Software Transactional Memory vs. Locking in a Functional Language

A Controlled Experiment

Fernando Castor, João Paulo Oliveira, and André L. M. Santos
Informatics Center, Federal University of Pernambuco, Recife, Brazil
{castor,jpso,alms}@cin.ufpe.br

Abstract

Many researchers in industry and academia believe that software transactional memory (STM) will play an important role in the transition from single-core to multicore systems. However, little effort has been placed on assessing whether STM delivers on its promises of avoiding common concurrent/parallel programming pitfalls.In this paper, we describe a controlled experiment aiming to evaluate the ease of using STM. The study targets Haskell, a purely functional programming language that includes a mature implementation of STM. It compares the use of STM and Haskell's lock-based concurrency control mechanism to develop a program with (coarse-grained) mutual exclusion and synchronization requirements. We organized the 51 subjects of the study in two groups, one for each technique. We found out that the two techniques did not differ significantly in terms of concurrency errors, number of LoC and time to develop the resulting programs. However, for programs where developers made only non-concurrency-related mistakes, STM programmers finished their assignments quicker.

1. Introduction

The transition from single core to multicore machines requires developers to rethink the ways in which they build software systems. This paradigm shift creates a plethora of opportunities for researchers, who must learn to balance correctness and performance requirements in the construction of applications that are not best fits for parallel execution. Many researchers believe that transactional memory [4], in general, and software transactional memory (STM) [12][1], in particular, will play an important part in the transition to multicore systems. However, few studies have attempted to discover whether STM delivers on its promises of avoiding common concurrent/parallel programming pitfalls.

In this paper, we describe a controlled experiment aiming to evaluate the ease of using STM. The study involved 51 undergraduate subjects programming in Haskell, a purely functional programming language that includes an implementation of STM in its mainstream distribution.

Functional programming, analogously to STM, is being touted by software developers [13] and researchers as an important tool to overcome the complexity of building multicore applications.Notwithstanding, even though there are proposals to combine the two approaches [3, 5], we are not aware of any evaluation of the resulting combination.This is surprising since the only industrial-strength languages we are aware of that include STM as part of their mainstream distributions are functional. We compare the ease of using Haskell's STM and its lock-based concurrency control mechanism to develop a simple program with (coarse-grained) mutual exclusion and synchronization requirements. Unlike most STMs, Haskell's does not require alternative mechanisms for condition-based synchronization, such as condition variables.

To evaluate the two approaches, we have examined the errors in the resulting programs and measured the number of lines of code and the time it took for each subject to finish. We found out that the difference between the two techniques in terms of the concurrency errors, number of LoC, and time spent was not statistically significant. Nonetheless, for programs with no concurrency-related errors, STM subjects finished their assignments quicker. Besides discussing these results, we present a classification of the types of errors that we have found in the programs.

This paper makes the following contributions:

- An experimental setup that other researchers can use to assess the ease of using STM when compared to a different concurrency control
- Data[2] and observations from an experiment involving 51 subjects using STM and a lock-based approach for concurrency control.

[1] In this paper, we assume the reader is familiar with the basics of STM.

[2] All the data for this study is available at http://sites.google.com/a/cin.ufpe.br/castor/tmc_2011

Copyright is held by the author/owner(s). This paper was published in the proceedings of the Workshop on Transitioning to MultiCore (TMC) at the ACM Systems, Programming, Languages and Applications: Software for Humanity (SPLASH) Conference, October, 2011. Portland, OR, USA.

- A discussion of the types of problems that we found in the programs that the subjects have produced. technique.

2. Haskell and Transactional Memory

Haskell is a purely functional language that implements a number of different approaches for concurrent and parallel programming [6], such as implicit parallelism, threads communicating by means of locking mutable variables, map-reduce for data-centric parallel computations, and STM.

An important concept in Haskell is Monad functions. A Monad is a kind of abstract data type constructor that encapsulates computations.Monads provide a bridge between purely functional computations and operations that have side effects, isolating the former from the latter. Haskell programs use of monads in computations involving I/O, exceptions, concurrency, etc. A programmer can compose monadic functions specifying the order in which these functions are called. Threads in Haskell run within the `IO` monad. STM runs with the `STM` monad.

