Synchronized-by-Default Concurrency for Shared-Memory Systems

Martin Bättig
Department of Computer Science
ETH Zurich, Switzerland

Thomas R. Gross
Department of Computer Science
ETH Zurich, Switzerland

Abstract

We explore a programming approach for concurrency that synchronizes all accesses to shared memory by default. Synchronization takes place by ensuring that all program code runs inside atomic sections even if the program code has external side effects. Threads are mapped to atomic sections that a programmer must explicitly split to increase concurrency.

A naive implementation of this approach incurs a large amount of overhead. We show how to reduce this overhead to make the approach suitable for realistic application programs on existing hardware. We present an implementation technique based on a special-purpose software transactional memory system. To reduce the overhead, the technique exploits properties of managed, object-oriented programming languages as well as intraprocedural static analyses and uses field-level granularity locking in combination with transactional I/O to provide good scaling properties.

We implemented the synchronized-by-default (SBD) approach for the Java language and evaluate its performance for six programs from the DaCapo benchmark suite. The evaluation shows that, compared to explicit synchronization, the SBD approach has an overhead between 0.4% and 102% depending on the benchmark and the number of threads, with a mean (geom.) of 23.9%.

Categories and Subject Descriptors D1.3 [*Concurrent Programming*]: Parallel programming

Keywords Atomic blocks, Concurrency, Parallel programming, Synchronization, Transactional memory

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, contact the Owner/Author(s). Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481.

PPoPP '17, February 04 - 08, 2017, Austin, TX, USA
Copyright © 2017 held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4493-7/17/02…$15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/3018743.3018747

1. Introduction

Explicit synchronization of threads using locks or atomic blocks is error prone. Missing synchronization may lead to data races or race conditions, while incorrect locking can lead to deadlocks. In this paper, we explore the synchronized-by-default (SBD) approach to concurrency: Instead of requiring the programmer to insert synchronization explicitly, the system synchronizes all shared memory accesses by default. Thus, increasing concurrency requires the programmer to relax synchronization explicitly. To realize this behavior, each thread executes all its instructions within atomic sections, even if the instructions have external side effects. By default, there is a single atomic section per thread. These atomic sections have transactional semantics and therefore are free from concurrency bugs such as data races or race conditions. Furthermore, they can be rolled back to resolve deadlocks. To increase concurrency, a programmer can insert instructions that split the atomic sections dynamically at runtime.

The SBD approach has been explored before [16, 23, 25, 26], to deserve further attention, it must be able to compete with explicit synchronization. The difficulty lies in implementing the approach in a way that reduces the synchronization overhead and supports scalability to the extent provided by explicit synchronization.

Synchronization overhead. Executing all instructions within atomic sections adds a large amount of synchronization overhead. Since atomic sections have transactional semantics, hardware transactional memory (HTM) [21] could be a solution. However, suitable unbounded HTM is not readily available [12], and using simulated HTM makes a comparison against explicit synchronization difficult, as it does not run on the same hardware. An alternative is to use a software transactional memory (STM) [30] as it runs on existing hardware. However, even fast STM implementations [8, 11, 29, 36] are optimized for language integration using the atomic block model [19], i.e., only a fraction of the memory accesses requires synchronization, thus such STMs can tolerate a higher overhead than is acceptable by the SBD approach.

Scalability. The approach must provide the means that allow a programmer to reduce the contention and eliminate deadlocks between concurrently running atomic sections.

Programming effort. The approach should simplify concurrent programming. Thus, it should add only a reasonable amount of additional language constructs over a language using explicit synchronization. Additionally, the approach should be self-contained in that all kinds of parallel programming idioms can be expressed. If a programmer must revert to explicit synchronization, e.g., to execute I/O operations, this step adds additional complexity as the programmer must deal with two concurrency models.

Our claim is that the SBD approach is practicable for realistic application programs because: (i) synchronization became cheaper [9], (ii) shared memory accesses make up only a part of applications, (iii) we can simplify the comparison against explicit synchronization by limiting the number of new language constructs and by adapting and reusing threading constructs known from explicit synchronization, and (iv) the approach can be implemented on existing hardware. The key contributions of this paper are:

- It describes how this approach can be integrated in a realistic, managed language like Java with only few additional language constructs and discusses how to make common concurrency constructs and operations with external side effects compatible with this approach.
- It shows how a special-purpose STM can be combined with compiler optimizations to reduce the synchronization overhead of the SBD approach to enable scalable applications.
- It compares a version of Java adapted for the SBD approach against regular Java with explicit synchronization using locks for six benchmarks from the DaCapo benchmark suite [2]. We made the evaluation in terms of sequential overhead, scalability, programming effort, and memory overhead.

2. Background

The SBD approach is based on previous work [16, 23, 25]: it executes all instructions of a program within atomic sections and uses the concept of splitting atomic sections. Unlike previous approaches, we focus on integrating these two concepts into a managed, object-oriented language using threads. We do this by combining them with the following additional concepts:

- Specific semantics for memory accesses.
- All instructions executed within atomic sections, even those with external side effects, such as I/O.
- Reusing and adapting existing thread control operations.
- Refinement of atomic sections: No nesting, and no splitting within constructors (to reduce overhead).

2.1 Atomic Sections

We describe the additional concepts in Section 3. In this section, we discuss the atomic sections and their integration. An atomic section is opaque [15], i.e., an atomic section S' cannot see any kind of modification (shared memory or I/O) made by a concurrently running atomic section S'' until S'' ends. Thus, atomic sections have transactional semantics, and the system can abort them and undo their modifications. In contrast to atomic blocks [19], a common form of STM language integration, atomic sections do not have a fixed start or end in the program code. Each thread T has a currently active atomic section S_A. Upon start of T, the system creates a new atomic section S and assigns it to T: $S_A \leftarrow S$. Thread T executes all its instructions within its S_A. Further, thread T can execute a `split` operation. Splitting ends S_A, and creates an atomic section S' that becomes the new active atomic section: $S_A \leftarrow S'$. Upon the end of thread T, its atomic section S_A ends as well. The main usage of splitting an atomic section S is to increase the concurrency by releasing resources that S acquired to ensure its transactional execution, and to make modifications as well as external side effects visible.

```
1  class Worker extends Thread {
2   static int processed;
3   void canSplit run() { // atomic section begin
4    for(Request req : getRequests()) {
5      processRequest(req);
6      ++processed;
7      split; // (explicit) atomic section end
8             // new atomic section begin
9     }
10  } // atomic section end
11  // in main:
12  Thread a = new Worker().start();
13  Thread b = new Worker().start();
14 }
```

Figure 1: Example of implicit and explicit atomic sections.

Figure 1 shows a small example of two threads that process requests. Without the `split` operation in line 7, the modification of shared field `processed` in line 6 would serialize the execution of threads `a` and `b`. The `split` instruction in line 7 ensures that resources that protect field `processed` are freed in each loop iteration, and consequently both threads can process requests concurrently (assuming eager conflict detection [28]).

The concurrency provided by SBD approach has the following properties:

Conflict resolution enforced. Simply dividing the work between multiple threads is not necessarily sufficient to enable parallelism. As Figure 1 shows, a programmer is forced to deal with frequently occurring conflicts that limit parallelism.

