Lightweight Transactional Arrays for Read-Dominated Workloads*

Ivo Anjo and João Cachopo

ESW
INESC-ID Lisboa/Instituto Superior Técnico/Universidade Técnica de Lisboa
Rua Alves Redol 9, 1000-029 Lisboa, Portugal
{ivo.anjo,joao.cachopo}@ist.utl.pt

Abstract. Many common workloads rely on arrays as a basic data structure on top of which they build more complex behavior. Others use them because they are a natural representation for their problem domains.
Software Transactional Memory (STM) has been proposed as a new concurrency control mechanism that simplifies concurrent programming. Yet, most STM implementations have no special representation for arrays. This results, on many STMs, in inefficient internal representations, where much overhead is added while tracking each array element individually, and on other STMs in false-sharing conflicts, because writes to different elements on the same array result in a conflict.
In this work we propose new designs for array implementations that are integrated with the STM, allowing for improved performance and reduced memory usage for read-dominated workloads, and present the results of our implementation of the new designs on top of the JVSTM, a Java library STM.

Keywords: Parallel Programming, Software Transactional Memory

1 Introduction

Software Transactional Memory (STM) [10,15] is a concurrency control mechanism for multicore and multiprocessor shared-memory systems, aimed at simplifying concurrent application development. STM provides features such as atomicity and isolation for program code, while eliminating common pitfalls of concurrent programming such as deadlocks and data races. During a transaction, most STMs internally work by tracking the memory read and write operations done by the application on thread-local read and write-sets.

Tracking this metadata adds overheads to applications that depend on the granularity of transactional memory locations. There are two main STM designs regarding granularity: Either word-based [4,8] or object-based [7,11]. Word-based designs associate metadata with either each individual memory location, or by

* This work was supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds and by the RuLAM project (PTDC/EIA-EIA/108240/2008).

mapping them to a fixed-size table; whereas object-based designs store transactional information on each object or structure's header, and all of the object's fields share the same piece of transactional metadata.

Arrays, however, are not treated specially by STM implementations. Thus, programmers either use an array of transactional containers in each position, or they wrap the entire array with a transactional object. Neither option is ideal, if we consider that array elements may be randomly but infrequently changed.

Because arrays are one of the most elemental data structures on computing systems, if we hope to extend the usage of STM to provide synchronization and isolation to array-heavy applications, minimizing the imposed overhead is very important.

In this paper, we describe how existing transactional arrays are implemented, and explore new approaches that are integrated with the STM, achieving better performance and reducing memory usage for read-dominated workloads. Our work is based on the Java Versioned Software Transactional Memory (JVSTM) [2,3], a multi-version STM.

The rest of this work is organized as follows. Section 2 introduces the JVSTM transactional memory. Section 3 describes current black-box approaches to arrays. Section 4 introduces the new proposals for handling arrays. In Section 5, we compare the different array implementations. Experimental results are presented in Section 6, followed, in Section 7, by a survey of related work. Finally, in Section 8, we finish by presenting the conclusions and future research directions.

2 The JVSTM Software Transactional Memory

The Java Versioned Software Transactional Memory (JVSTM) is a pure Java library implementing an STM [3]. JVSTM introduces the concept of versioned boxes [2], which are transactional locations that may be read and written during transactions, much in the same way of other STMs, except that they keep the history of values written to them by any committed transaction.

Programmers using the JVSTM must use instances of the `VBox` class to represent the shared mutable variables of a program that they want to access transactionally. In Java, those variables are either class fields (static or not) or array components (each element of an array).

As an example, consider a field `f` of type `T` in a class `C` whose instances may be accessed concurrently. To access `f` transactionally, the programmer must do two things: (1) transform the field `f` in `C` into a final field that holds an instance of type `VBox<T>`, and (2) replace all the previous accesses to `f` by the corresponding operations on the contents of the box now contained in `f`.

JVSTM implements versioned boxes by keeping a linked-list of `VBoxBody` instances inside each `VBox`: Each `VBoxBody` contains both the version number of the transaction that committed it and the value written by that transaction. This list of `VBoxBody` instances is sorted in descending order of the version number, with the most recent at the head. The key idea of this design is that transactions

typically need to access the most recent version of a box, which is only one indirection-level away from the box object.

