.4	1.5
	Fine	5.2	0.3	172.8	7.6
	TM	2.7	1.8	96.7	35.3
	Coarse2	3.8	0.2	160.9	11.6
	Fine2	6.8	0.3	247.2	5.0
	TM2	3.2	2.3	96.3	86.2
	CoarseCleaner	2.4	0.2	153.1	10.6
	FineCleaner	2.9	0.1	211.3	9.1
	TMCleaner	1.9	0.7	120.3	60.0
2	Coarse	2.9	0.8	104.5	25.2
	Fine	5.4	2.0	189.4	50.5
	TM	3.9	1.1	142.9	29.0
	Coarse2	3.6	0.9	137.2	32.1
	Fine2	6.4	2.5	257.9	59.6
	TM2	4.7	1.3	183.4	43.0
	CoarseCleaner	2.8	0.6	147.6	23.8
	FineCleaner	3.8	1.0	237.1	51.2
	TMCleaner	2.9	0.6	190.9	30.2
3	Coarse	2.9	0.2	116.5	1.6
	Fine	5.3	0.7	167.1	6.0
	TM	4.1	1.1	122.4	30.9
	Coarse2	3.5	0.4	153.4	12.9
	Fine2	6.1	1.2	254.5	36.5
	TM2	5.0	1.4	168.8	47.2
	CoarseCleaner	2.7	0.2	165.4	4.1
	FineCleaner	3.0	0.1	207.7	16.7
	TMCleaner	2.4	0.4	168.5	39.4

Table 5. Cyclomatic complexity and code size measurements for all rogue implementations for all three years of the study.

higher complexity than their coarse counterparts with values of 3.9, 4.7, and 2.9. However, TM solutions have noticeably lower complexity than solutions based on fine-grain locks, which, in year 2 show average complexity of 5.4, 6.4, and 3.8. The same trends are echoed by the average instruction counts for the various rogue solutions. Year 1 is an exception to this trend in that TM solutions showed lower complexity than coarse or fine-grain locks. This is largely because DSTM2 syntax does not require the programmer to provide exception handling and retry code to handle aborts due to conflict. The boiler-plate exception handling code required by JDASTM inflates the complexity measurement in years 2 and 3.

6. Related work

Hardware transactional memory research is an active research field with many competing proposals [5–8, 11, 12, 16–18, 22, 23, 26–28, 32, 34]. All this research on hardware mechanism is premature if researchers never validate the assumption that transactional programming is actually easier than lock-based programming.

This research uses software transactional memory [4, 10, 13–15, 19, 20, 31, 33]), but its purpose is to validate how untrained programmers learn to write correct and performant concurrent programs with locks and transactions. The programming interface for STM systems is the same as HTM systems, but without compiler support, STM implementations require explicit read-write barriers, which are not required in an HTM. Compiler integration creates a simpler programming model than using a TM library [9]. Future research could investigate whether compiler integration lowers the perceived programmer difficulty in using transactions.

There is a previous version of this study [30]. This version extends that study bringing a third year of survey results and two additional years worth of student programs into the data set. Pankratius et al. [25] describe a study in which 12 students working in 6 teams of two, wrote a parallel search engine over ten course of a 15 week project. Three teams used Pthreads, and three teams used the Intel STM compiler [24]. Like this study, the Pankratius study evaluates the resulting code, and surveys the developers involved to understand their experience. Their findings are complementary to ours: development investment required for the TM implementations in the study was lower. The study in this paper involves 20× more programmers, and 200× more programs are evaluated. In our own study, blocking and non-blocking synchronization, as well as coarse and fine-grain locking are explicitly considered.

7. Conclusion

This paper offers evidence that transactional programming really is less error-prone than high-performance locking, even if inexperienced programmers have some trouble understanding transactions. Students' subjective evaluation showed that they found transactional memory slightly harder to use than coarse locks, and easier to use than fine-grain locks and condition synchronization. However,

analysis of synchronization error rates in students' code yields a more dramatic result, showing that for similar programming tasks, transactions are considerably easier to get correct than locks.

References

[1] *Cyvis Software Complexity Visualizer*, 2009.

[2] *Spoon*, 2009. `http://spoon.gforge.inria.fr/`.

[3] *Sync-gallery survey: http://www.cs.utexas.edu/users/witchel/tx/sync-gallery-survey.html*, 2009.

[4] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana 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.

[5] Lee Baugh, Naveen Neelakantam, and Craig Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. *SIGARCH Comput. Archit. News*, 36(3):115–126, 2008.

[6] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. *SIGARCH Comput. Archit. News*, 35(2):24–34, 2007.

[7] Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. Tokentm: Efficient execution of large transactions with hardware transactional memory. *SIGARCH Comput. Archit. News*, 36(3):127–138, 2008.

[8] JaeWoong Chung, Chi Cao Minh, Austen McDonald, Travis Skare, Hassan Chafi, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. Tradeoffs in transactional memory virtualization. *SIGPLAN Not.*, 41(11):371–381, 2006.

[9] Luke Dalessandro, Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. Capabilities and limitations of library-based software transactional memory in c++. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*. Portland, OR, Aug 2007.

[10] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, 2006.

[11] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. *SIGPLAN Not.*, 44(3):157–168, 2009.

[12] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, 2004.

[13] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM.

[14] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 14–25, New York, NY, USA, 2006. ACM.

[15] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 253–262, New York, NY, USA, 2006. ACM.

[16] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[17] Owen S. Hofmann, Christopher J. Rossbach, and Emmett Witchel. Maximum benefit from a minimal htm. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 145–156, New York, NY, USA, 2009. ACM.

[18] Yossi Lev and Jan-Willem Maessen. Split hardware transactions: true nesting of transactions using best-effort hardware transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 197–206, New York, NY, USA, 2008. ACM.

[19] Yossi Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased transactional memory. In *Workshop on Transactional Computing (TRANSACT)*, 2007.

[20] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of software transactional memory. Technical Report TR 893, Computer Science Department, University of Rochester, Mar 2006. Condensed version submitted for publication.

[21] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, 1976.

[22] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. *SIGARCH Comput. Archit. News*, 34(2):53–65, 2006.

[23] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. Feb 2006.

[24] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin 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.

[25] Victor Pankratius, Ali-Reza Adl-Tabatabai, and Frank Otto. Does transactional memory keep its promises? results from and empirical study. publication, September 2009.

[26] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, 2005. IEEE Computer Society.

[27] Hany E. Ramadan, Christopher J. Rossbach, Donald E. Porter, Owen S. Hofmann, Aditya Bhandari, and Emmett Witchel. Metatm/txlinux: transactional memory for an operating system. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 92–103, New York, NY, USA, 2007. ACM.

[28] Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel. Dependence-aware transactional memory for increased concurrency. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 246–257, Washington, DC, USA, 2008. IEEE Computer Society.

[29] Hany E. Ramadan, Indrajit Roy, Maurice Herlihy, and Emmett Witchel. Committing conflicting transactions in an stm. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 163–172, New York, NY, USA, 2009. ACM.

[30] Christopher Rossbach, Owen Hofmann, and Emmett Witchel. Is transactional memory programming actually easier? In *WDDD '09: Proc. 8th Workshop on Duplicating, Deconstructing, and Debunking*, jun 2009.

[31] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, Aug 1995.

[32] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*. Jun 2008.

[33] Fuad Tabba, Cong Wang, James R. Goodman, and Mark Moir. NZTM: Nonblocking, zero-indirection transactional memory. In *Workshop on Transactional Computing (TRANSACT)*, 2007.

[34] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 261–272, Washington, DC, USA, 2007. IEEE Computer Society.