Asymmetric. The system resolves infrequent conflicts (e.g., program initialization, or configuration changes at runtime) automatically, while frequent conflicts may be more complicated to deal with. Thus, programmer effort with regard to concurrency is asymmetric in SBD compared to explicit locking, where every possible conflict must be found and dealt with.

Incremental. The approach allows to add concurrency incrementally: Missing `split` instructions lead to serialization, but never to data races or race conditions, unlike explicit synchronization, where missing or incorrect synchronization eventually results in a concurrency bug.

Explicit. While concurrency bugs like data races or race conditions can occur nevertheless, splits are the only instructions that can cause them. Splits are visible in the source code, thus, one can check what memory locations are used across a split at places where they should not be used. Therefore, pinpointing the cause of a concurrency bug is more explicit with SBD than with explicit synchronization.

2.2 Preventing unexpected split operations

The presented SBD approach does not allow the nesting of atomic sections. If a method A is in atomic section S' while invoking method B, and B executes a `split` operation, this operation ends atomic section S' and starts section S''. An example is atomic section B_1 shown in Figure 3 that starts in method `run` and ends in method `processPosition`. Therefore, it is necessary to prevent methods, e.g., in a library, from issuing unexpected split operations, as such operations can introduce concurrency bugs.

Modifier canSplit. Only designated methods, which are identified by the method modifier `canSplit` in their signature, may issue a `split` operation either directly, or by invoking a method having the `canSplit` property. Thus, methods not having the `canSplit` property are guaranteed not to issue splits. Thread and task entry points have the `canSplit` property set by default. Constructors cannot have this property to prevent uninitialized instances from escaping an atomic section. And finally, a method that has the `canSplit` property can only override a method that also has the `canSplit` property.

Modifier allowSplit. Further, a method A that has the `canSplit` property requires additional protection. If a method A invokes another method B, it must declare whether it allows B to execute `split` operations, by prefixing the invocation with the keyword `allowSplit`. Checking this at compile time makes it impossible to add the `canSplit` property to an existing method (e.g., in a library) without breaking the API.

Figure 2 shows an example of these modifiers. Here we assume requests contain multiple `Articles`, and conflicts between `Article` instances are frequent. To increase the possible concurrency, we uncomment the modifier `canSplit` of method `processRequest` (line 7) as well as the `split` instruction (line 10). Finally, we uncomment the modifier to the call (line 15). Figure 3 shows atomic sections created for an example execution, with time progressing from top to bottom. Timeline (a) shows the situation for the unmodified case: there is one atomic section per `processRequest`. Timeline (b) shows the situation after uncommenting. Now the system executes each invocation of `processPosition` in a separate atomic section.

```
1  void processPosition(Article a, int num) {
2    if (a.available > num) {
3      a.available -= num;
4      positions.add(a, num);
5    }
6  }
7  void /*canSplit*/ processRequest(Request r) {
8    for(RequestItem ri : r.getItems()) {
9      processPosition(ri.article, ri.quantity);
10     /*split;*/
11   }
12 }
13 public canSplit void run() {
14   for(Request req : getRequests()) {
15     /*allowSplit*/ processRequest(req);
16     ++processed;
17     split;
18   }
19 }
```

Figure 2: Inserting split operations, changes in comments.

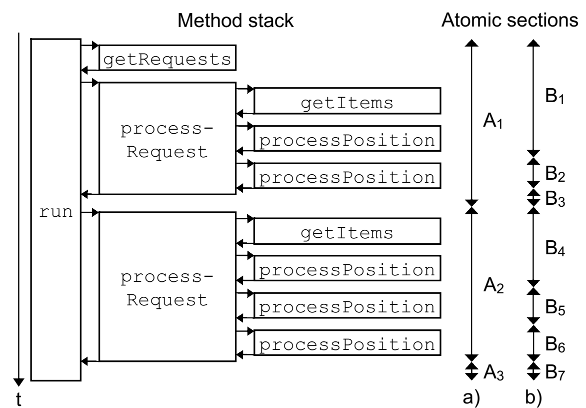

Figure 3: Nested methods vs. non-nested atomic sections.

3. Concepts

In this section, we elaborate the concepts that allow us to address the difficulties described in Section 1. We define the target system, present how we realize the atomic sections using a special-purpose STM system, describe how to reduce the synchronization overhead, and show how to support I/O and thread operations.

3.1 Target applications and programming languages

The first part of the solution is to restrict the class of applications and programming languages that an implementation of the SBD approach should support.

We focus on application programs that use a mix of computations, I/O operations, and memory accesses, e.g., a web server. Many programs fall into this class; the combination of different kinds of activities ensures that shared memory accesses make up only part of the program, thereby reducing the impact of the additional synchronization.

Further, we assume a managed language that uses a garbage collector. Using such a language allows exploiting guarantees regarding memory accesses, e.g., no pointers to arbitrary memory locations. These guarantees allow static analysis to detect and remove superfluous synchronizations.

3.2 Atomic sections

We use transactional memory (TM) to implement the atomic sections. Due to its transactional semantics, an atomic section corresponds directly to a transaction of a TM system. TM systems differ greatly in properties such as conflict detection, version management, etc. In the SBD approach, these properties have a substantial impact on how to implement a program, both from a functional and performance point of view. Thus, they become an integral part of the language semantics, and we cannot choose them arbitrarily. To determine the TM properties, we start by defining the following semantics for memory accesses:

1. Each memory location M that requires synchronization has an associated lock L. Before accessing memory location M, the system acquires lock L if not held already.
2. The system executes memory accesses that require synchronization in the same order as specified in the program.

The first point of this semantics matches pessimistic concurrency control. The second point implicitly allows a programmer to order memory accesses in a way that prevents or reduces the number of transaction aborts due to deadlocks without having to introduce additional language constructs.

To realize atomic sections with the above semantics, we use a special-purpose STM. The reason is that currently available HTM systems (e.g., Intel TSX), or fast STM systems [8, 11, 29, 36] do either not support the combination of the following TM properties, or are not optimized for them:

Conflict detection. Pessimistic concurrency control requires eager conflict detection [28]. The above defined semantics for memory accesses requires visible readers [32] to enforce the ordering of the memory accesses. As conflict detection granularity, we chose field- respectively array element-level granularity. Such a granularity prevents lock contention due to false-sharing compared to instance level locking, thereby avoiding artificial scalability issues that may arise because all instructions execute within a transaction.

Memory access type	**Check**	**Lock**	**Undo**
Non-final field or array element	✓	✓	✓
Final field			
New non-final field / array element	✓		
Local variable (with canSplit)			✓
Local variable (w/o canSplit)			

Table 1: Synchronization per memory accesses type.

Version management. We use eager version management [28], as the system must apply the undo log only in case of an abort. Since we defined a semantics that allows preventing aborts by ordering memory accesses, using eager version management allows manual optimization of a program if required. The overhead of a program with only few aborts consists mainly of locking, updating the undo log, and unlocking.