Yet, because the JVSTM keeps all the versions that may be needed by any of the active transactions, a transaction that got delayed for some reason can still access a version of the box that ensures that it will always perform consistent reads: The JVSTM satisfies the opacity correctness criteria [9]. In fact, a distinctive feature of the JVSTM is that read-only transactions are lock-free and never conflict with other transactions. They are also very lightweight, because there is no need to keep read-sets or write-sets: Each read of a transactional location consists only of traversing the linked-list to locate the correct `VBoxBody` from which the value is to be read. These two characteristics make the JVSTM especially suited for applications that have a high read/write transaction ratio.

Currently there are two versions of the JVSTM that differ on their commit algorithm. The original version of the JVSTM uses a lock-based commit algorithm, described below, whereas more recently Fernandes and Cachopo described a lock-free commit algorithm for the JVSTM [6]. Unless otherwise stated, the approaches described in this paper apply to both versions of the JVSTM.

To synchronize the commits of read-write transactions, the lock-based JVSTM uses a single global lock: Any thread executing a transaction must acquire this lock to commit its results, which means that all commits (of read-write transactions) execute in mutual exclusion. After the lock acquisition, the committing transaction validates its read-set and, if valid, writes-back its values to new `VBoxBody` instances, which are placed at the head of each `VBox`'s history of values.

To prevent unbounded growth of the memory used to store old values for boxes, the JVSTM implements a garbage collection algorithm, which works as follows: Each committing transaction creates a list with all the newly created instances of `VBoxBody` and stores this list on its descriptor. The transaction descriptors themselves also form a linked-list of transactions, with increasing version numbers. When the JVSTM detects that no transactions are running with version number older than some descriptor, it cleans the next field of each `VBoxBody` instance in the descriptor, allowing the Java GC to clean the old values.

3 Current Black-Box Array Implementations

In this section, we describe the two most common alternatives to implement transactional arrays with the JVSTM if we use only its provided API — that is, if we use the JVSTM as a black-box library.

3.1 Array of Versioned Boxes

The most direct and commonly used way of obtaining a transactional array with the JVSTM is the array of VBoxes. A graphical representation of the resulting structure is shown in Figure 1.

One of the shortcomings of this approach is the array initialization: All positions on the array need to be initialized with a `VBox` before they are used, typically as soon as the array is created and before it is published.

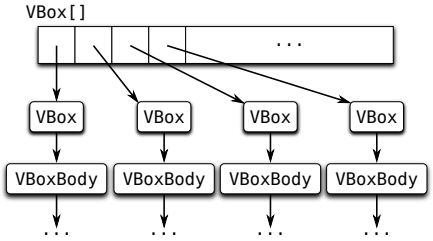

Fig. 1. Array of versioned boxes

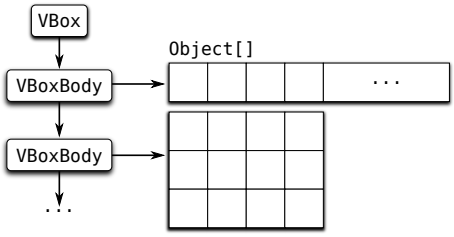

Fig. 2. Versioned box with array

Trying to perform lazy initialization highlights one of the issues of implementing such a data-structure outside the STM: the underlying native Java array is not under the control of the STM, and as such the programmer must provide his own synchronization mechanism for this operation. Side-stepping the synchronization provided by the STM while at the same time using the STM must be done carefully, or key STM characteristics might be lost, such as lock-freedom and atomicity, and common concurrent programming issues such as deadlocks might arise again. We will see in Section 4.1 a variant of this approach that uses lazy initialization and knowledge of the JVSTM's internals.

Since all VBoxes and their associated `VBoxBody` instances are normal Java objects, they still take up a considerable amount of memory when comparing to the amount needed to store each reference on the `VBox` array. As such, it is not unexpected for the application to spend more than twice the space needed for the native array to store these instances in memory.

3.2 Versioned Box with Array

The other simple implementation of a transactional array is one where a single `VBox` keeps the entire array, as shown in Figure 2.

Creation of this kind of array is straightforward, with overheads comparable to a normal non-transactional array. Array reads are the cheapest possible, only adding the cost of looking up the correct `VBoxBody` to read from; but writes are very expensive, as they need to duplicate the entire array just to change one of the positions. In addition, a single array write conflicts with every other (non read-only) transaction that is concurrently accessing the array, as the conflict detection granularity is the `VBox` holding the entire array.

