Avoiding Publication and Privatization Problems on Software Transactional Memory

Holger Machens and Volker Turau

Hamburg University of Technology, Institute of Telematics
Hamburg, Germany
`{machens,turau}@tuhh.de`

Abstract

This paper presents a new approach to exclude problems arising from dynamically switching between protected concurrent and unprotected single-threaded use of shared data when using software transactional memory in OO languages such as Java. The approach is based on a simple but effective programming model separating transactions from non-transactional operation. It prevents the application programmer from errors but does not force the software transactional memory library to observe non-transactional access and thereby preserves modularity of the software. A prototypical toolchain for validation and source code instrumentation was implemented as a proof of concept.

Keywords and phrases Software Transactional Memory, Publication, Privatization

Digital Object Identifier 10.4230/OASIcs.KiVS.2011.97

1 Introduction

Due to physical and economical reasons CPU manufacturers have stopped to increase CPU clock speed and now increase the number of cores on a single die instead. As software is constantly getting more complex and consumes more processing power with each new version today's software industry is coerced to move to parallel programming paradigms. Unfortunately parallel programming is a complex and error-prone discipline due to the necessity of data exchange between threads of execution whether by message passing or concurrent access to shared data. Access to shared data was traditionally controlled by locks that may cause deadlocks and do not scale well (cf. [6]). Another approach is the use of transactions as known from databases. Emerging from that discipline transactional memory [6] is currently discussed as a viable alternative for locks to ease parallel programming.

Transactional memory (TM) is a transaction-based concurrency control mechanism for shared non-persistent memory on a single machine. Some prototypes have been developed as hardware transactional memory (HTM) integrated into a CPU, e.g. Sun's Rock processor. Software transactional memory (STM, [11]), as addressed in this contribution, is implemented in software libraries usually providing transactions on heap memory of a process. HTM has not yet reached a productive state, therefore most research focuses on STM.

The underlying principle of software transactions is the repeated execution of a critical section to resolve occurring conflicts in terms of data consistency or overall progress (i.e. deadlocks). Data modifications are committed if no conflict occurred during a transaction and aborted (i.e. discarded) otherwise, resulting in a restart of the transaction. STM can be based on pessimistic or optimistic concurrency control techniques. While pessimistic STMs acquire locks before accessing shared data optimistic STMs do not. Thus, before the latter commits a transaction it checks if the order of concurrently occurred accesses on shared data is result-equivalent with some serial execution of the involved transactions (*serializability*).

17th GI/ITG Conference on Communication in Distributed Systems (KiVS'11).
Editors: Norbert Luttenberger, Hagen Peters; pp. 97–108
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Generally, TM provides the opportunity to easily move to concurrent programming without most of the common problems. It has the potential to push the use of concurrency and thereby prevent the software industry from losses caused by the change of the architecture paradigm on the hardware level. But transactions have fundamental differences compared to traditional locks. Some common patterns to solve tasks in concurrent applications do not apply to transactions and result in errors when used. Thus, to ease the transition to concurrent programming through TM its usability is of higher concern; in particular the prevention from its own known problems.

One class of problems of TM is called the privatization resp. publication problem [12]. Both problems emerge from a strategy used by application programmers to switch between protected shared use and unprotected local use of data and result in inconsistencies when applied with transactions. The currently accepted solution for these problems is to enforce strong isolation, which means that transactions are isolated from modifications of other transactions as well as they are isolated from non-transactional modifications on shared data. This usually requires the STM library to observe non-transactional access to shared data, which either requires modifications of the runtime environment (e.g. the virtual machine) or modifications of the software which uses a library implemented on an STM.

But a library must be self-contained. Considering larger ecosystems, such as the Eclipse platform, where a data model and plug-ins working on that data model are provided by different parties such modifications in plug-ins reduce the attractiveness of the platform. But even those large ecosystems would greatly benefit from deadlock prevention provided by STMs. Firstly it is hard to agree on some locking protocol for objects of the model which prevents deadlocks and secondly the usage of a single lock for the entire model is very inefficient. Therefore, there is a strong need for an alternative solution.