Progress guarantees. We chose a deterministic deadlock resolution policy since readers are visible. Deadlock resolution already partly ensures progress, but live locks can still occur due to the *friendly-fire* and *starving writers* performance pathologies [4]. To prevent *friendly-fire*, the system uses the following deadlock resolution policy: Always abort the youngest transaction that is part of a deadlock. Thus the oldest transaction, and therefore the program itself, can always make progress. To prevent *starving writers*, the TM uses fair queues: If a thread cannot acquire a lock, the system enqueues it at the end of the waiting queue, regardless of the operation being a read or a write. The only exception to this rule are upgrading readers. The system enqueues them in front of the queue to reduce the number of aborts.

Atomicity guarantees. The STM does not have to provide atomicity guarantees [3]. In the SBD approach all instructions execute within atomic sections. Non-transactional execution is not possible.

3.3 Reducing the synchronization overhead

In the following, we describe the concepts used to reduce the synchronization overhead:

Exploit language level memory access properties. We assume a managed, object-oriented language with the following memory access properties:

- Stack: The stack of a thread is isolated. A thread cannot access stacks of other threads.
- Heap: The language must not allow direct memory access but must provide indirect access to object and array instances via reference pointers. These point either to valid instances or to null. Access to properties of object instances is only possible through their fields. Access to content of array instances is only possible within their bounds and only for complete elements.

Accesses to local variables do not need synchronization because of the isolated stacks. Methods that have the canSplit property, and thus can issue `split` operations, must nevertheless save the old values of modified local variables in the undo log to be able to rebuild the stack in case a transaction is restarted. On the other hand, heap accesses via fields or array elements require synchronization including saving of old values in case of a modification. There are two exceptions: (i) accesses to fields declared as final do not need synchronization, since constructors cannot have the `canSplit` property, other transactions can see only initialized final fields; (ii) accesses to fields or elements of newly instantiated objects or arrays within the same transaction. These accesses do not require synchronization but instead need a dynamic check to determine whether an instance is new or not. Thus non-final field and array element accesses are the main source of synchronization overhead. Table 1 summarizes these rules.

Compile time optimizations. Compile time optimizations can remove unnecessary checks whether a field or array element is already locked. We use the following optimizations:

1. A dataflow analysis to remove redundant checks: An access A to a field or array element X requires synchronization unless the system can prove that X is already synchronized in all possible control flow paths that lead to A. This optimization makes use of the `canSplit` property: As methods without this modifier cannot issue `split` instructions, removal of the synchronization is possible even if such a method lies in a control flow path leading to A.
2. Move lock operations out of loops if the locking order can be preserved.
3. Combination of subsequent field accesses on the same instance to eliminate checks whether an instance is new.

These optimizations are intraprocedural. They benefit from method inlining and are suitable for dynamic compilation.

Combine read- and write-sets with locking information. With pessimistic concurrency control, each access to a non-final field or array element requires synchronization. This involves checking whether the accessed memory location is already in the read- or write-set of the transaction, and if not, to acquire the lock and to update the read- or write-set. We combine these operations by storing the set information directly on the accessed instance instead of storing it in a separate structure. Figure 4 (a) shows the instance layout with the additional locking structures. Each non-final field or array element has its own locking structure. The additional indirection, via the field `locks`, allows lazy allocation of the locking structures of an instance, thereby reducing the amount of memory required in case an instance is transient in a transaction.

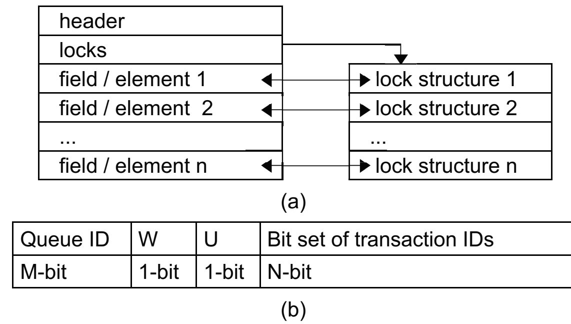

Figure 4: Memory layout of an instance (a) and a locking structure (b).

Figure 4 (b) shows the lock structure. It contains a bit set that encodes whether a transaction has a lock on the associated field or not and whether it is in the read- or write-set of the transaction. Each bit corresponds to a unique, but reusable ID that represents a transaction. The size of the bit set is limited, so that the size of the locking structure does not exceed the limit of a single compare-and-swap (CAS) operation. The fixed bit set size reduces the number of required instructions for a locking operation but also limits the number of transactions that can concurrently be active to the size of the bit set. Beside the bit set, a flag W indicates whether a write lock, or a read lock, is in place. The upgrader bit U [29] is optional but helps to detect *dueling write-upgrades* [4] early. To enable fair scheduling the lock structure includes a queue ID. If a queue ID is set, other transactions are waiting to acquire the lock on the specific queue and directly acquiring the lock is not possible. The maximal number of queues is fixed and equals the maximal number of concurrently active transactions, as in the worst case all concurrently active transactions are enqueued before deadlock resolution begins.

The number of available transaction IDs N limits the achievable actual parallelism. However, this limit only affects systems with P processing units, where $P > N$. On such a system, the actual parallelism of a program cannot exceed N. If no transaction ID is available, a transaction must wait until one becomes free. This strategy works because:

1. No nesting is possible: Once a transaction acquires an ID it is able to finish without acquiring another ID.
2. If a thread T_1 waits for a thread T_2 to update a condition C, T_1 must end its current transaction as otherwise T_2 cannot update C. Ending the transaction, T_1 frees a transaction ID, thus T_2, even if it was waiting for an ID to become free, can complete its work.

The above described locking structure allows to implement a locking operation using very few instructions, as shown in Figure 5: (1) The system checks whether an instance is newly allocated in the current transaction (`a.locks==null`). Newly allocated instances do not require locking. (2) Oth-

erwise, if the lock is unallocated (`a.locks==UNALLOC`), the system performs the lazy allocation. `UNALLOC` is a constant pointer. (3) Then the system checks whether the transaction must acquire the lock (`(lock & trx.mask))==0`). (4) If that is the case, the system acquires the lock by checking whether it is free and then executing a CAS. If the lock is not free or the CAS fails, the thread is enqueued. The system records lock acquisitions in the undo log together with the old value of the field or array element in case of a write.

```
1  // (1) allocated in previous transaction?
2  if (a.locks != null) {
3   // (2) lazy lock init
4   if (a.locks == UNALLOC /*const pointer*/) {
5    CAS(a, locksOffs, UNALLOC, new long[size]);
6   }
7   // (3) already locked?
8   long lock = a.locks[lockID];
9   if (((lock & trx.mask)) == 0) {
10   // (4) try to lock or enqueue
11   if ((lock | trx.mask) != trx.mask
12    || !CAS(a.locks,offs,lock,lock|trx.mask))
13   {
14     enqueue(a, lockID); // slow-path
15   }
16   updateUndoLog(offs, lock, fldOffs, oldVal);
17  }
18 }
```

Figure 5: Locking operation in pseudo code.

3.4 I/O and non-transactional code

In the SBD approach, no code can run outside of an atomic section. This rule applies also to instructions with external side effects, e.g., irreversible I/O operations or (non-transactional) operations invoked using a foreign function interface. There are two viable options to do so: Inevitable (irrevocable) transactions [33], and buffering using transactional wrappers [18].