Moreover, there is a very high overhead in keeping the history of values: For each version, an entire copy of the array is kept, even if only one element of the array was changed. This may lead the system to run out of memory very quickly, if writes to the array are frequent and some old running transaction prevents the garbage collector from running.

In conclusion, this approach is suited only for very specific workloads, with zero or almost-zero writes to the array. On the upside, for those workloads, it offers performance comparable to native arrays, while still benefiting from

```
Type value = getVBox(index).get(); // Reading from a VBoxArray
getVBox(index).put(newValue);      // Writing to a VBoxArray

VBox<Type> getVBox(int index) {    // Helper method getVBox
  VBox<Type> vbox = transArray[index];
  if (vbox == null) {
    vbox = new VBox<Type>((VBoxBody<Type>) null);
    vbox.commit(null, 0);
    if (!unsafe.compareAndSwapObject(transArray, ..., null, vbox))
      vbox = transArray[index];
  }
  return vbox;
}
```

Fig. 3. Code for the `VBoxArray` approach

transactional properties. It is also the only approach that allows the underlying array to change size and dimensions dynamically with no extra overhead.

4 New Array Proposals

In this section, we describe three proposals to implement transactional arrays that improve on the black-box approaches presented in the previous section.

4.1 VBoxArray and VBodyArray

The `VBoxArray` approach is obtained by adding lazy creation and initialization of VBoxes to the approach presented in Section 3.1. The main operations for this implementation are shown in Figure 3.

The `getVBox()` helper method first tries to obtain a `VBox` from the specified array position. If it exists, it is returned; otherwise a new one is created using an empty body, that is immediately written back, and tagged with version 0. This is conceptually the same as if the `VBox` was created by a transaction that ran before every other transaction and initialized all the boxes. The `VBox` is then put into the array in an atomic fashion: Either the `compareAndSwap`[1] operation succeeds, and the box is placed on the underlying array, or it fails, meaning that another thread already initialized it.

We can take the `VBoxArray` one step further and obtain the `VBodyArray` by doing away with the VBoxes altogether. The insight is that a `VBox` is needed only to uniquely identify a memory location on which we can transactionally read and write. If we provide our transactional array inside a wrapper `VBodyArray` class, we can use another method to identify uniquely a memory position: a pair <`VBodyArray`, index>. Using this pair, we no longer need the VBoxes, because the underlying array can directly contain the `VBoxBody` instances that would normally be kept inside them; initialization can still be done lazily.

[1] Available in the `sun.misc.Unsafe` class included in most JVM implementations.

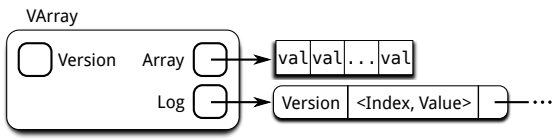

Fig. 4. The `VArray` transactional array

The `VBodyArray` saves a considerable amount of memory for larger arrays, and also lowers overhead on reads, as less memory reads need to be done to reach the values.

4.2 VArray

The `VArray` approach, shown in Figure 4, does away entirely with the normal storage mechanisms in the JVSTM: No VBoxes and no VBoxBodies are needed. It is designed to have the upsides of the VBox with Array approach described in Section 3.2, but to eliminate or minimize the downsides.[2]

The main design idea is to have an array that keeps both a set of values tagged with a version and a log containing the remaining versions.

Based on this design, two strategies are possible:

- The underlying array keeps the oldest values for each array position, and newer values are kept in the log; there must be a strategy to decide when to transfer values from the log to the main array.
- The underlying array keeps the latest values for each array position, and older values are kept in the log; there must be a strategy to allow garbage collection of older values from the log.

We argue that the second choice is more in line with the spirit of the JVSTM's design, because newer transactions find their values quickly, while older long-running transactions have to search through the log to find their older values. Additionally, JVSTM's existing garbage collection algorithm can, with minor modifications, be used to perform garbage collection of the log.

Reading from a VArray We start by reading the value directly from the array and then we check the array version: If it is older than the current transaction version number, we may return the value that we read directly from the array. If, instead, an older value is needed, we have to check the log to find the value corresponding to our current version, and return it, if found; otherwise, we may safely return the value originally read from the array because that position was never changed, although other array positions were. Figure 5 shows the code to read from a `VArray`.