We advocate an imperative style declaring a programming model which prevents common errors when using TM such as privatization and publication. The programming model is enforced by a validation tool and automated code instrumentation is used to achieve a simple and almost transparent API. The programming model is designed for Java STMs and it neither depends on language extensions nor forces the STM library to demand new requirements for the runtime environment or the non-transactional part of the software.

Instead of realizing strong isolation inside the TM system it is enforced by the coding rules of the programming model. All rules of the programming model are derived from the following goal: Transactions must operate in a predefined domain of the software only. Access to the domain is possible via methods of objects only, which automatically start transactions. This completely isolates transactions from non-transactional access logically and physically in terms of software architecture. Thus, it realises strong isolation without the disadvantages in respect to the remaining software system and its environment.

A prototypical implementation of a TM system based on a modified STM library was developed as a proof of concept. Following the main requirement to reduce complexity of concurrent programming it provides a very simple API and a toolchain consisting of a validator checking for violations of the programming model and an automated code instrumentation tool to enable transaction management for declared transactional objects.

Section 2 reviews details on problems resulting from the use of transactions, existing approaches and their pros and cons. Also relevant programming models for STM in Java are discussed. The next section explains the overall concept and the programming model. On this basis the developed prototype and required modifications of the STM library are covered. Section 5 contains a detailed discussion of the consequences on programming and impact in terms of non-functional properties of the STM library induced by the modifications such as performance and scalability. The paper finishes with a conclusion and a prospective outlook.

2 Pitfalls of Existing Approaches

2.1 Publication and Privatization

As mentioned above, publication and privatization describe strategies of the programmer to implement switching between local unprotected and shared protected access to data objects.

Initially: `data = 42; _private = false;`	
Thread T1	Thread T2
1 atomic {	1
2	2 atomic {
3	3 if (!_private) {
4 _private = **true**;	4
5 }	5
6 data = 0;	6
7	7 val = ++data;
8	8 }
9	9 }
May this code assign `1` to `val`?	

Table 1 Example of Privatization

Table 1 shows an example implementation of privatization. Thread T1 privatizes variable `data` and signals it to T2 by the flag `_private`. The code works fine with locks but represents a typical problem for transactions when executed in parallel line by line in the given order. Thread T1 expects the data to be private and accesses it non-transactionally but unfortunately thread T2 already tested the data to be public and reads it after the update of T1 in line six. When using conventional locks variable `val` will in T2 never get the value 1, this was the programmer's intention. When using TM instead one of the following may happen:

- The TM system treats the contention caused by the update in line four to be valid because there exists an equivalent serialization of both executions with the same result. This valid serialization is: Execute line seven before line four. It seems valid to the TM system because the access to data in line six is invisible to the transaction in T2 which means line four actually causes no contention.
- The TM system might even consider the contention in line four to be valid and will decide in line nine to rollback all changes. This includes to reset data to the state read at transaction begin, which was 42 and not 0.

A rare but even more obvious privatization problem occurs when trying to synchronize at an empty atomic block as demonstrated in Table 2. This is completely incompatible with transactions because no STM system will ever block a thread which is not accessing any data. But a synchronized block does!

These are just two examples of privatization problems and there are equivalent patterns in usage with publication [4]. Common to these problems is the unprotected use of shared data which is usually invisible to the TM system. The only way to overcome this problem is to protect any access to shared data which is potentially accessed by transactions concurrently. Such a TM system provides strong isolation, i.e. isolation of transactions from modifications of other transactions and from non-transactional modification as well. This is in contrast to weak isolation which isolates transactions from modifications of other transactions only.

Initially: `data = 42 ; _private = false;`	
Thread T1	Thread T2
	`atomic {`
	`if(!_private) {`
`_private = true;`	
`atomic {} ;`	
`data = 0;`	
	`val = ++data;`
	`}`
	`}`
May this code assign `1` to `val`?	

