Teaching Software Transactional Memory in Concurrency Courses with Clojure and Java

Antonio J. Tomeu[1(✉)], Alberto G. Salguero[1(✉)], and Manuel I. Capel[2]

[1] Dpto. de Ingeniería Informática, Universidad de Cadiz,
11519 Puerto Real, Spain
{antonio.tomeu,alberto.salguero}@uca.es
[2] College of Informatics and Telecommunications,
University of Granada, 18071 Granada, Spain
manuelcapel@ugr.es

Abstract. In the field of concurrency and parallelism, it is known that the use of lock-based synchronization mechanisms limits the programming efficiency of concurrent applications and reveals problems in thread synchronization. Software Transactional Memory (STM) is a consolidated concurrency control mechanism that may be considered as an alternative to lock-based constructs for programming critical software, although STM is still not fully accepted as a programming model for the industry. It is our opinion that STM programming must be more emphasized in undergraduate courses on concurrency and parallelism. In this paper we propose an academic experience regarding the introduction of STM programming in concurrency courses by using the Clojure language as the common vehicle for teaching Concurrent Programming. Java, the most popular and extended programming language for teaching concurrency, becomes a second language in our course, and thus our students can take advantage of Clojure API which is defined in Java in order to simplify the development of programming, lectures and assignments.

Keywords: Clojure · Concurrency · Java · Locks · Mutual exclusion
Threads · Transactions · Software Transactional Memory · Performance

1 Introduction

At moment, programming with locks at different abstraction levels is the dominant programming paradigm to teach how to program thread synchronization in concurrency courses. There is an ample range of concurrent constructs for programming concurrent applications; from the simple, standard locks or semaphores to the most sophisticated syntactical constructs such as monitors, they all offer good performance and a relative ease of use when it comes to program concurrent applications. However, all these syntactical mechanisms suffer from the lack of verifiability and reliability. Therefore, sometimes is difficult to obtain solutions applicable to concurrent programs that guarantee safety and

D. B. Heras and L. Bougé (Eds.): Euro-Par 2017 Workshops, LNCS 10659, pp. 266–277, 2018.
https://doi.org/10.1007/978-3-319-75178-8_22

liveness properties even when using formal techniques. The probability of a program code produced with non-verified synchronization mechanisms to crash or yield a deadlock situation is not negligible. However, if we analyze popular concurrent/parallel programming languages such as Java or C ++, we find that any specific API for managing concurrent tasks usually offer a wide variety of lock-based synchronization tools, being only a few of them based on STM. In particular, the last revision of Java [9] does not include native STM, whereas the C++14 [11] revision does as an "experimental feature". If we analyze the situation in concurrency courses, the situation is very similar. Information about STM programming model is mentioned superficially in concurrency courses. It is pointed out that, although STM programming is a mature model commonly used in research, it is still not used for commercial exploitation of parallel/-concurrent software development [3,16]. Moreover, recent curriculum guides [5], [17] that outline courses on concurrency do not pay special attention to that topic or include STM contents in course programs. This paper shows the results obtained from a study of teaching improvement in Concurrent and Real Time Programming course, which was carried out at the University of Cadiz (Spain) during one semester. The main objectives of the study have been:

(a) To introduce the STM model to students, as a viable alternative to the blocking thread model (thread synchronization based on locks) along with the model advantages and disadvantages.
(b) To provide the students with the necessary skills to allow the development of concurrent programs that include transactions for shared data access by Clojure's concurrent threads.
(c) To use Clojure as a programming language on top of Java for transactional programming in a multi-core environment, and thus to allow the students to develop programming solutions by programming Clojure's transactions.
(d) To show the students that both paradigms are not mutually exclusive but complementary.

The paper is organized as follows: Sect. 2 briefly describes the academic context of the study. Section 3 introduces the STM programming "paradigm" and how is presented to the students. Section 4 shows the way STM that uses Clojure is taught and Sect. 5 does the same with Java on Clojure. Section 6 gives further details on how the experiment was developed by the students and evaluated to check the performance of STM-based w.r.t. the solutions based on the classic blocking thread model. Section 7 outlines the conclusions reached and the future work to be developed.