[2] The full source-code for the JVSTM with the `VArray` class is available on the `jvstm-lock-free` branch at `http://groups.ist.utl.pt/esw-inesc-id/git/jvstm.git/`.

```
Type value = array.values.get(index); // Read value from array (volatile read!)
int version = array.version;          // Read array version

// If the array did not change, return the value read, otherwise check the log
if (version <= currentTransactionVersion) return value;
Type logValue = array.log.getLogValue(index, currentTransactionVersion);
return logValue != null ? logValue : value;
```

Fig. 5. Reading from a `VArray`

```
int txNumber;                       // Version of transaction being committed
VArrayEntry<Type>[] writesToCommit; // Sorted list of writes to be committed

// Create and initialize logEntryIndexes to be used in the log
int[] logEntryIndexes = new int[writesToCommit.length];
for (int i = 0; i < writesToCommit.length; i++)
  logEntryIndexes[i] = writesToCommit[i].index;
// Create and place log node
Type[] logEntryValues = (Type[]) new Object[writesToCommit.length];
array.log = new VArrayLogNode<Type>(logEntryIndexes, logEntryValues,
                                    txNumber - 1, array.log);
// Bump array version
array.version = txNumber;
// Writeback values
int i = 0;
for (VArrayEntry<Type> entry : writesToCommit) {
  // Read old value from the array, and copy it to the log
  logEntryValues[i++] = array.values.get(entry.index);
  // Write the new value
  array.values.lazySet(entry.index, entry.object); // Volatile write!
}
```

Fig. 6. Committing changes to a `VArray`

Writing and Committing to a VArray Writing to a `VArray` is similar to writing to a `VBox`: the value to be written is added to the transaction's write-set.

During the commit, the write-back to a `VArray` proceeds as follows:

1. Create a new log entry with the indexes of the array positions that are going to be overwritten and add that entry to the head of the log;
2. Update the array version;
3. Finally, backup to the log and write-back each changed array position. Each write operation is done with `volatile` semantics.

These steps need to be done while inside the commit lock. On the lock-free version of the JVSTM, because there is no commit lock, each array is locked individually; note that this approach eliminates the property of lock-freedom from the commit of transactions that wrote to a `VArray`, but lock-freedom is restored when no active transactions are writing to `VArray` instances. Figure 6 shows a simplified version of the `VArray` commit code.

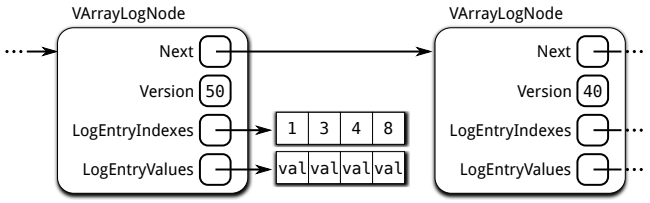

Fig. 7. Structure of a `VArrayLogNode`

Accessing the Log Figure 7 shows the structure of a `VArrayLogNode`. The `VArray` log is a linked list of VArrayLogNodes containing the older values of array positions that were overwritten by newer transactions. Inside each `VArrayLogNode`, a version field keeps the last version when the values contained in it were valid. This means that if the log contains two nodes, with versions 50 and 40, then the node with version 50 is valid for transactions with a version in the range $]40, 50]$ and the older entry is valid for transactions with versions $]0, 40]$.

Moreover, each `VArrayLogNode` maintains two arrays: `logEntryIndexes` and `logEntryValues`. The first keeps an ordered list of indexes that were changed by the transaction that created the log node, and the second keeps the values that were at those indexes, and were overwritten in the main array.

When a transaction with version n needs to look up the log for the value in index *index*, it first traverses the log nodes until it finds the log node with the smallest version $>= n$. It then checks that node for the index, by performing a binary search on the `logEntryIndexes` array. If this search finds the *index*, it returns the corresponding value. Otherwise, the search is resumed from the previous node, until a value is found, or the beginning of the log is reached — meaning that the requested value should be read from the main array.

Synchronization As we saw, the read algorithm first reads the value from the array, and then reads its version. To commit a new value we reverse this order: First the committer updates the version, and then writes back the new values.