Table 2 Privatization using an empty synchronized block

To enforce strong isolation a TM implementation needs a way to observe non-transactional access to shared data. Regarding Java, this may be achieved by modifications to the runtime environment (e.g. the VM), the use of an agent at the JVMTI or JVMPI interface, or by source or byte code instrumentation. The latter methods work with insertion of code into non-transactional code which accesses the shared data to get notified when certain code lines are executed. In summary, the implementation of strong isolation requires the TM system to either modify the target runtime environment or parts of the program code. Thus, libraries implemented based on the TM system with strong isolation are no longer self-contained.

2.2 Related Problems

A known problem of transactions is caused by its all-or-nothing guarantee which requires rollbacks in case of inconsistent states of data. Rollbacks result in repeated executions of the same code sequence which is incompatible with operations performing data exchange with external entities not supporting transactions (e.g. a text console). A common solution to this problem is the use of so-called irrevocable transactions [9]: A transaction is made irrevocable when hitting a non-transactional method such as an I/O operation. All concurrently running transactions which might get in conflict with the irrevocable transaction are rolled back which guarantees the irrevocable transaction to commit without rollback.

A similar problem is the use of condition-driven synchronization (monitor, condition variables or similar wait/signal primitives) e.g. by calling `wait()` and `notify()` in Java (see Table 3). Usually this scenario is bound to a lock, protecting some variables and an expression based on a subset of those variables representing the condition expected by a waiting thread. Another thread, which changes variables referenced by the expression signals the changes and the waiting thread is resumed. The condition is checked again and either the loop gets restarted or the condition is met.

Waiting Thread	Signaling Thread
`synchronized (<lock>) {`	`synchronized (<lock>) {`
`while (<condition>) {`	`<condition> = true;`
`wait();`	`notify();`
`}`	`}`
`}`	

Table 3 Blocking synchronization

The inherent problem of this scenario is that the waiting thread must release the lock to allow the signaling threads to change the variables referenced in the expression. There are several proposals to provide an equivalent mechanism for transactions (cf. [5], [2], [1]) but they all are complex, reintroduce the danger of lifelocks or deadlocks, or are of restricted use.

2.3 Programming Models of STMs

The most prominent way to declare code sections to be transactional is the use of `atomic` blocks (see Listing 1) as in the TM systems ATOMOS [2] and AtomJava [7].

Listing 1 Use of `atomic` blocks

```
atomic {
  // transactional code goes here ...
}
```

Appropriate code to begin, control and end a transaction inside a block are transparently added by a compiler or by dynamic code instrumentation, thus rollback or commit is in most cases invisible to the application programmer. This approach requires an extension of the language causing problems with existing tools such as IDEs and compilers. Alternatively, transaction handling may be explicitly programmed by the application programmer using calls into the TM system. A similar approach is to provide transactional code sections to the STM system through an object with a designated method (e.g. `call()`) as in DSTM2 [8].

Another prominent object-oriented programming model for STM which provides transparent handling of transactions is to declare so-called transactional objects which support transactions. Transactional objects define transactional code sections in their methods and transactional data as member variables of the object. Programmers are still allowed to use even non-transactional access to transactional objects. There are examples such as Deuce [10] and Multiverse [13] which are based on this model and use annotations to declare objects and methods to be transactional (see Listing **??**).

Listing 2 Example for transactional objects in Multiverse label

```
@TransactionalObject
class MyTxObject {
  private int x;
  @TransactionalMethod
  public int incX() {
    return ++x;
  }
}
```

Atomicity of transactional objects is just an optional feature as long as the underlying STM library does not enforce strong isolation. Member variables may be globally visible and accessed by non-transactional operations, and transactional methods are able to access non-transactional data.