2 Academic Context

The reported study was developed in the third semester of the CSE curriculum at the University of Cádiz (UCA), Spain, in an undergraduate course. A total amount of $n = 199$ students were enrolled in the course "Concurrent and Real Time Programming", which was divided into two groups for theoretical lectures

and eight groups for practical work in the labs in this study. The semester lasted fifteen weeks (60 h per student with 4 h of teaching per week: 2 h of lectures and 2 of practical work. The course structure and contents, according to the current recommendation guides [5,17], were the next ones:

1. Fundamental concepts of concurrent programming: race conditions, mutual exclusion, synchronization, and properties of concurrent systems (15%).
2. Mutual exclusion: algorithms for shared memory multiprocessors, semaphores and **software transactional memory**[1]. (20%).
3. Monitors: Hoare's monitor model, signaling semantics, verification of concurrency properties (security and liveness) (20%).
4. Message passing and distributed programs: RPC and RMI models, MPI, rendez-vous (15%).
5. Real-time systems: periodic tasks scheduling based on static priority assignment, scheduling tests, priority inversion anomaly and sporadic task scheduling (30%).

The distribution of course topics within the 60 h of teaching (lectures + lab) was the following one: fundamentals (4.5+6), mutual exclusion (6+8), monitors (6+8), message passing (4.5+6) and real-time systems (9+2). 30 h of lab work were spent to teach theoretical contents with the help of Java code-snippets, which were taught according to a weekly schedule proposed by teachers. A total of 3 h were spent to carry out the experiment regarding learning mutual exclusion conditions and solutions, which were distributed following the next format: one hour for a theoretical seminar on STM fundamental concepts and two hours for practical work at the laboratory, where the students can experiment with the STM Clojure and Java code-templates provided by teachers. The course development has been supported by a Moodle virtual platform, which provided students: previous readings to each lecture, the slides shown in classroom, and all the code samples used in the exercises proposed to the students during the semester.

3 Software Transactional Memory

It is well known that common synchronization techniques in concurrent programming suffer from several drawbacks, i.e., if these techniques are not used properly, or we forget to do a good lock release check control, the changes performed by one thread in the program may not be visible to the other threads. In spite of all that have been written about how to avoid these problems [12], [7], and the numerous formal techniques that have been proposed recently, concurrency control remains a complex issue in general. Not all people are able to produce valid code (free from race conditions, threads starvation and livelock).

[1] The transactional memory was introduced by us in this section during the academic year 2016–2017 to carry out the study.

The STM paradigm can change now the previous situation, i.e., it becomes feasible to program safe and fair concurrent code by everybody, by introducing the concept of transaction, which can be defined as a region of code that is executed atomically, consistently and in isolation with respect to other program regions. When two threads try to access the same data, the transaction manager is activated to resolve the conflict, without resorting to explicitly use blocks in the code. When a transaction is in progress, the transaction is completed and the changes are written into memory if there is no conflict with other threads/transactions. However, as soon as the transaction handler finds that one transaction has progressed beyond a certain point that makes the current transaction unsafe by compromising the data consistency, it undoes the changes and tries again.

When STM is used, the concurrent readings are done without any problems, and without the presence of contention. With the STM model, conflicts only occur when a thread is writing to shared data; in that case, the transaction manager records the program state, so that all previous work done by the thread can be *rolled-back* and then the thread retries until the transaction can be successfully completed; this occurs when the threads that are modifying data finish to do so and validate those changes in memory (*commit*). The STM model is very suitable when considering critical sections with many readings and occasional writings, where we can expect little containment [13]. By contrast, the blocking model degrades the performance in this case, since it implements a pessimistic control of the concurrency, eliminating the parallelism within the critical sections. At this point, we consider the need to choose an implementation of STM to work with our students. Compatibility with the Java language was fundamental, as the students had developed all their practical assignments in Java in other courses of the curriculum. There are multiple STM implementations for Java [1,7,10,14,15,20]. We did not choose any of them, because they are too complex for the objectives we set for the teaching of STM. Instead, we chose to use Clojure functional language, which is interpreted by the JVM, and yields compatibility between both languages/APIs, which was very useful for us.

4 Teaching STM with Clojure

In Clojure, the STM separates the identity of an object from its state [18]. Clojure is a functional language where the states never change, as they are immutable by definition. The changes are produced in identity of the object, which is actually the visible information for the threads. Values are only immutable within the scope of a Clojure transaction. By design, the identities are the mutable part, and therefore it is not possible to inconsistently change the states. Any attempt to change the identity of an object outside of a transaction is considered illegal in Clojure, and thus an exception is thrown if that situation arises.