Yet, without additional synchronization, we have a data race and the following can happen: The update of the array value may be reordered with the update of the version, which means that a reader may read the new value written by the committing transaction, but still read the old version value, causing the algorithm to return an invalid (newer) value to the application.

To solve this issue, and taking into account the Java memory model [12] we might be inclined to make the field that stores the array version `volatile`. Unfortunately, this will not work: If the committing thread first does a `volatile` write on the array version, and then updates the array, if the reading thread does not observe the write to the array version, no *synchronizes-with*[3] relation happens, and so the update to the array value may be freely reordered before

[3] The `volatile` keyword, when applied to a field states that if a thread *t1* writes to normal field *f1* and then to volatile field *f2*; then if other thread observes the write

Table 1. Comparison of array implementations. The memory overheads are considered for two workloads: a workload where only a single position is ever used after the array is created, and one where the entire array is used.

		Black-box Implementations		New Approaches		
		Array of VBoxes	**VBox with Array**	**VBoxArray**	**VBodyArray**	**VArray**
	Conflict Detection	Position	Entire Array	Position	Position	Position
Overheads Time	**Creation**	Very High	Very Low	Very Low	Very Low	Very Low
	Reading	Normal	Very Low	Normal	Low	Low
	Writing	Normal	Very High	Normal	Normal	High
Overheads Mem.	**History**	Normal	Very High	Normal	Normal	Low
	Single Position	N Box, N Body	1 Box, 1 Body	1 Box, 1 Body	1 Body	-
	Entire Array	N Box, N Body	1 Box, 1 Body	N Box, N Body	N Body	-

the version write, making a reader read the new value, and miss the new version. The other possible option would be for the committing thread to first write-back the value, and then update the array version with a `volatile` write; in this case, a simple delay or context switch between the two writes would cause issues.

As such, we can see that no ordering of writes to update both the array value and version can work correctly if just the version is declared `volatile`. As it turns out, the commit algorithm works correctly if only the array value is read and written with volatile semantics (through the usage of the `AtomicReferenceArray` class), and the version as a normal variable. This way, the reader can never read a newer value and an old version, because by `volatile` definition, if we observe a value, we at least observe the correct version for that value, but may also observe a later version, which poses no problem: In both cases the algorithm will correctly decide to check the log.

Garbage Collection We also extended the JVSTM garbage collection algorithm to work with the `VArray` log. As the linked list structure of the array log is similar to the linked list of bodies inside a `VBox`, new instances of `VArrayLogNode` that are created during transaction commit are also saved in the transaction descriptor, and from then the mechanism described in Section 2 is used.

5 Comparison of Approaches

Table 1 summarizes the key characteristics of the multiple approaches described in this paper. The single position memory overhead test case considers an array of n positions, where, after creation, only one of those positions is ever used during the entire program; conversely the entire array test case considers one where every position of the array is used. The memory overheads considered are in addition to a native array of size n, which all implementations use.

The main objective of this work was the creation of an array implementation that provided better performance for read-only operations, while minimizing memory usage and still supporting write operations without major overheads. We believe `VArray` fulfills those objectives, as it combines the advantages of the "VBox with Array" approach, such as having a very low memory footprint and

on *f2*, it is guaranteed that it will also see the write to *f1*, and also every other write done by *t1* before the write to *f2*. This is called a *synchronizes-with* [12] relationship.

read overhead, with advantages from other approaches, notably conflict detection done at the array position level, and low history overhead. Writes to a `VArray` are still more complex than most other approaches, but as we will see in Section 6 they can still be competitive.

6 Experimental Results

We shall now present experimental results of the current implementation of `VArray`. They were obtained on two machines: one with two Intel Xeon E5520 processors (8 cores total) and 32GB of RAM, and another with four AMD Opteron 6168 processors (48 cores total) and 128GB of RAM, both running Ubuntu 10.04.2 LTS 64-bit and Oracle Java 1.6.0_22. For our testing, we compared `VArray` to the Array of VBoxes approach, using the array benchmark,[4] which can simulate multiple array-heavy workloads. Before each test, the array was entirely initialized— note that after being fully initialized, the Array of Versioned Boxes and VBoxArray behave similarly. Each test was run multiple times, and the results presented are the average over all executions.