Implementation of inevitable transactions using pessimistic concurrency control in combination with read/write locks is straightforward [10] but has the problem of limiting actual concurrency. At most one transaction can be inevitable at any given moment in time. E.g., two or more transactions cannot execute I/O at the same time, even if they use different devices. To achieve good scalability, we use transactional wrappers instead, i.e., buffering irreversible actions until an atomic section finishes. Wrappers are handwritten, and must implement an appropriate locking scheme that ensures the atomicity and isolation of the performed external side effects or any returned values.

Using transactional wrappers results in the following two notable changes for a programmer:

1. If an output is made in an atomic section S an observer can see the output only after S ends.
2. All irreversible operations must use transactional wrappers.

The first point causes a small API change for all functionality that has external side effects, as these functions, depending on the situation, may require an additional `split` operation to make the effects visible. Single threaded programs require these additional `split` instructions as well. The second point primarily concerns developers of system libraries but not application developers, as system libraries of a language implementing the SBD approach must already contain such transactional wrappers, e.g., for file system operations.

3.5 Thread operations

Thread operations in the SBD approach work similarly as with explicit synchronization using locks but require minor adaptation. In the following, we describe the adaptations for a number of common operations.

Thread or task start. When an atomic section S' starts a new thread or task T, the start of T is deferred until S' ends. Deferring of T simplifies its reversal, since an instantaneous start of T by S' has the following implications (S'' is the initial atomic section executed by T and is independent of S'):

1. Aborting S' requires aborting T as well, thus S'' of T cannot end until S' ends.
2. If S'' makes modifications, these can become visible only after S' ends.
3. S'' can acquire locks that S' requires after it started T, thus creating a deadlock.

Deferring the start of T until the end of S' avoids this additional complexity (items 1 and 2 above) and the problem (3) of an instantaneous start. Additionally, in many cases an atomic section S' that starts T already holds locks on data that T is supposed to work on. Thus, a deferred thread start does not impose a restriction as only after S' ends, T can start to work on this data. The thread T will always start after S' ends, although its atomic section S'' may be blocked initially, if no free transaction IDs are available. AME [23] uses a similar mechanism to initiate asynchronous method calls.

Thread or task join. A thread or task join operation always issues a `split` operation before waiting. This ensures that the thread or task T to wait for has started. Additionally, the `split` operation causes the waiting thread to release its transaction ID while waiting, thereby ensuring that T can execute.

Sending of a signal. The system defers sending of a signal N until the current atomic section S ends. This deferring avoids unnecessary notifications if the system aborts S after sending. It also ensures that locks on the waiting condition C acquired by S are freed, and the modifications made by S become visible. This allows the threads that are waiting for N to check C and to continue.

Change	Type	Description
Keyword `split`	Add	Ends current atomic section and starts new atomic section.
Modifier `canSplit`	Add	Allows to issue split instructions.
Modifier `allowSplit`	Add	Allows to invoke methods issuing split instructions.
Field access (instance or static)	Change	First lock field (read or write) then access field.
Array element access	Change	First lock array element (read or write) then access array element.
Thread control operations	Change	Change as described in Section 3.5
I/O operations	Change	Use wrapper to execute I/O operations transactionally.
Foreign code execution	Change	Use wrapper to execute non-transactional library operations transactionally.
Explicit synchronization constructs	Remove	Both language-based, and library code.
Direct memory access	Remove	E.g., volatile access properties.

Table 2: Changes required to adapt a managed object-oriented language to the SBD approach.

```
1  class Barrier {
2    private final int expected;
3    private int arrived;
4    public Barrier(int expected) {
5        this.expected = expected;
6    }
7    public canSplit void sync() {
8      arrived++;
9      if(arrived < expected) {
10       while(arrived < expected) {
11         wait(); // split atomic section
12       }
13     } else {
14       notifyAll();
15       split; // split atomic section
16     }
17   }
18 }
```

Figure 6: Example of `notifyAll` and `wait` usage.

Waiting for a signal. A thread T that waits for a signal first issues a `split` operation, thereby releasing all locks it holds, including the ones on the waiting condition C. This step ensures that another thread T' can update condition C. Additionally, the `split` operation causes the T to release its transaction ID thereby ensuring that T' can execute.

Figure 6 shows an example of signaling by way of a barrier implementation. The `notifyAll` (line 14) must release the lock on the field `arrived` to allow the waiters to test the condition (line 10) after signaling. A waiter must release the lock to allow other threads to update the condition `arrived` (line 8).

Thread local memory. The only adaptation to thread local memory is an undo buffer that allows restoring the state in case of an abort.

3.6 Summary

Table 2 summarizes the changes required to adapt a managed object-oriented language using explicit synchronization using locks to the SBD approach.

3.7 Composability

Composability [20] is a feature of TM that allows to combine two or more atomic operations by composing them into a larger atomic operation, i.e., executing them within the same transaction. Extending the SBD approach to allow composition of atomic sections (the smallest atomic operation in SBD) is helpful in case of library methods that perform split operations. A way to compose a function `canSplit f()` and `canSplit g()` into a single atomic section is to extend SBD by adding a block level statement `noSplit{...}`. Within such a `noSplit` block, the system ignores `split` instructions. Care has to be taken because certain methods must be able to split, e.g., a method that sends data over the network and expects a response. Therefore, to be safe, a programmer must explicitly allow execution of a method within such a `nosplit` block, e.g., by an additional property `splitOptional`.

4. Implementation

To evaluate the SBD approach, we use a proof of concept implementation based on the Java language. This implementation runs on an unmodified Oracle JVM. It consists of three components: A bytecode transformation tool to add synchronization, the STM runtime and adapted versions of those classes of the Java Class Library (JCL) that the benchmark programs require. In this section, we describe the relevant parts of this implementation.

4.1 Bytecode transformation

The bytecode transformation tool inserts code to interface with the STM runtime, and performs all code optimizations as described in Section 3.3. The tool uses the Soot framework [34] as basis.

The adaptation of Java arrays to the instance layout as described in Figure 4 (a) requires an additional reference to the locking structures. Altering the array structure is not possible using an unmodified JVM. Therefore, we use wrapper classes to combine an array reference and a reference to an array holding the locking structures of the array elements.

These array wrapper classes replace the regular arrays. To reduce overhead, access to the arrays within the wrappers remains direct (no getters or setters).

Further, to increase the effectivity of the code optimizations, the tool inlines methods statically during transformation. It uses the same inlining decisions for a method as the actual HotSpot Just-in-Time compiler (JIT) of the Oracle JVM would use. The JIT optionally writes its inlining decisions into a compilation log. Based on such a compilation log created during a previous execution, the tool uses the last log of a method for its inline decisions.

Finally, the tool adds support for static initialization. A rollback can revert a static initialization, in which case the system must execute it again. For that, the tool inserts guards before each static access and each constructor call. These guards trigger the static initialization if needed.

4.2 STM details

We implemented the STM as described in Section 3.2. The implementation uses the CAS instruction provided by the `sun.misc.Unsafe` API. The locking structure has a size of 64-bits, the size of the largest CAS operation supported by this API. Therefore, the bit set to store locking information has a size of 56 bits, and the queue ID has a size of 6 bits.