Since there are no locks, concurrency is improved in comparison to the thread blocking model [2,6]. Correct understanding of this separation between identity and state is crucial for the students to internalize the operation of the STM in

Clojure. It is also explained to the students that the STM model works as long as its implementation can guarantee that threads always get a consistent view of the world during the program execution. This is true with Clojure, so we do not have to worry about checking it, which is an advantage for newcomers to the STM world. The transaction manager, which supports STM, is responsible for doing it for us. Teaching the Clojure transactional control to our students was not an issue, since we used a set of Clojure code patterns, as the one shown below. In that code our students can visualize how to perform the identity change that we want to achieve by wrapping it in a transaction (`(dosync...)`). Clojure implementation of STM guarantees that any transaction execution is atomic, isolated and consistent. We also specially insisted on the similarities and differences that the pattern presents with respect to the classic blocking thread synchronization pattern with locks.

```
1 ;;how to use transactions in Clojure
2 ;;now, the shared data...
3 (def n (ref 0))
4 (println "n is: " @value)
5 ;;doing the transaction...
6 (dosync
7   (ref-set n 1))
8 (println "n is: " @value)
```

A thread's transaction is only completed if there is no conflict with another running threads/transactions at the moment, and the changes are written to memory (*commit*). If some conflict is detected by the transaction handler, as result of multiple threads concurrently accessing[2] to the shared data, the transaction handler pauses the contending threads, undoes the transaction (*roll-back*) and starts them again. Therefore, blocking situations among threads cannot arise with Clojure transactions, though there is obviously a price to pay for that, i.e., transactions require an extra processing time [4] compared with thread synchronization based on locks. As one part of the correct understanding and basic use of Clojure transactions, there were foreseen practical work assignments at the laboratory that included the following actions: the elaboration of a multi-threaded application for the concurrent access to the variable `n` as in the previous code, and the elaboration and analysis of a number of critical sections following the previous model.

5 Teaching STM with Java over Clojure

Once the theoretical and practical concepts to develop secure transactions with Clojure have been presented to students, we have extended our experience to the field of Java language, which was used during all the practical lessons conducted at the laboratory during the semester. To develop the analysis of the STM behavior in the Java language, we began to familiarize the students with the transactional pattern that had to be used, which is shown below,

[2] Students were asked, within the corresponding assignment, to do just that.

```
myThread h = (myThread)Thread.currentThread();
while (true) {
  t.beginTransaction();
  ... // do critical section
 if (t.commitTransaction()) {
 break;
}
}
```

The code illustrates how the general transaction pattern in Java surrounds and protects the access to the shared data in a transaction, within which the threads remains until the transaction ends up and the writing of data in memory is successfully validated. The pattern shows to the students the transaction execution continuity, by following a continuous iterative form while the transaction needs to perform, and without the presence of locks. When this pattern was correctly understood by our students, we went on to develop two Java STM experiments in Clojure with the students, (1) concurrent multi-thread access to a shared variable using a standard race condition, and (2) concurrent access to a bank account abstraction[3]. Below we show the control of a race condition with transactions. The control of the bank account is very similar, and is not shown for reasons of space. The code provided to our students for solving a standard race condition was as it follows,

```
import clojure.lang.Ref;
import clojure.lang.LockingTransaction;
import java.util.concurrent.Callable;

public class Counter {
  final private Ref count;

  public Counter(final int valInic) throws Exception {
    count = new Ref(valInic);
  }

  public int getCount() { return (Integer) count.deref(); }

  public void inc() throws Exception {
   LockingTransaction.runInTransaction(new Callable<Integer>() {
     public Integer call()  {
         int countNow = (Integer) count.deref();
         count.set(countNow+1);
         return (Integer) count.deref();

      }
    });
  }

```

[3] All code shown in the rest of the document is available at the following URL: https://antoniotomeu.wixsite.com/atomeu/stmjavaonclojure.

```
25  public void dec() throws Exception {
26    LockingTransaction.runInTransaction(new Callable<Integer>() {
27      public Integer call()  {
28
29          int countNow = (Integer) count.deref();
30        count.set(countNow-1);
31        return (Integer) count.deref();
32
33      }
34    });
35  }
36
37 }
```