Figure 8 shows the scaling of `VArray` versus the Array of VBoxes approach for a read-only workload, with a varying number of threads. Each run consisted of timing the execution of 1 million transactions, with an array size of 1,000,000 on the 8-core machine, and 10,000,000 on the 48-core machine. Due to the reduced overheads imposed on array reads, `VArray` presents better performance.

Figure 9 shows the scaling of `VArray` versus a Array of VBoxes approach for a workload with a varying percentage of read-only and read-write transactions. Each read-only transaction reads 1000 (random) array positions, and each read-write transaction reads 1000 array positions and additionally writes to 10. Each run consisted of timing the execution of 100,000 transactions. As we can see, the increased write overhead of `VArray` eventually takes its toll and beyond a certain number of cores (that depend on the percentage of read-write transactions), `VArray` presents worse results than the Array of VBoxes approach. These results show that while `VArray` is better suited for read-only workloads, if needed it can still support a moderate read-write workload.

To test the memory overheads of `VArray`, we measured the minimum amount of memory needed to run a read-only workload in the array benchmark, on a single CPU, for an array with 10 million Integer objects. Due to its design, `VArray` was able to complete the benchmark using only 57MB of RAM, 10% of the 550MB needed by the Array of VBoxes approach.

Finally, we measured, using a workload comprised of 10% read-write transactions and 90% read-write transactions, and 4 threads, the minimum memory needed for both approaches to present acceptable performance, when compared with a benchmark run with a large heap. In this test, `VArray` took approximately 25% longer to execute with a 256MB heap, when compared to a 3GB heap; runs with an Array of VBoxes needed at least 800MB and also took 25% longer.

[4] `http://web.ist.utl.pt/sergio.fernandes/darcs/array/`

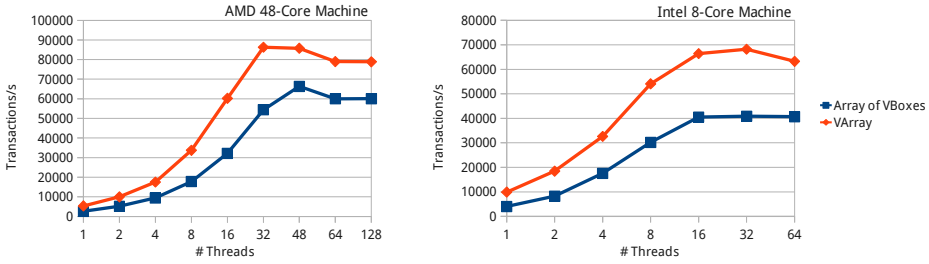

Fig. 8. Comparison of `VArray` versus the Array of VBoxes approach for the array benchmark, with a read-only workload on our two test systems.

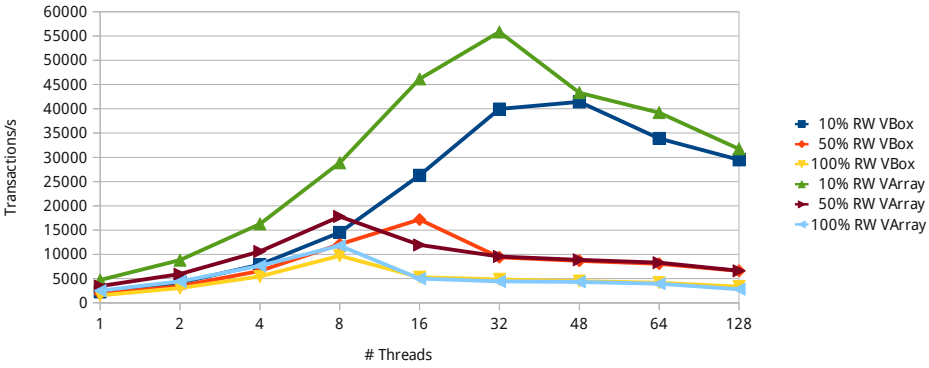

Fig. 9. Comparison of `VArray` versus the Array of VBoxes approach for the array benchmark, with varying number of read-write transactions (10%, 50% and 100%) on the 48-core AMD machine.

7 Related Work

Software Transactional Memory (STM) [15] is an optimistic approach to concurrency control on shared-memory systems. Many implementations have been proposed — Harris et al.'s book [10] provides a very good overview of the subject.