The deterministic deadlock resolution uses a blocking variant of the dreadlocks algorithm [24] modified to work with read-write locks.

4.3 Adaptations of JCL classes

We made the following adaptations to the JCL classes:

- Use the array wrapper classes instead of direct access.
- Implement the changes to the thread operations.
- Implement the transactional wrappers for I/O classes.
- Additional modifications as shown in Table 4.

As the JVM requires some of these classes during startup, we made the modifications to copies of the classes placed under a different namespace to avoid bootstrapping issues. We then redirected the benchmarks to use these copies instead. We only modified JCL classes that we require for the evaluation.

4.4 Implementation of transactional wrappers

For a class C, we manually implemented the transactional wrappers using the following scheme:

1. Implement an adapter class of class C with the same interface, and forward each call.
2. Add a buffer B to save the state before a modification.
3. Before calling a method that queries or modifies the state of an instance, add synchronization to ensure atomicity and isolation. Before modifying, save the state to the buffer B. In case a modification is irreversible or its reversal is nontrivial, the system must defer its execution until the end of the atomic section.
4. Implement commit and rollback operations. Usually, a commit applies deferred operations and clears the buffer B, and a rollback undoes the modifications using data from the buffer B.

E.g., when reading data from a network device, the system places a copy of this data into a buffer B_R. Buffer B_R is associated with the network device. After an abort, the next time the system reads data from the network device it uses the buffer B_R instead. It does this as long as the buffer B_R holds data, then it continues to read from the device. On success, the system discards the buffer B_R. When writing to the network device, the system first writes the data to buffer B_W, thus it defers the write. Only on success, it forwards the data in buffer B_W to the network device. After an abort, the system discards the buffer B_W.

5. Evaluation

To evaluate the SBD approach, we compare it to explicit synchronization using locks with regard to the following four aspects:

1. Programming effort
2. Sequential overhead
3. Scalability
4. Memory overhead

We do this by using multiple benchmarks that originally use explicit synchronization using locks and adapting them to the SBD approach, thus having two variants of each benchmark to compare against each other.

5.1 Setup and testing methodology

The test system has four Intel Xeon E7-4830 CPUs and 64GB RAM. Each CPU runs at 2.13GHz and has eight cores. For all experiments, we enable hyper-threading and disable the turbo mode. We use the Oracle JVM 1.7 for the evaluation. To compare the performance, we measure the steady state performance using the method presented in [13] to reduce the effect of the JIT on the measurements. For each benchmark run, we perform 10 JVM invocations. Per invocation, we require 30 consecutive benchmark iterations with coefficient of variation (CoV) lower than or equal to 0.01. If an invocation does not reach this value after 60 iterations, we use the last 30 iterations.

For testing, we use benchmarks from the DaCapo 09 Benchmark Suite [2] using their default workload. We chose this suite because it contains realistic application programs that we target with the SBD approach (see Section 3.1). We selected six multi-threaded benchmarks with different threading models (task, threads, signaling, custom synchronization, main/worker), and different types of I/O (none, disk, network, database).

We applied the following modifications to all selected benchmarks:

Benchm.	Modification
Tomcat	Use Http11Protocol as Java NIO is not implemented. Use statically compiled JSP pages, as dynamic compilation is not implemented.
PMD	Use single-byte charset as support for multi-byte character sets is not implemented.
H2	Allow a variable number of threads. Use multi-threaded engine and newer version (1.4.193) to reduce contention.

Table 3: Benchmark modifications not relevant for approach.

- Array wrappers as described in Section 4.1.
- Conversion to Soot internal representation, and back to bytecode.
- Static inlining on bytecode level for the optimizations.
- Additional modifications due to implementation restrictions or benchmark enhancements as shown in Table 3.

The above modifications are necessary due to the bytecode transformation, and not due to the SBD approach. To allow a fair comparison, we applied them to both variants. Table 4 lists custom modifications that increase the scalability of the SBD variant. If one of those modifications also reduced the runtime of the explicit synchronization variant, we applied it there as well. The static inlining uses the identical HotSpot compilation profiles for both variants. To generate the profile, we let each benchmark perform 20 iterations in a single invocation.

5.2 Programming effort

To compare the programming effort of both variants, we adapted the benchmarks from explicit synchronization using locks to the SBD approach. Then, we examined the required modifications. We modified only the parts used by the benchmarks, and not the complete code of the underlying application.

Adapting the benchmarks required functional and non-functional modifications. Functional modifications are additional split operations due to the changed I/O and threading behavior. Non-functional modifications decrease the sequential overhead, and increase the scalability of a benchmark by removing lock contention or by preventing deadlocks. Depending on the situation, we use different solutions for these issues:

1. Split as soon as possible after the contented access.
2. Use thread local memory.
3. Reduce update frequency of contented memory location.
4. Reorder accesses to memory locations.
5. Remove functionality if not needed.

We prefer solutions that do not require additional `split` operations (1), since longer atomic sections reduce the number of locking operations. Table 4 lists the custom modifications that we had to apply to the benchmarks to increase their scalability without a `split` operation (2-5). Only one of the custom modifications (disabling of the string cache, *Tomcat*) had a measurable impact on the *baseline* variant (small speed up). All thread local modifications fix scalability issues and do not reduce sequential overhead.

Table 5 shows the number of modifications we applied for each benchmark. To be able to put the number of modifications in relation, the table lists the number of lines of code (LOC) executed by the benchmark. Columns *Split* (*CanSplit*) show how often this instruction (modifier) is added. Column *Custom* shows the number of custom modifications that we had to implement. Column *Final* lists the number of added final modifiers. The bytecode transformer automatically adds the final modifier to private fields that are modified only in the constructor of a class and thus can be declared as final. Beside being good programming practice, this reduces overhead, as final fields do not need synchronization (see Table 1). Finally, columns *Synchronization* and *Volatile* list the number of synchronization constructs required by the baseline variant, which are not needed for the SBD variant.

Modifications to add `canSplit` and `final` modifiers are quite numerous, the former especially if signaling was used deep down in the method hierarchy. Since both type of changes are rather mechanical, they can benefit from code editor support, e.g., by using static analysis to suggest addition of the modifier. Comparing the combined number of `split` instructions and custom changes to the combined number of synchronized and volatile changes shows that the SBD approach and explicit locking have usually a similar amount of synchronization code. The exceptions are LuSearch, LuIndex, and Tomcat were the SBD approach requires a lower amount of synchronization code but on the other hand requires custom modifications (LuSearch, Tomcat), thus showing the asymmetry of SBD mentioned in Section 2.

5.3 Sequential overhead

The sequential (single thread) overhead of the SBD variant consists of the cost of the executed lock operations (see Figure 5). The cost depends on the effect of the locking operations:

Init: Initialize `locks` field of new instance.

Check New: New instances (check only).

Check Owned: Lock already owned (check only).

Acq. & Rls.: Lock acquire, and release (incl. undo).