The basic concurrency primitive of Haskell is the `MVar`. An `MVar` can be thought of as a box that is either empty or full. Haskell provides two main operations to work with `MVars`: (i) `takeMVar` removes a value from a full box, returning it within the `IO` monad; and (ii) `putMVar` puts a value in an empty box. These operations are blocking. An attempt to call `takeMVar` with an empty `MVar` blocks the calling thread. The same applies to `putMVar` and a full `MVar`. `MVars` are higher-level abstractions than simple locks. They combine locking and condition-based synchronization.

Haskell's mainstream distribution includes support for STM since 2005. The Haskell STM [3] provides the `TVar` type to implement mutable variables that only transactions manipulate. The use of monads does not allow manipulation of non-transactional variables within transactions (and vice-versa). Functions `readTVar` and `writeTVar` return and modify the value of `TVars`, respectively. To run transactionally, a piece of code must be executed by the `atomically` function. This function bridges the `IO` and `STM` monads, thus allowing STM operations to be executed by threads. It works like the `atomic` block [4, 12] and, hereafter, we refer to it as the `atomically` *block*. Haskell's STM supports condition-based synchronization using the `retry` function. The latter aborts the transaction when it is called and tries again from the start. Figure 2 (Section 4) provides an example.

3. Study Setting

In this section we present study setting: its subjects, the specification of the programs they implemented, how we collected data, our hypotheses, and threats to validity.

Subjects. The subjects of this study were undergraduate students undergoing a programming course on functional and concurrent/parallel programming. The course has two parts. The first one deals with functional programming in Haskell. It comprises 18 hours of in-class training plus a number of home assignments, a test, and a small project. In the second part of the course, the students received 16 hours of in-class training on concurrent and parallel programming, ranging from basic concepts (e.g., races, deadlocks, etc.) to advanced techniques (e.g., STM). It also comprises a test and many home assignments. For the two parts, the students received support from 8 teaching assistants.

The study had 51 subjects[3]. It was conducted as one of the tests of the course. Each subject was randomly assigned to one of two groups. In one group (STM group), we only allowed the subjects to use STM to implement the specified program, except for one special case, discussed in Section 4.1. In the other one (MV group), they could only employ mutable variables (`MVars`). The STM group received 25 subjects, whereas the MV group received 26 subjects. All the subjects took part in the experiment at the same time. During the experiment, each subject had access to a computer and was allowed to use a text editor and a Haskell compiler (GHC). The subjects could not communicate among themselves, nor look at references such as books and the Internet. Two authors of this paper, as well as 5 of the TAs, were available to answer questions.

Assignment. We can summarize the assignment as follows:

Due to heavy rain, a man was experiencing serious leaks in his house. Even after he repaired most of them, that still left three leaks dripping between 10 and 100 times each time they dripped. Below each leak, he put a bucket that supports a volume equivalent to 5000 drops. However, on a particularly stormy night, he saw the need to empty the buckets before they overflowed. To do so, he uses a mug able to remove 500 drops every time he dips it and throws the contents of the mug out the window. He choses the bucket from which he will remove water randomly. Also, while he is removing water from each bucket, not a drop falls between the time he puts the mug in the bucket and when he takes it out. Similarly, if a bucket becomes full, the water simply overflows to the floor and the bucket remains full (keeping their 5000 drops).

Implement a simulation of this situation considering that the man corresponds to a thread and that each leak is also implemented by a thread. The man should randomly choose the bucket from which to remove water. After removing the water, the man choses a new bucket and repeats the procedure. If the chosen bucket does not have at least 500 drops, he waits until this occurs without changing the bucket. The simulation ends when ***exactly*** *one million drops have fallen into the buckets (drops that fell out when the buckets when they were full do not count). Your program must have a* `main` *function and must follow all the recommendations presented above. It must also be compilable and should not enter deadlock or exhibit race conditions.*

The assignment also required the subjects to use either `MVars` or `TVars` to represent the buckets and store the overall number of fallen drops, based on the group of the subject.