The support for STM programming that Clojure offers to its users is imported in lines 1 and 2 into the Java code. Line 6 declares the shared resource with the implicit separation of the identity and the state that Clojure offers. The `Counter` class shows an API with three methods. The first one is an observer that allows the client to obtain the value of `count`. The other two methods are modifiers that increase or decrease the value of `count`. Please, notice that `count` is a counter with the initial value 0 set by the class constructor. The `inc ()` method increments the value of `count`, which is value referenced, and thus it is firstly necessary to dereference it, i.e., we have to follow the reference to obtain its value (line 17). Line 18 increments the counter value by means of an auxiliary variable, and sets the reference to that new value by using `count.set(CountNow + 1)`. Since the program uses Clojure to support transactions the code that is executed inside the transaction must implement the interface `Callable`, which models the asynchronous execution in Java. This is not a problem, since the students acquired familiarity with this interface from previous practical assignments. The entire code of the method is programmed within only one transaction defined in line number 15 and supported by Clojure. The referred transaction includes the entire code of the method with the appropriate syntax delimiter, which is written as:

```
1   LockingTransaction.runInTransaction{
2      //critical section
3   }
```

It was crucial for our students to understand that this delimiter encompasses the persistent looping behavior shown above and that the transaction is continuously running until it is capable of validating data writing into memory. If several threads make a call to the `inc ()` or `dec ()` methods, the Clojure transaction handler makes sure that the modification process is performed properly so that the final value of `count` is consistent. Within the practical assignment that the students had to develop, an exercise was included to develop a Java program that activates multiple threads against an object of the class `Counter`. Half of the threads must invoke the `inc()` method in a `for` loop and the remainder must invoke the `dec()` method. To finish this experiment, students must check that

the resulting final value was 0. An example of that program, which we developed, is contained in `userCounter.java` and can be downloaded and tested from the given `url`.

6 Performance Analysis

We also wanted to offer to our students a benchmark for the comparison of the performance between the transactional and the standard thread blocking model. To do this, we developed an experiment, during practical work on transactional memory in the lab, consisting on defining a fine-grain standard critical section code region (`n++`), and to write code for threads that concurrently accessed to it. We have used different control techniques with locks [8] to achieve secure access to the critical section region. More specifically, the access to the region was controlled using `synchronized` methods; versions using the standard API for concurrency, i.e., the `AtomicInteger`, `ReentrantLock` and `Semaphore` classes, included in the high-level API for concurrency control, were also developed; all of those primitives were already known by the students. Finally, we have written a version that wraps the critical region within a transaction written in Java by means of the Clojure API. In addition, we have written an alternative version in Clojure without the Java API, which supports access to the critical data section through its native STM.

Using this code, and the Java previously described models, we proposed two additional experiments to our students for conducting performance measurement:

(a) Basic load experiment: in this experiment the students had to measure the time required to execute a protected critical region that was defined either by using the standard synchronization control techniques of the Java language, or by using STM in Java by means the Clojure language.
(b) High load experiment: in this experiment the students must perform a temporal analysis by using multiple threads which contend to access to a shared resource during a high number of iterations (2×10^6).

Below we describe with more detail the experiments that our students developed under our direction.

6.1 Basic Load Experiment

An elementary critical section with a single write operation was used, and the time required to execute that operation under all typologies of the blocking model and under the STM in Java through Clojure were measured[4]. The students were

[4] Time were measured using the `nanoTime()` method of Java `System` class. This implies that it is a time that only and exclusively makes sense in the realm of the virtual machine, and has no relation whatsoever was the time provided by the system clock. However, Clojure, like Java itself, executes *bytecodes* on the JVM, which gives consistency to the results.

required to complete a tabular questionnaire with these times as part of their practical assignment in order to make them aware of the actual time cost of each control technique.

The table, once completed, should show to students how executing a single transaction to provide a safe access to a shared resource nearly doubles the execution time needed by a slower lock-based access in regular Java code. This can happen even in a scenario without multiple threads in execution. Of course, students checked through this exercise that the use of transactions of Clojure was a good election in a situation that requires few accesses to shared resources. However, when the number of accesses to shared resources is high, it is necessary to evaluate the performance of using STM with Clojure in Java.