The first approach proposing so-called type-level enforcement of strong isolation was published by Harris et al. when introducing composable memory transactions [5] as an extension for Concurrent Haskell. Concurrent Haskell requires operations performing I/O actions to be explicitly declared as IO actions. With the proposed extension by Harris et al. the programmer can also define so-called STM operations. These require a running transaction and are restricted to use non-I/O operations only. Thus, strong isolation is enforced through the type system.

3 Proposed Programming Model

When looking at the problems mentioned in Section 2 it seems to be non-trivial to use TM and it was no surprise when the question appeared whether TM was just a research toy [3]. In databases those problems simply do not occur, because transactional data and transactions are completely separated from non-transactional operations. This implicitly provides strong isolation in terms of TM and it even prohibits calls to non-transactional operations and use of blocking statements. Correspondingly, the fundamental and generic concept of our programming model is to move TM and associated transactional code into a closed part of the system which is separated from non-transactional code.

In order to reduce the complexity in concurrent programming our programming model addresses the following requirements:

- Avoidance of privatization and publication problems: To eliminate these problems the programming model should support strong isolation.
- Preserve software modularity: To reduce impact on the runtime environment and to keep software modular, strong isolation should be enforced through the programming model rather than through the STM library as explained in Section 2.1.
- Simple API: The application programmer should be free from explicitly controlling concurrency by calling methods of the STM library.
- No changes to the programming language's syntax: The syntax should remain unmodified to preserve compatibility with existing tools. Therefore, language extensions such as usage of new keywords (e.g. `atomic`) are excluded.

3.1 Overall Concept

As mentioned above, the generic fundament of the programming model presented here requires to move transactions and transactional data into a logically separated *transactional domain* of the software. The transactional domain is meant to guarantee consistency by restricting access to contained data and operations to transactions only.

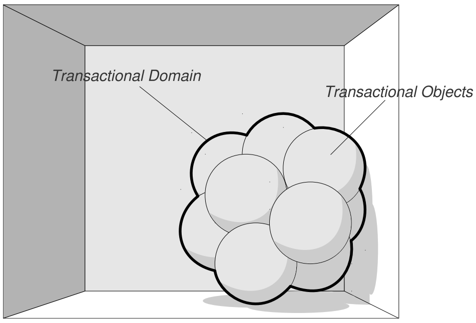

Figure 1 Transactional domain

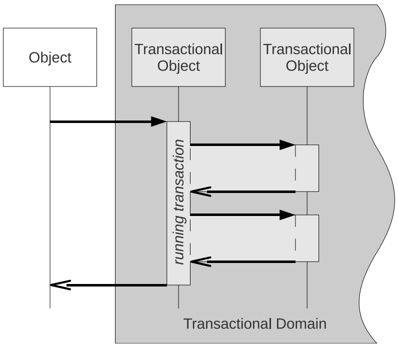

Figure 2 Implicit transactions

To achieve such a guarantee for Java the transactional domain is build up from transactional objects as depicted in Figure 1. Each transactional object is responsible to ensure guaranteed data consistency for its member variables. Each transactional object therefore offers no member variables but transactional methods only and each such transactional method transparently starts a transaction if not already running (see Figure 2). This way transactional data inside of transactional objects is accessed via transactional methods only which guarantees the consistency inside of them.

The interface of the transactional domain consists of the public transactional methods offered by the transactional objects. Crossing the boundaries of the transactional domain without starting a transaction is made impossible.

3.2 Coding Rules

Transactional objects are instances of transactional classes which are declared by implementing the empty interface `Transactional`. A class declared to be transactional will get all of its methods and constructors turned into transactional methods. Compared to annotations the usage of interfaces provides type-safety even for a development environment without the toolchain of the STM (e.g. when using a library with transactional objects). APIs of the STM can refer to objects implementing the interface `Transactional` while annotations need runtime checks or a tool for extended type checks at compile time.

