Efficient Support for In-Place Metadata in Transactional Memory

Ricardo J. Dias, Tiago M. Vale, and João M. Lourenço*

Departamento de Informática and CITI
Universidade Nova de Lisboa, Portugal
{ricardo.dias,t.vale}@campus.fct.unl.pt
joao.lourenco@fct.unl.pt

Abstract. Implementations of Software Transactional Memory (STM) algorithms associate metadata with the memory locations accessed during a transaction's lifetime. This metadata may be stored either in-place, by wrapping every memory cell in a container that includes the memory cell itself and the corresponding metadata; or out-place (also called external), by resorting to a mapping function that associates the memory cell address with an external table entry containing the corresponding metadata. The implementation techniques for these two approaches are very different and each STM framework is usually biased towards one of them, only allowing the efficient implementation of STM algorithms following that approach, hence inhibiting the fair comparison with STM algorithms falling into the other. In this paper we introduce a technique to implement in-place metadata that does not wrap memory cells, thus overcoming the bias by allowing STM algorithms to directly access the transactional metadata. The proposed technique is available as an extension to the DeuceSTM framework, and enables the efficient implementation of a wide range of STM algorithms and their fair (unbiased) comparison in a common STM infrastructure. We illustrate the benefits of our approach by analyzing its impact in two popular TM algorithms with two different transactional workloads, TL2 and multi-versioning, with bias to out-place and in-place respectively.

1 Introduction

Software Transactional Memory (STM) algorithms differ in the used read strategies (visible or invisible), update strategies (direct or deferred), conflict resolution policies (contention management), progress guarantees (blocking or non-blocking), consistency guarantees (opacity or snapshot isolation), and interaction with non-transactional code (weak or strong isolation), among others. Some STM

* This research was partially supported by the EU COST Action IC1001 (Euro-TM), the Portuguese national research projects RepComp (PTDC/EIA-EIA/108963/2008), Synergy-VM (PTDC/EIA-EIA/113613/2009), and the research grant SFRH/BD/41765/2007.

frameworks (e.g., DSTM2 [7] and DeuceSTM [8]) aim at allowing the implementation and comparison of different STM algorithms using a unique transactional interface, and are frequently used for experimenting with new algorithms.

STM algorithms manage information per transaction (frequently referred to as a *transaction descriptor*), and per memory location (or object reference) accessed within that transaction. The transaction descriptor is typically stored in a thread-local memory space and maintains the information required to validate and commit the transaction. The per memory location information depends on the nature of the STM algorithm, which we will henceforth refer to as *metadata*, and may be composed by e.g. locks, timestamps or version lists. Metadata is stored either "near" each memory location (*in-place* strategy), or in an external mapping table that associates the metadata with the corresponding memory location (*out-place* or *external* strategy).

STM libraries targeting imperative languages, such as C, frequently use an out-place strategy, while those targeting object-oriented languages bias towards the in-place strategy. The out-place strategy is implemented by using a table-like data-structure that efficiently maps memory references to its metadata. Storing the metadata in a pre-allocated table avoids the overhead of dynamic memory allocation, but incurs in overhead for evaluating the location-metadata mapping function and has limitations imposed by the size of the table. The in-place strategy is usually implemented by using the *decorator* design pattern [6] that is used to extend the functionality of an original class by wrapping it in a *decorator* class, which also contains the required metadata. This technique allows the direct access to the object metadata without significant overhead, but is very intrusive to the application code, which must be rewritten to use the decorator classes. This *decorator* pattern based technique also incurs in two other problems: some additional overhead for non-transactional code, and multiple difficulties to cope with primitive and array types. Riegel et al. [10] briefly describe the tradeoffs of using in-place *versus* out-place strategies.

DeuceSTM is among the most efficient STM frameworks for the Java programming language and provides a well defined interface that is used to implement several STM algorithms. On the application developer's side, a memory transaction is defined by adding the annotation `@Atomic` to a Java method, and the framework automatically instruments the application's bytecode by injecting callbacks to the STM algorithm, intercepting the read and write memory accesses. The injected callbacks provide the referenced memory address as argument, limiting the range of viable STM algorithms to be used by forcing an out-place strategy.