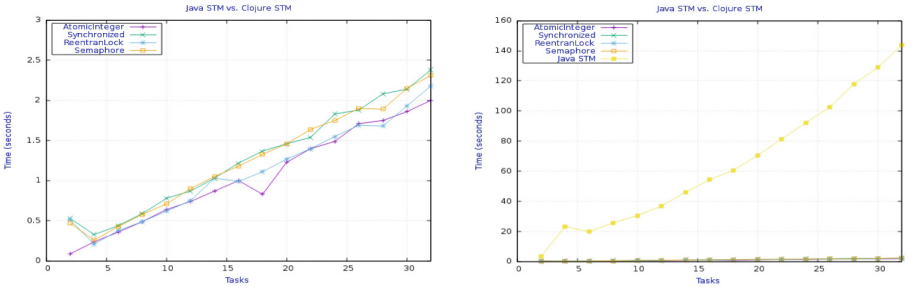

Fig. 1. Java synchronization vs. Java-STM

6.2 High Load Experiment

In this scenario, students were required to run each test program with an increasing number of threads, from 2 to 32. Half of the threads had to increase the counter, and the rest had to decrease it. In all cases the threads had to be launched using a fixed-size executor. A condition had to be entered in all programs for waiting the executor to run all threads, followed by a control printout of the value of the shared variable, which should always be 0 in our case. Each thread made a total of 2×10^6 iterations. The students were then required to develop the measurements for the scenario described, and to draw the curves $Time = F(threads)$. To do this, we made available the required *GnuPlot scripts* to the students through the virtual platform of the course. We also provided our own curves as a working guide, indicating the parameters that supported our own experiment: Intel (R) Core i5-4440 CPU @ 3.10 GHz processor, with 4 physical cores without *hyper-threading*, using Fedora 22 as the Linux platform. The version used for the JDK was 1.8.0_54, and version 1.8.0 was used for Clojure. The results of our test, were given as a guide to the students, are shown in Figs. 1 and 2.

The Fig. 1 (left) illustrates the behavior of standard synchronization techniques in Java, and has no further interest. The Fig. 1 (right) shows the comparative performance of standard Java synchronization techniques compared with

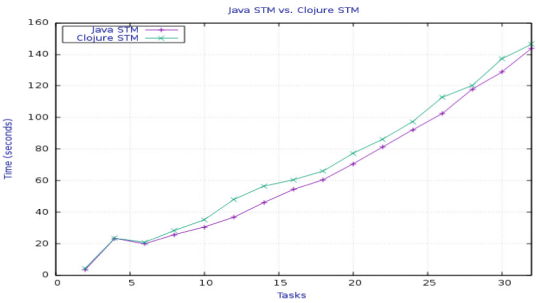

Fig. 2. Java-STM vs. Clojure-STM

STM with Clojure in Java. We can appreciate that the performance of this particular implementation of the STM is bad for tasks that try to frequently access the shared resource, since the necessary roll-backs are very expensive overhead [10]. Finally, the Fig. 2 compares the usage of STM in both languages (Clojure and plain Java). Even in this case, in which we compare a native Clojure implementation of STM with the Java implementation of the STM, we see how Java always behaves better in the range of tasks analyzed, which cannot be considered as a surprise, because Clojure is a pure interpreted functional language.

It is necessary to clarify, however, that the behavior we have shown here corresponds to the analysis of really extreme scenarios, where the typology of the developed threads is very specific, and always use the critical section to perform data writing. It is important to persuade the students to analyze and decide on these aspects by their own [19].

7 Experience Results and Conclusions

To measure the results of the experiment, we asked our students to respond a survey ($n = 124$), where the answers range from 1 (completely disagree) to 5 (fully agree). The value 0 was used when the student did not respond to an item. The items selected were:

(a) I have understood the concept of transaction as an alternative to the use of blocking techniques based on locks.
(b) I have learned how to use transactions with Clojure to protect concurrent access to shared data.
(c) I have learned how to use transactions with Java to protect concurrent access to shared data.
(d) I have understood the advantages and disadvantages of using STM.

The results of the survey are shown in Fig. 3, which shows that the results of the experiment were satisfactory, and that students finally reach an adequate level of understanding of the concept of transactional memory presented, both theoretical and practical.

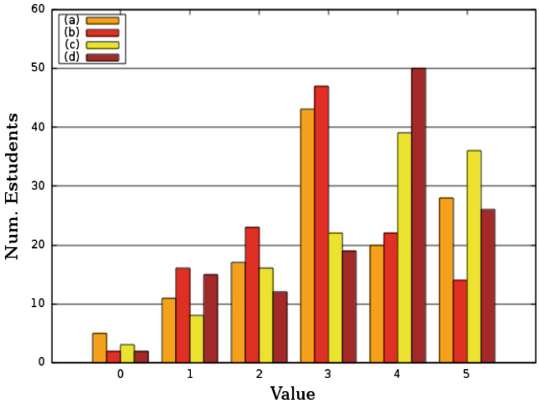

Fig. 3. Valuation survey

From the experiment results evaluation, we decided to keep on teaching the STM as part of the concurrent programming education programme in future editions of the course, and perhaps to slightly extend the time planned for this topic within the course schedule. We also believe that it could be of great interest for other courses on concurrency the development of a similar experience with other programming languages such as Akka, Scala or perhaps C++ if it finally includes transactional memory in the corresponding API.