Data Collection. Initially, for each subject, start and end time of the experiment were recorded. Each subject had

[3] It started out with 55, but 4 of them gave up.

at most three hours to finish the test. Afterwards, a single author, to ensure uniformity, graded all the tests.

Grading comprised three steps. For each step, the grading author recorded all the kinds of problems that he found (Section 4). **Step 1**. The author first attempted to compile the program[4]. If that failed, he examined the sources of the errors to determine if they were syntactic or semantic and skipped the second step. **Step 2**. After compilation, the author executed the program[5] to check if it ended and inspected any debugging information that the program provided. For each program, the author waited up to a minute before interruption. In general, correct programs finished their execution almost instantaneously. **Step 3**. Finally, the author would inspect the source code and look for errors of three kinds: errors that caused the program to hang, such as deadlocks, infinite loops ("hanging errors"), concurrency errors that do not cause the program to hang, such as race conditions ("non-hanging errors"), and logic errors not related to concurrency.

Using the `wc -l` command of UNIX, we measured the number of lines of code (LoC) of all the programs. Furthermore, we asked the subjects to answer a survey with questions such "What technique is the easiest to reason about?". We also asked them to estimate the time they spent implementing and debugging. Due to space constraints, we do not present the results of the survey in this paper. However, we make them available at the companion website[6].

Variables and Hypotheses. The independent variable of this study is the technique that each subject employed, either STM or mutable variables. The dependent variables are the kinds of errors that each subject committed, the number of LoC of the programs, and the time they spent developing these programs. Let NL be the number of LoC of a program, T the time spent by a subject to finish the assignment, and NE the number of types of errors (compilation, hanging, non-hanging, logic) that each subject committed. Also, PC denotes the presence or absence of compilation errors in a program (0 for absence, 1 for presence), PH denotes the presence or absence of hanging errors, PN denotes the presence or absence of non-hanging errors, and PL denotes the presence or absence of logic errors. Finally, NLC and TC correspond to NL and T, respectively, considering only programs for which PC, PH, and $PN = 0$. Given these definitions, the null hypotheses (composed as a single statement) are the following:

Null hypothesis (H01..9): The average values of (1) NL, (2) T, (3) NE, (4) PC, (5) PH, (6) PN, (7) PL, (8) NLC, and (9) TC when using STM are not significantly different from the values of the same metrics when using mutable variables.

Threats to Validity Empirical studies in software engineering are a difficult endeavor. Since they involve humans, a number of factors can affect the validity of an otherwise well-designed study. This study is no exception.

[4] Using `ghc --make -threaded -rtsopts filename.hs`.

[5] Using `./filename +RTS -N2`.

[6] http://sites.google.com/a/cin.ufpe.br/castor/tmc_2011

The subjects of the study were undergraduate students who were novices to concurrent programming, Haskell, and STM. On the one hand, this means that they are not as resourceful and experienced on software development as a real developer. Hence, if performed with actual developers this study might yield different results. On the other hand, they had no prior contact with concurrent programming. Therefore, we can say that they are free of bias against STM because, unlike an experienced developer, they had no a priori familiarity with locks.

Another threat to validity is the grading process, which was manual. We tried to minimize bias and promote uniformity by only considering the presence or absence of errors of a certain kind, instead of counting occurrences of each type of error. Counting would create grounds for different interpretations of what represents one occurrence and therefore hinder reproducibility of the results. Furthermore, to reduce subjectiveness, the same author graded all the programs.

4. Study Results

This section presents the results of the study. We start out by presenting the errors that we encountered in the programs that the subjects produced. Afterwards, we discuss the time spent by the subjects and the sizes (in number of LoC) of the resulting programs.

4.1 Errors

Table 1 presents the overall results of the experiment. Each row provides information about a program produced by a subject, using the technique indicated by the first column. Columns 2-5 points out occurrences of errors of each kind, specifically marking the types of errors we have found. The key below the table provides a classification of types of errors that may occur when one uses Haskell's mutable variables and transactional memory. The organization of the error types is in accordance with the four kinds of error of Section 3: compilation (PC), hanging (PH), non-hanging (PN), and logic (PL) errors.