A transactional class has to fulfill the following rules to guarantee strong isolation:

- Instance variables of transactional objects must be of primitive type, of type `String`, or references on transactional objects. This rule even prevents a programmer to put consistency on risk through methods returning references on internal non-transactional data or to keep temporary non-transactional data between two method calls. In Java primitives are returned by value and strings are constants. This makes modifications of both kinds of internal data impossible.
- Non-**`final`** instance variables of transactional classes must be **`private`**. By doing so, only methods - which start a transaction if necessary - can access internal variables.
- Inside of transactional methods any access such as method calls or variable access to external non-transactional objects is prohibited. That way it is impossible to break up strong isolation and get access to inconsistent data inside a transactional method. Usage of temporary local non-transactional objects in transactional methods (e.g. on the process stack) are allowed.
- Parameters of a transactional method must be of primitive type, of type `String` or transactional. Parameters provided to a method might be modified in the transaction and therefore need to support transactions as well. For all method parameters of primitive type, Java provides a call-by-value semantics. Thus, the method is working on its own copy and needs no concurrency control. The same applies to objects of class `String`.
- Arrays as parameters of transactional methods are prohibited because they do not support transactions. Arrays have to be replaced by a generic transactional wrapper class provided by the STM and supporting access to an underlying array.
- Base classes of transactional classes and sub-classes of transactional classes or interfaces must be transactional as well. This prevents a programmer from calling non-transactional methods of the base class or sub-class when expecting guaranteed consistency. It also allows to declare any implementation of an interface to be transactional. Methods of the super class `Object` are overridden by transactional methods or prohibited when final.
- Type arguments of transactional generic classes must be either of type `String` or transactional. Therefore, a transactional generic class declaration must have at least one transactional type bound (e.g. `<`*`T`* **`extends`** `Transactional>`) or a type bound of type `String`. Generics allow to implement type-safe transactional versions of standard interfaces such as `Collection`, `Set`, `List` or `Map`.

Use of non-transactional irrevocable operations is implicitly prevented by the rules. Additional rules are required to guarantee deadlock-freedom. This is not addressed here but is needed to meet the requirement of a simple API. Therefore, explicitly locking or suspending of threads inside transactional objects using `wait`, `notify`, `notifyAll`, or keyword **`synchronized`** is prohibited. There are already approaches to support blocking transactions and irrevocable transactions (see Section 2.2) but those are not addressed in this paper.

3.3 Ramifications on Software Development

The proposed programming model prevents a programmer from errors concerning privatization or publication because it is impossible to use shared transactional data in an unprotected local fashion. In contrast to other approaches the proposed programming model primarily expects the data to be declared transactional and not only the critical code sections which leaves the responsibility to avoid those problems at the application code.

The programming model guarantees consistency for individual transactional objects only. To achieve consistency of an operation over a set of transactional objects a programmer still needs to move the operation into a transactional method. Appropriate rules, to force a programmer to do this could consist of an access restriction on transactional objects to transactions and the requirement for explicit declaration of intended non-transactional access to transactional objects. But both rules are at odds with the requirement to enable the implementation of self-contained libraries with transactional objects.

Currently the only way to privatize data and to get the performance advantage of unprotected data access is to create a corresponding non-transactional class to receive a copy of the data of a transactional object. This method will be applicable on certain but not all use cases because copying may produce a lot of overhead and narrows the intended speedup. There exist other feasible techniques to support privatization and publication through the STM system (cf. [12]). Those techniques have smaller per-call overhead than transactions and very low initial overhead compared to copying. Thus, a preferable alternative is to provide such a mechanism and offer an appropriate simple API for the following operations:

- **privatize:** Switches visibility mode of the transactional object to private, i.e. moves the object into the local area of a thread. Exactly one thread gets the owner of the object and has exclusive access to it. Transaction management on this object is turned off but the correct ownership of accessing threads is still checked. A thread trying to access a privatized object of another thread causes a runtime exception to occur.
- **check visibility:** Allows every thread to identify the current owner of the object, e.g. to check if the data is currently privatized. This is the only operation of a privatized object available to threads not owning the object.
- **publish:** Switches visibility mode of the transactional object back to public and returns it into the shared area. The method is available to the current owner of the object as long as the object is private. It turns on transaction handling for this object.