This paper describes the adaptation and extension of DeuceSTM to support the in-place metadata strategy without making use of the decorator pattern. Our new approach complies to the following properties:

Efficiency Our extension does not rely on an auxiliary mapping table, thus providing fast direct access to the transactional metadata; transactional code avoids the extra memory dereference imposed by the decorator pattern; no performance overhead is introduced for non-transactional code, as

it is oblivious to the presence of metadata in objects; primitive types are fully supported, even in transactional code; and we propose a solution for supporting transactional N-dimensional arrays with a negligible overhead for non-transactional code.

Flexibility Our extension supports both the original out-place and the new in-place strategies simultaneously, hence imposing no restrictions on the nature of the algorithms and their implementations.

Transparency Our extension automatically identifies, creates and initializes all the necessary additional metadata fields in objects; non-transactional code is oblivious to the presence of metadata in objects, hence no source code changes are required, although it does some light transformation on the non-transactional bytecode; the new transactional array types (that support metadata for individual cells) are compatible with the standard arrays, hence not requiring pre-/post-processing of the arrays when invoking standard or third-party non-transactional libraries.

Compatibility Our extension is fully backwards compatible and the already existing implementations of STM algorithms are executed with no changes and with null or negligible performance overhead.

In the remainder of this paper, we describe the DeuceSTM framework and the usage of out-place strategy in §2. In §3 we describe the properties of in-place strategy, and its implementation as an extension to DeuceSTM. We evaluate our implementation with some benchmarks in §4, and discuss the related work in §5. We finish with some concluding remarks in §6.

2 DeuceSTM and the Out-Place Strategy

Algorithms such as TL2 [4] or LSA [11] use an out-place strategy by resorting to a very fast hashing function and storing a single lock in each table entry. However, due to performance issues, the mapping table does not avoid hash collisions and thus two memory locations may be mapped to the same table entry, resulting in the false sharing of a lock for two different memory locations.

The out-place strategy fits well to algorithms whose metadata information does not depend on the memory locations, such as locks and timestamps, but is unfitting for algorithms that need to store location-dependent metadata information, e.g., multi-version based algorithms. The out-place implementations for these algorithms require a mapping table with collision lists, which impose a significant and unacceptable performance overhead.

DeuceSTM provides the STM algorithms with a unique identifier for an object field, composed by a reference to the object and the field's logical offset within that object. This unique identifier can then be used by the STM algorithms as a key to any map implementation that associate the object fields with the transactional metadata. Likewise for array types, the unique identifier of an array's cell is composed by the array reference and the index of that cell. It is worthwhile to mention that DeuceSTM relies heavily on bytecode instrumentation to provide a transparent transactional interface to application developers,

which are not aware of how the STM algorithms are implemented nor of the strategy being used to store the transactional metadata.

DeuceSTM is an extensible STM framework that may be used to compare different STM algorithm implementations. However, it is not fair to compare an algorithm that fits very well to the out-place strategy with another algorithm that does not. In the concrete case of DeuceSTM, the framework only supports an out-place strategy, therefore being inappropriate for e.g. multi-version oriented STM algorithms. We have extended DeuceSTM to, in addition to the out-place strategy, also support an efficient in-place strategy, while keeping the same transparent transactional interface to the applications.

3 Support for In-Place Strategy

The unique identifier of an object's field is composed by the object reference and the field's logical offset. DeuceSTM computes that logical offset at compile time, and for every field f in every class C an extra static field f^o is added to that class, whose value represents the logical offset of f in class C. No extra fields are added for array cells, as the logical offset of each cell corresponds to its index. When there is a read or write memory access (within a memory transaction) to a field f of an object O, or to the array element $A[i]$, the run-time passes the pair (O, f^o) or (A, i) respectively as the argument to the callback function. The STM algorithm shall not differentiate between field and array accesses. In DeuceSTM, if the algorithm needs to e.g. associate a lock with a field, it has to store the lock in an external table indexed by the hash value of the pair (O, f^o).

In our approach for extending DeuceSTM to support an in-place strategy, we replace the previous pair of arguments to callback functions (O, f^o) with a new metadata object f^m, whose class is specified by the STM algorithm's programmer. We guarantee that there is a unique metadata object f^m for each field f of each object O, and hence the use of f^m to identify an object's field is equivalent to the pair (O, f^o). The same applies to arrays where we ensure that there is a unique metadata object a^m for each position of an array A.

3.1 Implementation

Although the implementation of the support for in-place metadata objects differs considerably for class fields and array elements, a common interface is used to interact with the STM algorithm implementation. This common interface is supported by a well defined hierarchy of metadata classes, illustrated in Figure 1, where the rounded rectangle classes are defined by the STM algorithm developer.