Figure 1 presents the number of occurrences of each kind of error considering the two techniques. When comparing the numbers of occurrences errors (of all kinds) for the two techniques, the difference is not statistically significant (using the double-tailed Student's T-Test for p-value < 0.05). Therefore, we cannot reject hypotheses H3-H7.

Logic and Compilation Errors. The number of occurrences of logic errors was similar for the two techniques. This is expected, since these errors are not related to concurrency control. On the other hand, STM programs exhibited compilation errors more than twice more frequently than MV programs. Most of these errors were in some way related to monads, for example, due to an attempt to retry a transaction out of the `STM` monad. A number of authors (e.g., Bracha [2]) have commented on the complications that monads create. Threads and functions manipulating mutable variables run within the `IO` monad. The data we gathered suggests that inexperienced programmers might have diffi-

Technique	Compilation errors	Hanging errors	Non-hanging errors	Logic errors	Time (mins.)	# LoC
MV					174	165
MV				PL1, PL6	137	125
MV		PH2, PH3			52	142
MV				PL1	111	127
MV			PN1	PL2	95	128
MV					143	158
MV					94	140
MV		PH2	PN1, PN6	PL1, PL2	107	175
MV		PH3		PL1, PL2	108	180
MV				PL1, PL2	74	142
MV				PL6	109	127
MV		PH2			173	120
MV			PN5		87	120
MV	PC1	PH2	PN1	PL2	111	180
MV		PH2, PH3		PL2	100	135
MV		PH1			160	175
MV					130	126
MV	PC2			PL2	115	180
MV					75	180
MV	PC1, PC2		PN6	PL1,PL2,PL3	160	75
MV		PH2	PN6		100	170
MV		PH2	PN6	PL1	87	170
MV			PN1, PN6	PL2	127	90
MV					101	125
MV					163	180
MV					111	120
STM					136	140
STM		PH3	PN2	PL2, PL5	217	180
STM	PC1, PC2	PH3	PN2, PN3		143	130
STM			PN2, PN3	PL2	134	162
STM				PL5	65	134
STM			PN2	PL4	99	180
STM					101	132
STM	PC2	PH4			83	167
STM					119	150
STM					68	120
STM		PH3		PL5	99	105
STM	PC1				157	120
STM	PC2	PH4			95	180
STM	PC2		PN2		70	145
STM	PC2		PN2	PL1, PL2	111	180
STM					92	113
STM					77	103
STM				PL2	82	109
STM	PC2		PN2	PL2, PL3	110	173
STM	PC2	PH4		PL1	114	180
STM			PN6	PL3	108	115
STM			PN4		93	80
STM		PH2, PH3	PN1	PL2	191	100
STM					96	80
STM		PH3	PN2	PL2	96	180

PC1 syntax error
PC2 semantic error

PH1 circular dependency
PH2 non-matching putMVar and takeMVar
PH3 infinite recursion
PH4 entire recursion running within a single transaction (livelock)

PN1 race condition with MVar (fine locking or non-locking access)
PN2 race condition due to too fine-grained transactions
PN3 busy wait instead of retry
PN4 uses a global lock even though the solution is based on transactions
PN5 nonsensical results due to unknown causes
PN6 main thread finishes before the program is done

PL1 updates the total number of drops and the number in the bucket inconsistently
PL2 does not correctly limit the maximum number of drops
PL3 does not randomly generate the number of drops
PL4 uses the wrong variable type (e.g., an MVar when should be using a TVar)
PL5 very specific logic errors
PL6 employed a monad-unsafe operation

Table 1. Overall results of the experiment and the list of types of problems of each kind.

culty in conciliating this with transactions running within the `STM` monad.