The sequential overheads of the examined benchmarks are below 100%, even though the STM uses a CAS operation to implement the reader-writer locks. The reason is not only the performance of the STM, but also its usage. Table 6 shows execution time for a microbenchmark. This bench-

Component	Solution type	Description	Count	LOC
JCL	Frequency	Use separate isEmpty flag (instead of size) in get method for empty check.	1	18
	Thread local	Aggregate output to console per transaction to reduce contention. Uses a reusable thread local `OutputStream` aggregation class.	1	5
LuSearch	Thread local	Make shared message digest instance thread local.	1	20
	Reorder	Frequently updated file system directory cache (resolve read/write conflict)	1	3
PMD	Thread local	Thread local of update statistic counters, aggregate on read. Uses a reusable thread local integer aggregation class.	2	4
Tomcat	Thread local	Made tag handler pool thread local.	1	30
	Thread local	Use a separate connection per client thread, instead of connection pool.	1	1
	Thread local	Thread local update of statistic counters, aggregate on read. Uses a reusable thread local integer aggregation class.	7	3-10
	Frequency	Set initialization flag only once.	1	2
	Remove	Disabled the cache in the string manager class.	1	2

Table 4: Custom Java Class Library (JCL) and benchmark modifications. Column *LOC* excludes aggregation classes.

		Synchronized-by-default				Explicit synchronization	
Benchmark	**LOC**	**Split**	**Custom**	**CanSplit**	**Final**	**Synchronized**	**Volatile**
LuIndex	5222	1		38	76	27	9
LuSearch	2452	4	2	2	46	9	4
PMD	7121	2	2	4	158	2	
Sunflow	3827	3		9	50	3	
H2	1235	1		39	14	1	
Tomcat	29314	15	11	50	333	140	6

Table 5: Number of benchmark modifications. The numbers apply to code executed by the benchmark.

mark executes 100 million read or write operations, on 100 million instances each having a single field. We report time without lock (row *Baseline*) and for lock operations with effects: Check new, check owned, or acq. & rls. (includes undo log). The table shows two different access patterns: pseudo-random and sequential. The results show that the overhead of the checks, both new and owned, is relatively small compared to the overhead of acquire & release, regardless of the access pattern.

The other reason for these overheads are the SBD semantics that allow large transactions sizes. All examined benchmarks allow using such large transactions: Table 5.2 shows the low amount of required split instructions compared to the size of executed LOC. Within such large transactions, lock acquires and releases are relatively infrequent compared to new or owned checks that have a low overhead.

To allow a better understanding of the sequential overhead, Table 7 shows the number of lock operations that the benchmarks execute per second, on average. They are subdivided into four columns based on the effect of the operation (init, check new, check owned, acq. & rls). Note that the lock operation counts and the execution times are from two different executions, and thus may not be completely comparable, as each execution may have invoked the JIT compiler in a different context.

The benchmarks that have an overhead around the geom. mean 35.4% (LuIndex 46.7%, LuSearch 29.9%, PMD 43.3%, Tomcat 24.4%) have a mix of these operations. Sunflow has a high overhead (99.3%), because it executes a large number of lock initializations as well as owned checks. Note that Sunflow does not perform any I/O. On the other hand, H2 experiences a low overhead (13.4%) because it spends most of its time doing I/O operations, i.e., data base accesses via JDBC. As databases use transactions we integrated the JDBC classes using transactional wrappers.

The additional final fields reduce the sequential overhead (geom. mean over all thread counts) for Sunflow by 19.4%, and for PMD by 1.69%. The other benchmarks do not show a measurable reduction.

5.4 Scalability

Figure 7 shows the speedup curves of all benchmarks with the exclusion of LuIndex, which uses a fixed number of threads (main and worker). The speedup curves of Sunflow, PMD and H2 are similar in both variants. For LuSearch and Tomcat, beginning with 32 threads the speedup of explicit synchronization is higher than of the SBD variant. For LuSe-

Effect	Read		Write	
	Random	Seq.	Random	Seq.
Baseline	17.6s	0.92s	32.8s	1.11s
New	17.8s	0.93s	32.9s	1.12s
	(+1.14%)	(+1.09%)	(+0.30%)	(+0.90%)
Owned	28.8s	1.90s	47.5s	2.37s
	(+63.6%)	(+107%)	(+44.8%)	(+114%)
Acq. & Rls.	62.9s	6.75s	68.8s	8.08s
	(+257%)	(+634%)	(+110%)	(+628%)

Table 6: Microbenchmark: 100 million read or write operations on a single core. Avg. over 20 iterations (STD $< 2\%$).

Benchmark	Init	Check New	Check Owned	Acq.
H2	42k	198k	1 036k	520k
LuIndex	46k	186 639k	35 928k	3k
LuSearch	516k	149 164k	18 327k	32k
PMD	940k	56 013k	10 065k	253k
Sunflow	1 596k	66 146k	111 156k	41k
Tomcat	30k	3 078k	7 430k	7 212k

Table 7: Locking operations per second (avg.).

Benchm.	Baseline	SBD			
		Locks	R-W set	Buffers	Init
H2	48 650k	18k	3k	0k	0k
LuIndex	2 664k	75k	13k	1 280k	94k
LuSearch	818k	538k	86k	25k	231k
Pmd	5 758k	831k	56k	171k	2 477k
Sunflow	2 202k	1 034k	159k	0k	3 361k
Tomcat	4 071k	1 146k	511k	5k	0k

Table 8: Memory overhead (avg., single threaded execution).

arch, the reason is the garbage collection (GC) that requires additional time to process the undo log. We consider this an issue of using a Java STM implementation. In a VM-based implementation, the GC could, e.g., avoid processing the undo log, as the lock structures already contain information whether the undo log references an instance. For Tomcat, the reason is that it uses 32 client and 32 server threads thus exceeding the limit of maximal 56 concurrently running threads (restriction of the STM implementation). Table 9 shows the transaction abort rate (abort/success, *Abr.*), the number of contented lock acquires (*Con.*), and the number of CAS-failures (*Fail.*). Synchronization issues are generally low. The exception is the abort rate for Sunflow, but it does not negatively affect the runtime of the benchmark.

5.5 Memory overhead

The memory overhead of the SBD approach consists of the additional memory used by the field level locks and the transaction logs. To measure the memory usage, a separate thread triggers a GC run every 50 ms. The thread samples the memory usage after each GC run. The reported numbers are the average of the samples interpolated linearly. All values are for 20 invocations of a single, sequential execution (STD $< 5\%$). The proof-of-concept implementation of the SBD approach does not allow to measure memory overhead directly. Instead, we report additional memory requirements of the data that is relevant for the SBD approach (largest contributors).