CCSTM [1] is a library-based STM for Scala based on SwissTM [5]. Similarly to the JVSTM, the programmer has to explicitly make use of a special type of reference, that mediates access to a STM-managed mutable value. Multiple memory locations can share the same STM metadata, enabling several levels of granularity for conflict detection. The CCSTM also provides a transactional array implementation that eliminates some of the indirections needed to access transactional metadata, similar to our `VBodyArray` approach.

The DSTM2 [11] STM framework allows the automatic creation of transactional versions of objects based on supplied interfaces. Fields on transactional objects are allowed to be either scalar or other transactional types, which disallows arrays; to work around this issue, the DSTM2 includes the `AtomicArray`

class that provides its own specific synchronization and recovery, but no further details on its implementation are given.

Another approach to reducing the memory footprint of STM metadata on arrays and other data structures is changing the granularity of conflict detection. Word-based STMs such as Fraser and Harris's WSTM [8] and TL2 in *per-stripe* mode [4] use a hash function to map memory addresses to a fixed-size transactional metadata table; hash collisions may result in false positives, but memory usage is bounded to the chosen table size.

Marathe et al. [13] compared word-based with object-based STMs, including the overheads added and memory usage; one of their conclusions is that the studied systems incur significant bookkeeping overhead for read-only transactions. Riegel and Brum [14] studied the impact of word-based versus object-based STMs for unmanaged environments, concluding that object-based STMs can reach better performance than purely word-based STMs.

Our `VArray` implementation is novel because it presents the same memory overheads of word-based schemes, while still detecting conflicts for each individual array position. Processing overhead for read-write transactions is still larger than with word-based approaches, because the transaction read-set must contain all individual array positions that were read, and all of them must be validated at commit-time, which is something word-based STMs can further reduce.

8 Conclusions and Future Work

Software transactional memory is a very promising approach to concurrency. Still, to expand into most application domains, many research and engineering issues need to be examined and solved. The usage of arrays is one such issue.

In this work we presented the first comprehensive analysis of transactional array designs, described how arrays are currently implemented on top of the JVSTM, and presented two implementations that improve on previous designs. In particular, the `VArray` implementation has memory usage comparable to native arrays, while preserving the lock-free property of JVSTM's read-only transactions. In addition, our experimental results show that `VArray` is highly performant for read-dominated workloads, and competitive for read-write workloads.

Future research directions include researching the possibility of a lock-free `VArray` commit algorithm, and exploring the usage of bloom filters for log lookups.

References

1. Bronson, N., Chafi, H., Olukotun, K.: CCSTM: A library-based STM for Scala
2. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Science of Computer Programming 63(2), 172–185 (2006)
3. Cachopo, J.: Development of Rich Domain Models with Atomic Actions. Ph.D. thesis, Technical University of Lisbon (2007)
4. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. Distributed Computing pp. 194–208 (2006)

5. Dragojević, A., Guerraoui, R., Kapalka, M.: Stretching transactional memory. In: ACM SIGPLAN Notices. vol. 44, pp. 155–165. ACM (2009)
6. Fernandes, S., Cachopo, J.: Lock-free and scalable multi-version software transactional memory. In: Proceedings of the 16th ACM symposium on Principles and practice of parallel programming. pp. 179–188. ACM (2011)
7. Fraser, K., Harris, T.: Practical lock-freedom. Tech. rep. (2004)
8. Fraser, K., Harris, T.: Concurrent programming without locks. ACM Trans. Comput. Syst. 25 (2007)
9. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. pp. 175–184. ACM, New York, NY, USA (2008)
10. Harris, T., Larus, J., Rajwar, R.: Transactional memory. Synthesis Lectures on Computer Architecture 5(1), 1–263 (2010)
11. Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. ACM SIGPLAN Notices 41(10), 253–262 (2006)
12. Manson, J., Pugh, W., Adve, S.: The Java Memory Model
13. Marathe, V.J., Scherer, W.N., Scott, M.L.: Design tradeoffs in modern software transactional memory systems. In: Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems. pp. 1–7. LCR '04, ACM, New York, NY, USA (2004)
14. Riegel, T., Brum, D.B.D.: Making object-based STM practical in unmanaged environments. In: TRANSACT '08: 3rd Workshop on Transactional Computing (2008)
15. Shavit, N., Touitou, D.: Software transactional memory. Distributed Computing 10(2), 99–116 (1997)