All three operations should support transaction handling. This allows for example to perform the visibility check and the intended access to the data in a transaction.

Privatization and publication follow a general pattern which is supported by these operations: The visibility of an object is indicated by a flag which must be considered by all threads prior to an access to a potentially privatized object. A thread can neither privatize, publish or access an object owned by another thread. To prevent programmers from any error in usage with this pattern each access violation according to this pattern leads to a runtime exception. There are other patterns which rely on the modification of the reference on the shared object. When the reference on a privatized object is redirected to a new object or null or has been removed from a list, the object is simply no longer reachable for other threads. To get the performance advantage in this case the programmer still needs to use the privatization operation with the modification of the reference. Thus, combined with transactions these operations are enough to support all use cases addressed by privatization.

By the restrictive rules of the programming model, a programmer is forced to decouple transaction enabled code from unprotected code. Thereby a programmer merges transactional data and related code which should be protected by transactions in the transactional domain. Each operation on transactional data will be delegated to transactional methods inside the transactional domain. In larger ecosystems, such as the Eclipse platform, there will exist even a three level architecture as depicted in Figure 3. A lower level will provide simple transactional objects with just getter and setter methods representing the shared data model (data objects). The programmer of a specific software package for the ecosystem will create some additional transactional objects offering transactional methods for complex operations of its specific application on top of this level (data access services). The non-transactional application logic operating on the transactional shared data model will be on the highest layer (application objects). Thus, there will be three layers separating basic data objects from some application specific data access service objects and the application logic on top.

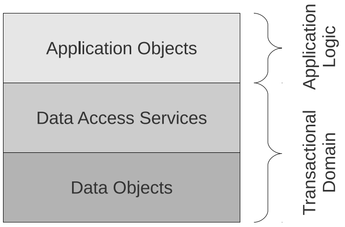

Figure 3 Sample application

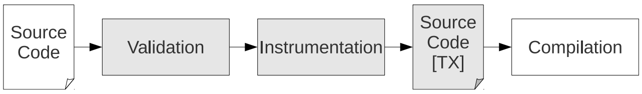

Figure 4 Toolchain for the compilation

4 Prototypical Toolchain

To support the designed programming model a prototypical toolchain has been developed (see Figure 4). It consists of a validation tool which checks the given source code for conformance with the coding rules (*Validation*) and an instrumentation tool which injects code for transaction handling in transactional objects (*Instrumentation*) prior to compilation.

Source code that does not pass the validation is not given to the instrumentation. An advantage of the separation of both tools is that the instrumentation tool can be easily exchanged with an instrumentation tool using another instrumentation style such as byte code instrumentation or supports another STM library.

The usage of AtomJava, which supports source code instrumentation, allows the debugging of instrumented classes. The following modifications and extensions were incorporated in AtomJava to enforce the guaranteed strong isolation and modularity of transactional objects.

- The STM library now supports transactional constructors keeping the order of constructor calls in respect to the inheritance dependencies.
- The initialisation expressions of Java member variables are executed before a constructor is called. Those expressions may contain method calls to other objects. Because a transactional object must guarantee consistency throughout its life cycle, the initialisation of member variables of a class was made transactional as a whole.
- Initialisation of static variables of a class takes place at class loading time. To guarantee consistency of those initialisations too, they are now put into one transactional method executed at class loading time.

- AtomJava replaced the main thread by an instance of `AThread` requiring an instrumentation of method `main`. This provides fast access to thread context information but it requires modifications in the environment. The dependency on class `AThread` was removed and a thread context object stored in thread local storage has been introduced.
- AtomJava uses a special strategy for lock handling to implement deadlock detection. A thread holding locks keeps them as long as no other thread signals its need for it even when the transaction has already been finished. The owner of the lock constantly polls for lock requests of other threads and releases them on demand. This required instrumentation of any source code which uses the STM and is in conflict with the modularity requirement. Therefore, all locks are made available immediately after a transaction has completed and the instrumentation of calling source code is discarded.