Hanging Errors. Surprisingly, STM programs exhibited hanging errors almost as often as MV programs. Out of the 9 MV programs with hanging errors, 8 deadlock (PH1 or PH2). Among the latter, 7 deadlock due to non-matching `putMVar` and `takeMVar` operations. For one of the programs, the deadlock would never happen in practice because it does compile. However, manual inspection of the code revealed the (in this case, potential) problem. Among the STM programs, 8 exhibit hanging errors. For four of them, it was only possible to find this out by inspecting the source code because the programs did not compile. Most of the hanging errors in programs that use STM stem from one of two reasons: (i) race conditions stemming from too fine-grained

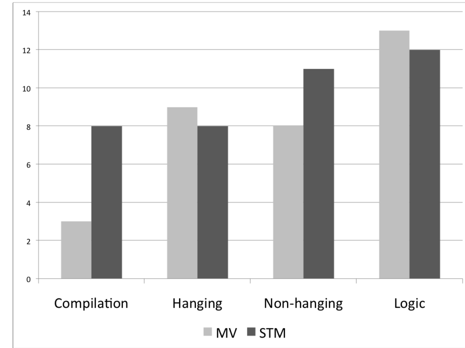

Figure 1. Number of occurrences of each kind of error, for each technique.

transactions (PH3); and (ii) inability to make progress due to transactions that are too course-grained (PH4). In the former case, the race condition may cause the counter that keeps track of the overall number of drops to become greater than 1,000,000 or a bucket to hold more than 5,000 or less than 0 drops. In the latter, the subject places a recursive call in the `atomically` block, leading to a livelock where transactions fail indefinitely, as described below:

1. Thread Man starts and choses bucket 1, which is empty.
2. Thread $Leak$ starts and fills bucket 1 with water. It is executed recursively and the recursive call appears within the `atomically` block.
3. Thread Man keeps retrying until bucket 1 has more than 500 drops.
4. Thread Man keeps failing after the bucket reaches 500 drops because thread $Leak$ runs within a single transaction.
5. The buckets reaches 5000 drops and thread $Leak$ retries. When that happens, the bucket reverts to 0 drops. The process restarts from step 1.

To evade the aforementioned problem, 14 out of the 25 STM subjects employed a double-checking idiom that Figure 2 summarizes. Although double-checking is known to have problems in some languages when combined with locking [1], we do not know if it leads to similar problems in Haskell. All that we can say is that we did not find problems by running the programs. To settle this matter, we looked for a specification of Haskell's memory model, but could not find it. Some recent mailing list discussions [7] indicates that this documentation is not publicly available. In addition, the need to create a second, short-lived transaction imposes an unnecessary overhead. Notice that this is only necessary because it is not possible to create a transaction whose implementation is not a block [14][7]. To avoid creating a separate transaction and reduce the risk of races or instruction re-ordering, the language could include a simple construct to perform a "tail commit", analogous to tail recursion: if a recursive call is the last operation that a transaction performs, the programmer should be allowed to state that it commits right before making the recursive call.

[7] Strictly speaking, in Haskell the implementation of a transaction is in fact an expression, not a block.

```
1 leak bucket drops ... =
2   do ...
3     atomically (do
4       total <- readTVar drops;
5       if (total < 1000000) then do {
6         ...
7         if (volBucket < 5000)
8         then do ...  -- add drops to the bucket
9         else retry;  -- fail now. Try again later.
10        ...
11      } else return (); })
12    nTot <- atomically (readTVar drops);
13    if (nTot < 1000000) then leak bucket drops ...
14    else ...        --got to 1000000 drops. Finish.
```

Figure 2. Double-checking idiom that most subjects using STM employed

```
1 f <- atomically (readTVar tDrops)
2 if (f < 1000000) -- RACE! tDrops might have changed.
3 then do atomically (writeTVar tDrops f) --loses the
      new value of tDrops, if it changed
4      ...
```

Figure 3. Transactions that are too fine-grained.

Non-hanging Errors. Subjects using STM produced a larger number of non-hanging errors than MV subjects. 11 STM programs exhibited some kind of non-hanging error. In 8 of them, transactions were too fine-grained, allowing race conditions to happen, as illustrated in Figure 3. In addition, three programs had errors that do not corrupt their results but affect their performance. Two subjects employed busy wait instead of `retry`. Moreover, a single STM subject used an `MVar` as a global lock, thus serializing program execution.