All metadata classes associated with class fields extend directly from the top class `TxField`. For array elements, we created specialized metadata classes for each primitive type in Java, the `TxArr*Field` classes, where `*` ranges over the Java primitive types[1]. All the `TxArr*Field` classes extend from `TxField`, pro-

[1] **`int`**, **`long`**, **`float`**, **`double`**, **`short`**, **`char`**, **`byte`**, **`boolean`**, and `Object`.

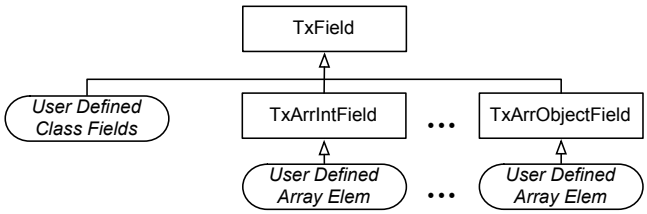

Fig. 1. Metadata classes hierarchy.

viding the STM algorithm with a simple and uniform interface for callback functions, which shall be extended to include the support of new STM algorithms. The newly defined metadata classes need to be registered in our framework to enable its use by the instrumentation process, using a Java annotation in the class that implements the STM algorithm, as exemplified in Listing 1.1.

Listing 1.1. Declaration of the STM algorithm specific metadata.

```
@InPlaceMetadata(
  fieldObjectClass="TL2Field",
  fieldIntClass="TL2Field",
  ...
  arrayObjectClass="TL2ArrObjectField",
  arrayIntClass="TL2ArrIntField",
  ...
)
final public class TL2Context implements ContextMetadata {
  ...
}
```

The STM algorithm must implement a well defined interface that includes a callback function for the read and write operations on each Java type. These functions always receive an instance of the super class `TxField`, but each one knows precisely which metadata subclass was actually used to instantiate the metadata object.

Lets now see where and how the metadata objects are stored, and how they are used on invocation of the callback functions. We will explain separately the management of metadata objects for class fields and for array elements.

Class Fields During the execution of a transaction, there must be a metadata object f^m for each accessed field f of object O. A very efficient way to implement this metadata object f^m is by making it accessible by a single dereference operation from object O. Therefore, for each declared field in a class C, we add an additional metadata field of the appropriate type. The general rule can be described as: given a class C that has a set of declared fields $\{f_1, \ldots, f_n\}$, we

add a metadata object field for each of the initial fields, such that the class ends with the set of fields $\{f_1, \ldots, f_n, f^m_{1+n}, \ldots, f^m_{n+n}\}$ where the field f_k is associated with the metadata field f^m_{k+n} for any $k \leq n$. In Listings 1.2 and 1.3 we show a concrete example of the transformation of a class with two fields.

Listing 1.2. The original class.

```
class C {
  int a;
  Object b;
}
```

$\Longrightarrow$

Listing 1.3. The transformed class.

```
class C {
  int a;
  Object b;
  TxField a_metadata;
  TxField b_metadata;
}
```

Each metadata field is instantiated at the constructor of the class where the field is declared. This ensures that whenever a new instance of a class is created, the corresponding metadata objects are also new and unique.

Opposed to the approach based in the *decorator* pattern, where primitive types must be replaced with their object equivalents (e.g., an **`int`** field is replaced by an `Integer` object), our transformation approach keeps the primitive type fields untouched, simplifying the interaction with non-transactional code, limiting the code instrumentation and avoiding autoboxing and its overhead.

Array Elements The structure of an array is very strict, with each array cell containing a single value of a well defined type, and no other information can be added to those elements. The common approach to overcome this limitation is to change the array to an array of objects that wrap the original value and the additional information. This transformation has strong implications in the remaining of the application code, as code statements expecting the original array type or array element will now have to be rewritten to receive the new array type or wrapping class respectively. This problem is even more complex if the arrays with wrapped elements were to be manipulated by non-instrumented libraries, such as the JDK libraries.