References

1. Brevnov, E., Dolgov, Y., Kuznetsov, B., Yhershov, D., Shakin, V., Chen, D., Menon, V., Srinivas, S.: Practical experiences with Java software transactional memory. In: The 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2008)
2. Carlstrom, B., Chung, J., Chafi, H., McDonald, A., Minh, C., Hammond, L., Kozyrakis, K., Olukotun, K.: Executing Java programs with transactional memory. Sci. Comput. Program. **63**, 111–129 (2006)
3. Cascaval, C., Blundell, C., Michael, M., Cain, H., Wu, P., Chiras, S., Chaterjee, S.: Software transactional memory: why is it only a research toy. Commun. ACM **51**(11) (2008). https://doi.org/10.1145/1400214.1400228
4. Clarke, F., Ekeland, I., Pedrero, M., Gutierrez, E., Romero, S., Plata, O.: Improving transactional memory performance for irregular applications. Procedia Comput. Sci. **51**, 2714–2718 (2015)
5. The Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM) and IEEE Computer Society. Computer Science Curricula 2013 Curriculum Guidelines for Undergraduate Degree Programs in Computer Science. https://www.acm.org/education/CS2013-final-report.pdf. Accessed 27 Mar 2017
6. Dias, R., Vale, T., Lourenço, J.: Efficient support for in-place metadata in Java software transactional memory. Concurr. Comput. Pract. Exp. **25**, 2394–2411 (2013). https://doi.org/10.1002/cpe.3098. Wiley Online Library

7. Diegues, N., Fernandes, S., Cachopo, J.: Parallel nesting in a lock-free multi-version software transactional memory. Technical report RT/2/2012 (2012). http://algos.inesc-id.pt/~jpa/InscI/poisson/varwwwhtml/portal/ficheiros/publicacoes/7621.pdf. Accessed 20 Mar 2017
8. Fernández, J.: Java 7 Concurrency CookBook. Packt Publishing, Birmingham (2012)
9. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A.: The Java Language Specification. Java SE, 8 edn (2015). https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf. Accessed 15 Feb 2017
10. Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.394.9533&rep=rep1&type=pdf. Accessed 20 Mar 2017
11. ISO: Working Draft, Standard for Programming Language C++ (2016). http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/n4618.pdf. Accessed 16 Feb 2017
12. Malde, K.: Can software transactional memory make concurrent programs simple and safe? http://cs.brown.edu/~mph/HerlihyLM06/dstm2.pdf. Accessed 20 Mar 2017
13. Mizuno, K., Nakaike, T., Nakatani, T.: Reducing rollbacks of transactional memory using ordered shared locks. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 704–715. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03869-3_66
14. Mohamedin, M., Ravindran, B., Palmieri, R.: ByteSTM: Virtual Machine-level Java Software Transactional Memory. http://www.ssrg.ece.vt.edu/papers/coordination_15_CR.pdf. Accessed 20 Mar 2017
15. Nakaike, T., Odaira, R., Nakatani, T., Michael, M.: Real Java applications in software transactional memory. In: IEEE International Symposium on Workload Characterization (2010). https://doi.org/10.1109/IISWC.2010.5654431
16. Pankratius, V., Adl-Tatabai, A.: Software engineering with transactional memory versus locks in practice. Theory Comput. Syst. **55**(3), 555–590 (2013). https://doi.org/10.1007/s00224-013-9452-5
17. Prasad, S.K., et al.: NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing - core topics for undergraduates. https://grid.cs.gsu.edu/~tcpp/curriculum/?q=system/files/NSF-TCPP-curriculum-version1.pdf. Accessed 20 Mar 2017
18. Subramanian, V.: Programming Concurrency on the JVM: Masterig Synchronization, STMA, and Actors. The Pragmatic Bookshelf, Dallas (2011)
19. Yamada, Y., Iwasaki, H., Ugawa, T.: SAW: Java synchronization selection from lock or software transactional memory. In: Proceeding of IEEE 17th International Conference on Parallel and Distributed Systems, pp. 104–111 (2011)
20. Ziarek, L., Welc, A., Adl-Tabatabi, A., Menon, V., Shpeisman, T., Jagannathan, S.: A uniform transactional execution environment for Java. https://www.cs.purdue.edu/homes/suresh/papers/ecoop08.pdf. Accessed 20 Mar 2017