Amongst the MV programs, 8 exhibited non-hanging errors. In five of them the main thread does not wait for the other threads to finish and ends program execution too early. It is not clear why the MV subjects committed this error so many times when it only occurred once among the STM subjects. Even for this small sample, the difference is significant with 90% confidence. Furthermore, 4 MV programs had race conditions involving mutable variables. The use of the `readMVar` function was the chief cause for this problem. This function reads the value of an `MVar` without removing it. Hence, its use is analogous to, in a language like C or Java, reading the value of a variable in a critical section without enforcing mutual exclusion. Function `takeMVar` avoids this by locking access to the `MVar` until its contents are put back.

One of the STM programs has a race condition due to too fine-grained locking, the only PN1 error in an STM program. During the experiment, we allowed all the subjects to use a single mutable variable to stop the main thread from finishing before the other ones. We made this exception because they were familiar with an idiom based on this approach. The race is associated with this variable. We emphasize that this exception did not bias the results: only one monad problem occurred because of it and MV programs manifested most of the problems involving (not) stopping the main thread.

Technique	Measure	Mean	Std. Dev.	Median	Min.	Max
MV	LoC	115.54	31.83	110	52	174
	Time	144.42	29.51	141	75	180
STM	LoC	110.24	36.63	99	65	217
	Time	138.32	33.33	134	80	180

Table 2. Statistics for time spent and number of LoC.

4.2 Time Spent and Number of LoC

Similarly to errors, the time spent developing the programs and the number of LoC of the latter did not differ significantly among the two techniques. Table 2 shows that, in fact, the results for time spent and number of LoC were very similar. This similarity made us question whether it is uniform among the subjects, considering that some of them performed very well (their programs had no concurrency-related errors) whereas others had poor performance. Thus, we gathered the time and number of LoC for programs with no concurrency-related errors. We then verified whether they differed, using a T-test with 95% confidence. For time spent, we obtained a p-value of 0.0292, thus rejecting hypothesis H9. STM subjects spent, on the average, spent 16% less time than MV subjects. For number of LoC, the p-value was 0.0562, not enough to reject hypothesis H8, but by a narrow margin. STM programs had 21% less LoCs.

Given the difference in performance between the best STM subjects and the remaining ones, we conjecture that STM subjects spent more time debugging than MV subjects. Our rationale is that students who "got it right" from the outset were quicker when using STM. On the other hand, confusion about issues such as transaction granularity and the semantics of `retry` complicate debugging activities and account for the extra time. To gather evidence to support this conjecture, we examined the data obtained through the survey (Section 3) we conducted with the subjects. MV subjects estimate that they spent 30.6% of their time on debugging tasks, whereas STM subjects spent 42.95%. The answers that the subjects provided differed significantly, with 95% confidence (p-value = 0,0277). This results suggests that our conjecture is valid.

5. Related Work

Perfumo et al. [10] present a benchmark aiming to assess the performance of Haskell STM (and extensions to it) and a set of metrics for the performance of applications that use STM. They performed a number of measurements using the applications that compose the benchmark. This work aims to support the evaluation of new ideas in STM, whereas ours evaluates the current state of the practice in Haskell STM.

Nakaike and colleagues [8] evaluate the performance of STM. They refactored three real Java applications so that critical parts of the applications executed transactionally. The authors found out that existing application-level performance optimizations for locks do not work well with transactions because they produce many conflicts. By disabling these optimizations, STM achieved higher or competitive performance. Zyulkyarov and colleagues [14] refactored a parallel version of the Quake game so as to use transac-

tions wherever possible. The resulting system exhibits transactions with a wide variety of characteristics. The paper presents examples of interesting situations such as error handling code within transactions and transactional code that is not block-structured.These studies complement ours in understanding trade-offs associated with STM. Both take existing applications that already use locks as a starting point.