The solution we propose is also based on changing the type of the array to be manipulated by the instrumented application code, but strongly limiting the implications for the remaining non-instrumented code. We keep all the values in the original array, and have a sibling second array, only manipulated by the instrumented code, that contains the additional information and references to the original array. The type of the declaration of the base array is changed to the type of the corresponding sibling array (`TxArr*Field`), as shown in Figure 2. This Figure also illustrates the general structure of the sibling `TxArr*Field` arrays (in this case, a `TxArrIntField` array). Each cell of the sibling array has the metadata information required by the STM algorithm, its own position/index in the array, and a reference to the original array where the data is stored (i.e., where the reads and updates take place). This scheme allows the sibling array to keep a metadata object for each element of the original array, while maintaining

the original array always updated and compatible with non-transactional legacy code.

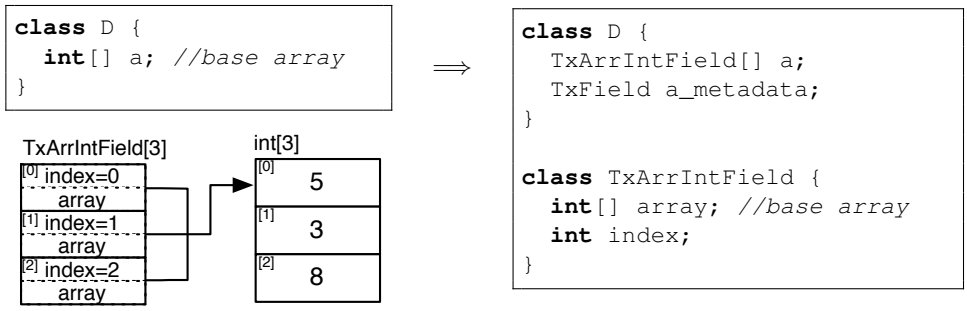

Fig. 2. Memory structure of a `TxArrIntField` array.

Non-transactional methods that have arrays as parameters are also instrumented to replace the array type by the corresponding sibling `TxArr*Field`. The value of an array element is then obtained by dereferencing the pointer to the original array kept in the sibling, as illustrated in Listings 1.4 and 1.5. When passing an array as argument to an uninstrumented method (e.g., from the JDK library), we can just pass the original array instance. Although the instrumentation of non-transactional code adds a dereference operation when accessing an array, we do avoid the autoboxing of primitive types that would impose an increased overhead.

Listing 1.4. Access to an array cell.

```
void foo(int[] a) {
  // ...
  t = a[i];
}
```

Listing 1.5. Access to an array cell from the transformed array.

```
void foo(TxArrIntField[] a) {
  // ...
  t = a[0].array[i];
}
```

Multi-dimensional arrays The special case of multi-dimensional arrays is tackled using the `TxArrObjectField` class, which has a different implementation from the other specialized metadata array classes. This class has the additional field `nextDim`, which may be null in the case of a uni-dimensional reference type array, or may hold the reference of the next array dimension by pointing to another array of type `TxArr*Field`. Once again, the original multi-dimensional array is always up to date and can be safely used by non-transactional code.

Figure 3 depicts the memory structure of a bi-dimensional array of integers. Each element of the first dimension of the sibling array has a reference to the original integer matrix. The elements of the second dimension of the sibling array have a reference to the second dimension of the matrix array.

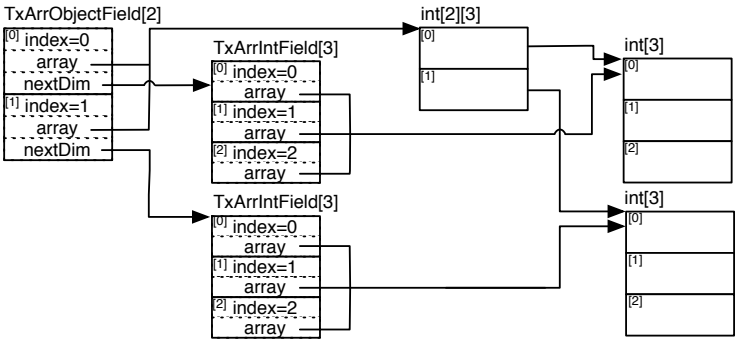

Fig. 3. Memory structure of a multi-dimensional `TxArrIntField` array.

4 Performance Evaluation

We evaluated our approach in two dimensions: the performance overhead resulting from the introduction of metadata associated with object fields, and the performance improvements achieved by implementing a multi-versioning STM algorithm (JVSTM [3]) using our extension (with in-place metadata), when compared to an equivalent implementation in the original DeuceSTM (with out-place metadata). To measure the transactional throughput we used the vanilla micro-benchmarks available in the DeuceSTM framework. No changes were necessary to execute the benchmarks on our extension of DeuceSTM with in-place metadata, as all the necessary bytecode transformations were performed automatically.