5 Evaluation of the Prototype

The compliance with the functional requirements such as avoidance from privatization and publication, consistency of class and object instantiations, has already been tested during development. Of major interest for the evaluation was the impact of required modifications to the STM library even with this first not optimized prototype. A second interest was to show the difference of source code instrumentation and byte code instrumentation which actually provides more possibilities for optimizations.

The evaluation was performed on a shared memory system with two Intel Xeon Quadcore X5365 processors at 3.00GHz, 16GB DDR RAM memory, a Linux Debian operating system with 64 bit kernel in version 2.6.26-2 supporting SMP and a Sun Java VM version 1.6.0_21 also supporting 64 bit using the incremental garbage collector to get more precise results.

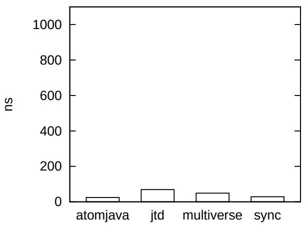

Figure 5
Method calls

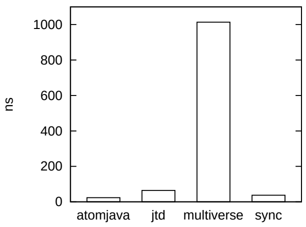

Figure 6
Object instantiations

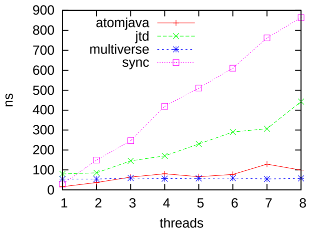

Figure 7
Hotspot behaviour

Measurements were made over 10^7 iterations of a loop over a single test unit (method or constructor call) which have been repeated one hundred times to check the precision of the calculated average. Measured constructors and methods contained a read and a write operation on a four byte integer instance variable. All measurements were performed for instrumentations with AtomJava (*atomjava*), our modified version (*jtd*), Multiverse with byte code instrumentation (*multiverse*) and for comparison with `synchronized` versions of methods and constructors (*sync*) which provides similar abstraction. The difference of the calculated average to the average duration of the same method or constructor without concurrency control provided the average overhead of a concurrency control mechanism.

Figure 5 and 6 show the average overhead of method calls and object instantiations over ordinary calls or instantiations without concurrency control. As expected the modified version shows a significant additional overhead compared to the original AtomJava instrumentation but it is still similar to the overhead produced by Multiverse and even lower in case of transactional object instantiations.

When the number of threads concurrently calling a method is increased the impact of the access to thread local storage is getting more obvious (see Figure 7). This is the main difference to the original version of AtomJava. While Multiverse shows a nearly constant behaviour the overhead of the modified AtomJava version increases with each additional thread but is still far less than the overhead of the synchronized version.