Rossbach et al. [11] examined more than 1000 implementations of a concurrent shooting gallery simulation developed by three groups of undergraduate students. They cover various approaches for concurrent programming: coarse- and fine-grained locking, condition-based synchronization, and transactions. The study reports that students committed more mistakes using fine-grained locking than STM, although coarse-grained locking and STM had similar results and presents common error patterns. As pointed out elsewhere [9], this study has some methodological problems. For example, students of the first year used a different implementation of STM and 3rd year students could have leveraged knowledge of the 2nd year students. On the other hand, Pankratius and Tabatabai [9] organized twelve graduate students in pairs to develop a parallel search engine. Three pairs used STM and the other three employed locks. The authors found out that TM pairs where among the first to have a prototype parallel search engine, spent less time debugging segmentation faults, and produced code that is easier to understand. Even though it presents many valuable lessons, this study makes strong claims based on a small amount of data. Hence, we cannot generalize its findings. For example, one of the STM pairs could not finish the assigned task. On the other hand, another STM pair was the first to produce a parallel prototype. These results suggest that the employed technique may not have had a significant effect on the performance of the subjects.

Except for the first one, all the aforementioned studies target languages (Java and C++) that have STMs that are considerably different from Haskell's, in particular, with no support for condition-based synchronization. The latter two report data about time spent using estimations, whereas we use actual measurements.

6. Acknowledgments

We would like to thank the anonymous referees, who helped to improve this paper. Fernando is supported by CNPq/Brazil (308383/2008-7 and 475157/2010-9) and FACEPE/Brazil (APQ-0395-1.03/10). João is supported by FACEPE. This work is partially supported by INES (CNPq 573964/2008-4 and FACEPE APQ-1037-1.03/08).

7. Concluding Remarks

To the best of our knowledge, this is the first assessment of STM targeting a functional language. It is also the first evaluation of Haskell's concurrent programming constructs. Furthermore, we are not aware of any controlled experiment evaluating the ease of using STM when compared to a lock-based approach. Hypotheses H1-8 of the study (Section 3) could not be rejected. This might indicate that Haskell's STM is not easier to use than mutable variables when only coarse-grained locking and synchronization are required. STM programs did not exhibit less errors, less LoC, nor required less time to implement. The only exception to this was the time spent by subjects who did not commit concurrency errors. It was only possible to reject hypothesis H9. We stress that mutable variables are not locks. They are higher-level abstractions that combine mutual exclusion and synchronization. Thus, it is not realistic to extrapolate the results of this study to other lock-based approaches.

We intend to proceed in two directions. First, by conducting a controlled experiment with a focus on fine-grained locking. Second, by conducting an understanding evaluation, asking the subjects to point out problems in a program that should adhere to a given specification.

References

[1] D. Bacon et al. The "double-checked locking is broken" declaration. http://www.cs.umd.edu/˜pugh/java/memoryModel/DoubleCheckedLocking.html, 2000.

[2] G. Bracha. Maybe monads might not matter. http://gbracha.blogspot.com/2011/01/maybe-monads-might-not-matter.html, January 2011.

[3] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the 10th PPoPP*, pages 48–60, 2005.

[4] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. 20th ISCA*, pages 289–300, 1993.

[5] R. Hickey. The clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic languages*, 2008.

[6] S. Marlow. Parallel and concurrent programming in haskell. http://community.haskell.org/ simonmar/par-tutorial.pdf, June 2011.

[7] S. Marlow et al. [haskell-cafe] haskell memory model (was iterio-0.1). http://www.mail-archive.com/haskell-cafe@haskell.org/msg89866.html, May 2011.

[8] T. Nakaike et al. Real java applications in software transactional memory. In *Proceedings of IISWC'10*, pages 1–10, December 2010.

[9] V. Pankratius and A.-R. Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *Proceedings of the 23rd SPAA*, pages 43–52, June 2011.

[10] C. Perfumo et al. The limits of software transactional memory (stm): dissecting haskell stm applications on a many-core environment. In *Proc. 5th Conf.Computing Frontiers*, pages 67–78, May 2008.

[11] C. J. Rossbach et al. Is transactional programming actually easier? In *Proceedings of the 15th PPoPP*, pages 47–56, January 2010.

[12] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[13] M. Swaine. It's time to get good at functional programming. *Dr. Dobb's Journal*, December 2008.

[14] F. Zyulkyarov et al. Atomic quake: using transactional memory in an interactive multiplayer game server. In *Proceedings of the 14th PPoPP*, pages 25–34, February 2009.