Benchm.	Thr.	Base. [s]	Sbd. [s]	Ovr. [%]	Abr. [%]	Con.	Fail.
H2	**1**	4.19	4.75	13.4	0.0	0	0
	2	4.15	4.42	6.6	0.0	7	7
	4	4.24	4.33	1.9	0.0	17	19
	8	5.25	5.31	1.3	0.0	12	13
	16	6.95	7.04	1.3	0.0	14	13
	32	10.12	10.17	0.4	0.0	10	12
LuIndex	**1**	0.92	1.35	46.7	0.0	0	0
LuSearch	**1**	5.68	7.37	29.9	0.0	0	0
	2	2.82	3.71	31.4	0.0	4	0
	4	1.44	1.86	29.3	0.0	11	1
	8	0.72	0.96	32.2	0.0	34	3
	16	0.38	0.5	31.8	0.0	70	5
	32	0.25	0.36	42.7	0.0	76	4
PMD	**1**	3.99	5.72	43.3	0.0	0	0
	2	2.08	2.84	36.2	0.0	4	6
	4	1.4	1.94	38.5	0.0	18	24
	8	1.36	1.84	35.4	0.0	45	69
	16	1.41	1.87	32.8	0.0	78	116
	32	1.5	1.94	29.1	0.0	119	147
Sunflow	**1**	15.11	30.1	99.3	0.0	0	0
	2	7.69	15.15	97.0	3.5	53	11
	4	3.94	7.45	88.8	44.4	209	30
	8	1.98	3.99	102	54.4	264	23
	16	1.17	2.33	98.8	59.9	355	30
	32	0.9	1.82	102	112	666	128
Tomcat	**1**	5.97	7.42	24.4	0.0	0	0
	2	3.04	3.75	23.3	0.0	298	317
	4	1.56	1.91	22.4	0.0	1002	1653
	8	0.82	1.01	24.1	0.1	1605	3030
	16	0.45	0.58	29.0	0.3	1260	2101
	32	0.34	0.51	50.2	0.4	1214	1906

Table 9: Overhead (*Ovr.*) of SBD approach compared to explicit locking (*Base.*). STD of execution times $< 5\%$, except for Tomcat/32/Base: 9.02%.

Table 8 shows the results of the measurement. The column *Baseline* lists the average memory usage (heap size after full GC) of the baseline variant. The column *Locks* shows the average additional memory for the lock structures of the SBD variant. Further, columns *R-W set*, *Buffers*, and column *Init* show the size of a transaction, split into the read-

write set (incl. old values), undo buffers (reads from streams, and writes to streams/files), and initialization log (list of instances to mark as `UNALLOC` on commit).

Due to the lazy lock structure allocation, additional memory usage of the field level locks is quite low, except for LuSearch (+66%) and Sunflow (+47%). The memory usage of the undo log is in general low, except for LuIndex, which writes a large file in a single transaction, and Tomcat, which has a large R-W set as a result of acquiring many write locks (also visible in Table 7). H2 is mostly using its database interface, thus there is almost no additional memory usage. Benchmarks with large transactions usually also have large initialization logs (Sunflow, PMD).

6. Discussion

Whether the performance overhead associated with the SBD approach is acceptable depends on the use case. We believe that developers that appreciate features like garbage collection (GC), or array index bounds checks, would also appreciate the SBD approach. All these techniques have additional performance overhead but provide in exchange additional safety. The analogy of TM and GC has been observed before [14].

The SBD approach has an impact on the usage of the language itself. However, other features have this as well, e.g., a language that integrates GC would not provide an operation to deallocate memory manually. We consider the transactional external side effects the biggest change of our approach, since it even affects sequential programs. While inevitable transactions remain an alternative, they reduce scalability as explained in Section 3.4.

One concern with the SBD approach is that it is unclear what locks a method acquires. For the examined programs, this was in general a non-issue: Most methods behave as expected, e.g., returning the list size locks the list size, and not arbitrary memory locations. If a method has an unexpected locking behavior, e.g., it uses a cache or an object pool, this can be documented in a similar way as documenting the thread-safety property when using explicit synchronization. Finally, conflicts can be detected dynamically. We implemented a small debug mode in our runtime system that logs the blocked threads, and deadlock situations. This information together with the fact that SBD allows a programmer to incrementally add concurrency allows to resolve these issues mechanically by looking through this log. This was especially helpful as we did not know the benchmark programs beforehand.

7. Related work

The SBD approach relates to various earlier efforts. We group them into three categories.

Synchronized-by-default. Kuszmaul et al. proposed atomicity by default using transactions in combination with language support, i.e., the use of HTM in combination with language extensions to the fork-join mechanism of Clik [25]. Hammond et al. proposed to use HTM and language extensions to loop and fork-join mechanisms. They performed scalability measurements using a simulated HTM [16]. Isard et al. proposed Automatic-Mutual-Exclusion (AME) [23], a system for concurrent asynchronous programming that allows the integration of functionality performing external side effects outside of transactions, and suggested the use of STM. Abadi et al. further investigated the semantics of the AME approach [1]. A proposal to implement atomicity by default without transactions is Coqa [26] by Liu et al., a language that uses message-passing ideas to demark atomic sections and to identify concurrency in a programs. Although not synchronizing by default, the TIC model [31] provides a way to optionally punctuate (split) a transaction, e.g., to perform I/O. The SBD approach shows novel ways to integrate the transactional version of the approach into an object-oriented, managed language based on threads. Further, it shows how to exploit properties of such a language for optimizations and compares a version of Java adapted for the SBD approach against regular Java with explicit synchronization, using several realistic application programs and existing hardware.

STM using readers-writer (RW) locks. Not much attention was paid to these STMs in the past, because they have been slower than STMs using a global clock [29]. Recent work provided examples of well performing STMs using RW locks. Dice et al. show that such an STM can outperform a TL2 global-clock based STM [10, 11] on single-chip multicore processors. Zhang et al. show that a STM using RW locks performs well for low-contention workloads [36], similar to those we used in our experiments. Our approach displays a usage for STMs using RW locks. Unlike previous work, our STM contains only the minimal required features to avoid overhead.

No critical sections. Other techniques that provide synchronization without critical sections are: *Data-centric* approaches such as [7, 35] that infer synchronization from annotations. These annotations group data into atomic units and mark control flow where the system must maintain atomicity. Thus, these approaches require explicit synchronization through annotations. *Effect systems* [5, 22] enable a programmer to annotate a sequence of statements with their effects that a scheduler statically or dynamically can use to execute the statements without conflicts. A further alternative is *isolation* [6, 27] to avoid direct shared memory access by working on copies of data within the context of fork-join parallelism. And finally, Brinch Hansen's Distributed Processes [17] achieve synchronization by forbidding shared memory accesses and by requiring processes (threads) to communicate if shared memory access is needed. All three approaches (effect systems, isolation, and forbidding shared memory accesses) avoid synchronization, but limit the number of expressible parallel programming idioms.