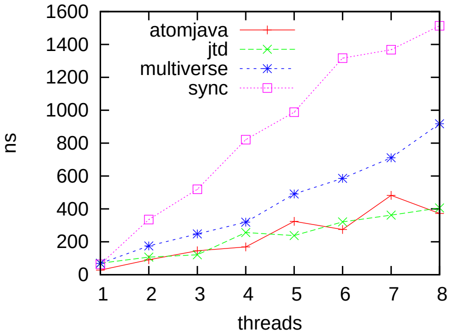

Figure 8 Load distribution 1:10

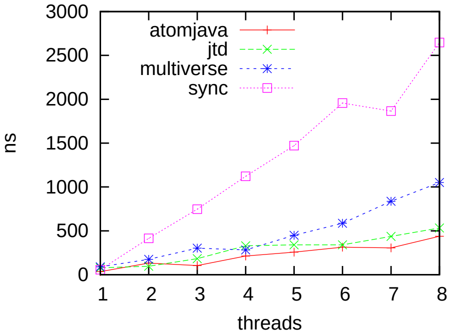

Figure 9 Load distribution 10:1

STM is known to have problems with so-called hotspot objects, which are rapidly accessed by many threads. Therefore, another measurement was performed where the amount of operations a thread performs inside and outside the instrumented method was synthetically controlled by iterations of a loop with read/write operations (approximately ten Java byte code operations) on a shared four byte integer variable. Figure 8 shows an example where the load inside the instrumented method is ten times higher than the load the thread spent outside of the method. In this case the modified version performs better because the effort to access thread local storage is lower compared to the total work inside the transaction. It is even better than Multiverse and has nearly the same performance as the original AtomJava STM system. When increasing the work outside the method as depicted in Figure 9 the overhead of the instrumentation by the modified version rises again due to the same reason of the increased access to the thread local storage.

6 Summary and Future Prospectives

This paper proposes an alternative programming model for Java STMs which prevents a programmer from errors with privatization and publication in use of STM. The fundamental concept is to keep transactions and transactional data in a closed domain. The model relies on source code instrumentation to provide a simpler API. The concept keeps libraries using STM self-contained preventing an impact on software which uses the STM-based library and the target runtime environment. Ramifications on the software development process are discussed and a proof of concept is given by a prototypical implementation of a toolchain consisting of a validation tool and an instrumentation tool based on AtomJava. The evaluation of the prototypical instrumentation tool shows a significant overhead due to the necessary modifications to the original instrumentation tool which is still lower as the overhead produced by Java monitors.

Existing STM systems have been predominantly developed with the focus on performance and transaction throughput and already reached acceptable quality compared to traditional locks. But the complexity of the APIs is still high and significantly reduces the advantages over traditional locks such as deadlock-freedom. The work presented here is an attempt towards a simpler API preventing a programmer from errors while using an STM system. The overall goal of this work is to ease the move to parallel programming. But there are still open issues left which are considered in the design of the programming model and will be addressed in subsequent work. The issues include support for privatization and publication by the STM system as described above, blocking transactions, irrevocable transactions, and long running transactions. For all concepts elaborated approaches already exist and it may be possible to improve them in order to achieve simpler error-preventing APIs.

References

1 A.-R. Adl-Tabatabai et al. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 26–37, New York, USA, 2006. ACM.

2 B. D. Carlstrom et al. The Atomos transactional programming language. *SIGPLAN Not.*, 41(6):1–13, 2006.

3 C. Cascaval et al. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008.

4 V. Menon et al. Practical Weak-atomicity Semantics for Java STM. In *SPAA '08: Proc. Annual Symp. on Parallelism in Algorithms and Architectures*, pages 314–325, New York, USA, 2008. ACM.

5 T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proc. 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 48–60, New York, USA, 2005. ACM.

6 M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

7 B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *MSPC '06: Proc. Workshop on Memory System Performance and Correctness*, pages 82–91, New York, USA, 2006. ACM.

8 M. Herlihy; V. Luchangco and M. Moir. A flexible framework for implementing software transactional memory. *SIGPLAN Not.*, 41(10):253–262, 2006.

9 A. Welc; B. Saha and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA '08: Proc. 20th Annual Symp. on Parallelism in Algorithms and Architectures*, pages 285–296, New York, USA, 2008. ACM.

10 G. Korland; N. Shavit and P. Felber. Noninvasive java concurrency with deuce stm - an extensible java stm framework. poster. In *SYSTOR 2009: The Israeli Experimental Systems Conference*, 2009.

11 N. Shavit and D. Touitou. Software transactional memory. In *PODC '95: Proc. 14th Annual ACM Symp. on Principles of Distributed Computing*, pages 204–213, New York, USA, 1995. ACM.

12 M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. In *PODC '07: Proc. 26th Annual ACM Symp. on Principles of Distributed Computing*, pages 338–339, New York, USA, 2007. ACM.

13 P. Veentjer. Multiverse - Software Transactional Memory for Java, 2009. `http://multiverse.codehaus.org`.