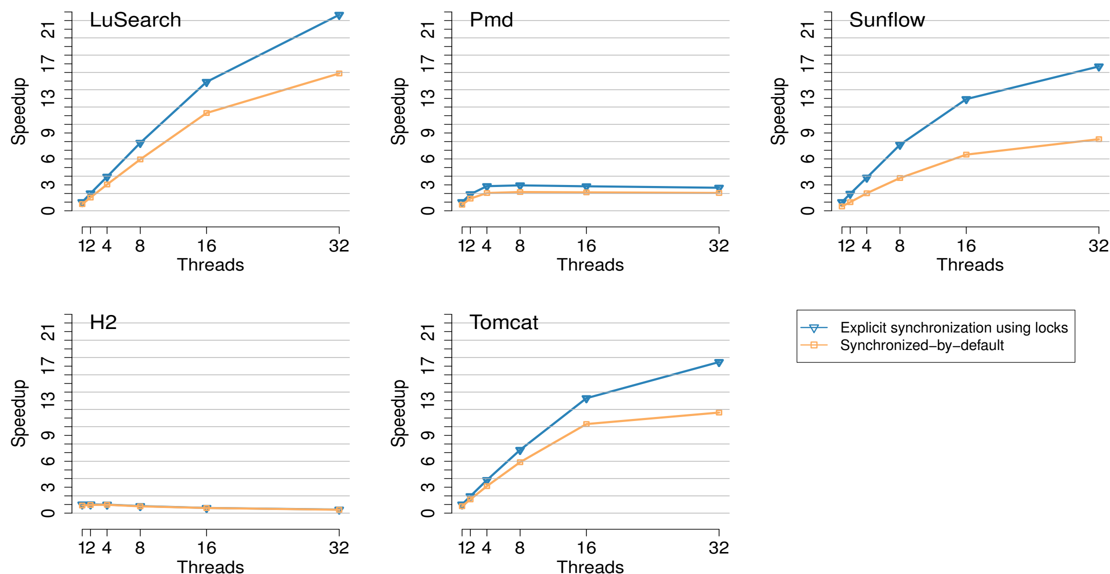

Figure 7: Scalability of the SBD approach vs. explicit synchronization. Base: Single threaded explicit synchronization.

8. Conclusion

The SBD approach for concurrency aims to reduce the occurrence of concurrency bugs due to data races, race conditions, and deadlocks. The approach allows the implementation of realistic applications that are scalable. We measured an overhead of 23.9% (geom. mean) compared to explicit synchronization using locks by executing realistic benchmarks on existing hardware.

The SBD approach requires only a small number of extensions to an existing language. Our prototype implementation is based on Java. There are numerous benefits that are obtained from integrating the SBD approach into an existing framework: it is easy to convert a program to SBD and the result remains similar the version with explicit synchronization using locks, and a comparison is meaningful. Thus, knowledge and experience gained by programmers using explicit synchronization is not lost. An additional benefit is that SBD reuses the existing threading constructs, transactional external wrappers for side effects, and the JIT compiler.

We have not yet studied to what extend SBD eases professional program development outside of an academic setting. However given the difficulty of constructing correct parallel programs, SBD is an attractive option that shows another approach to the development of concurrent programs. To the extend that the execution time of a concurrent program is not dominated by synchronization time, SBD offers an appealing alternative.

Acknowledgments

We thank the anonymous reviewers for their thorough reviews and helpful comments. We acknowledge computing resources provided by SNF grant 206021_133835.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of Transactional Memory and Automatic Mutual Exclusion. In *POPL '08*, pages 63–74, 2008.

[2] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06*, pages 169–190, 2006.

[3] C. Blundell, E. C. Lewis, and M. M. K. Martin. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters*, 5(2), 2006.

[4] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance Pathologies in Hardware Transactional Memory. In *ISCA '07*, pages 81–91, 2007.

[5] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA '09*, pages 97–116, 2009.

[6] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent Programming with Revisions and Isolation Types. In *OOPSLA '10*, pages 691–707, 2010.

[7] L. Ceze, C. von Praun, C. Caşcaval, P. Montesinos, and J. Torrellas. Concurrency Control with Data Coloring. In *MSPC*

'08, pages 6–10, 2008.

[8] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *PPoPP '10*, pages 67–78, 2010.

[9] T. David, R. Guerraoui, and V. Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *SOSP '13*, pages 33–48, 2013.

[10] D. Dice and N. Shavit. TLRW: Return of the Read-write Lock. In *SPAA '10*, pages 284–293, 2010.

[11] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *DISC '06*, pages 194–208, 2006.

[12] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *ASPLOS XIV*, pages 157–168, 2009.

[13] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. In *OOPSLA '07*, pages 57–76, 2007.

[14] D. Grossman. The Transactional Memory / Garbage Collection Analogy. In *OOPSLA '07*, pages 695–706, 2007.

[15] R. Guerraoui and M. Kapalka. On the Correctness of Transactional Memory. In *PPoPP '08*, pages 175–184, 2008.

[16] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with Transactional Coherence and Consistency (TCC). In *ASPLOS XI*, pages 1–13, 2004.

[17] P. B. Hansen. Distributed Processes: A Concurrent Programming Concept. *Commun. ACM*, 21(11):934–941, 1978.

[18] T. Harris. Exceptions and Side-effects in Atomic Blocks. *Sci. Comput. Program.*, 58(3):325–343, 2005.

[19] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *OOPSLA '03*, pages 388–402, 2003.

[20] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable Memory Transactions. In *PPoPP '05*, pages 48–60, 2005.

[21] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *ISCA '93*, pages 289–300, 1993.

[22] S. T. Heumann, V. S. Adve, and S. Wang. The Tasks with Effects Model for Safe Concurrency. In *PPoPP '13*, pages 239–250, 2013.

[23] M. Isard and A. Birrell. Automatic Mutual Exclusion. In *HOTOS '07*, pages 3:1–3:6, 2007.

[24] E. Koskinen and M. Herlihy. Dreadlocks: Efficient Deadlock Detection. In *SPAA '08*, pages 297–303, 2008.

[25] B. C. Kuszmaul, C. E. Leiserson, and S. Fellow. Transactions Everywhere. Technical report, 2003.

[26] Y. D. Liu, X. Lu, and S. F. Smith. Coqa: Concurrent Objects with Quantized Atomicity. In *CC '08/ETAPS '08*, pages 260–275, 2008.

[27] N. D. Matsakis. Parallel Closures: A New Twist on an Old Idea. In *HotPar '12*, pages 5–5, 2012.

[28] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *HPCA-12*, pages 254–265, 2006.

[29] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-core Runtime. In *PPoPP '06*, pages 187–197, 2006.

[30] N. Shavit and D. Touitou. Software Transactional Memory. In *PODC '95*, pages 204–213, 1995.

[31] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with Isolation and Cooperation. In *OOPSLA '07*, pages 191–210, 2007.

[32] M. F. Spear, V. J. Marathe, W. N. S. III, and M. L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. In *DISC '06*, pages 179–193, 2006.

[33] M. F. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott. Implementing and Exploiting Inevitability in Software Transactional Memory. In *ICPP 2008*, pages 59–66, 2008.

[34] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java Bytecode Optimization Framework. In *CASCON '99*, 1999.

[35] M. Vaziri, F. Tip, and J. Dolby. Associating Synchronization Constraints with Data in an Object-oriented Language. In *POPL '06*, pages 334–345, 2006.

[36] M. Zhang, J. Huang, M. Cao, and M. D. Bond. Low-overhead Software Transactional Memory with Progress Guarantees and Strong Semantics. In *PPoPP '15*, pages 97–108, 2015.

A. Artifact Evaluation

The evaluated artifact contained all the necessary code and data to reproduce the performance overhead measurements in Table 9, the locking operations per second in Table 7, and the speedup curves in Figure 7. The results of the microbenchmark in Table 6, and the memory overhead measurements in Table 8 were not part of the evaluation. The artifact is available from the authors upon request.