The benchmarks were executed in a computer with four AMD Opteron 6168 12-Core processors @ 1.9 GHz with 12x512 KB of L2 cache and 128 GB of RAM, running Red Hat Enterprise Linux Server Release 6.2 with Linux 2.6.32 x86_64.

To evaluate the overhead of our extension, we compared the performance of the TL2 algorithm as provided by the original DeuceSTM distribution, with another implementation of TL2 using the new interface of our modified DeuceSTM. The original DeuceSTM interface for callback functions provide a pair with the object reference and the field logical offset. The new interface provides a reference to the field metadata (`TxField`) object. Despite using the in-place metadata feature, the new implementation of TL2 resembles the original one as much as possible and still uses an external table to map memory references to locks. By comparing these two similar implementations, we can measure the overhead introduced by the management of the metadata object fields and sibling arrays.

Figure 4 depicts the overhead of our extension with respect to the original DeuceSTM for two data structures: a Red-Black Tree and a Skip List. The former only uses metadata objects for class fields, while the latter also make use of metadata arrays. We executed each data structure with two different workloads: a *read-only* workload, and a *read-write* workload with an average of 10% of update operations. The overhead is in percentage and is relative to the out-place implementation of TL2 in the original DeuceSTM.

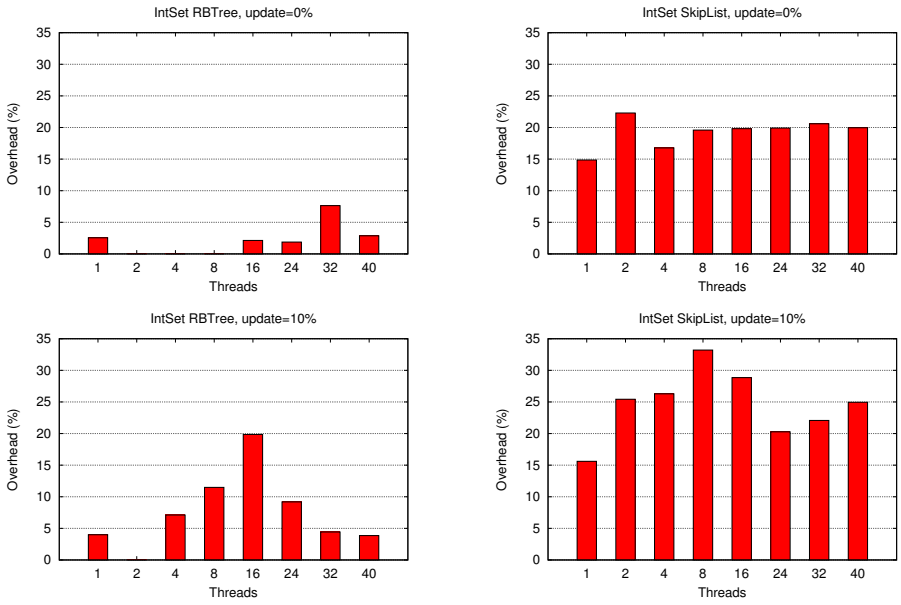

Fig. 4. Overhead measure of the usage of metadata objects relative to out-place TL2.

In the Red-Black tree benchmark, the use of metadata objects in class fields in a read-only workload (top-left chart) has a negligible overhead. In a read-write workload (bottom-left chart) there is an average overhead of 10% with respect to the out-place version. This overhead results from the additional allocations necessary to initialize metadata objects. For instance, when adding a new node to the tree, we need to allocate additional metadata objects for the value, color, left, right and parent fields. In the case of the Skip List benchmark, each node contains an array of nodes. In the read-only workload (top-right chart), there is an average overhead of 20% with respect to the out-place strategy that uses the original arrays declared in the program. Although no new nodes are allocated, there is a performance penalty to pay for the additional dereference imposed by the support of in-place metadata for arrays. In the read-write workload (bottom-right chart), which allocates new nodes, we get a slightly higher overhead averaging 25%.

From this analysis we conclude that our in-place strategy is a viable option for implementing algorithms biased to in-place transactional metadata. To stress this fact, we implemented two versions of the JVSTM algorithm as proposed in [3], one in the original DeuceSTM framework using the out-place strategy (JVSTM-Out), and another in the extended DeuceSTM using the in-place strategy (JVSTM-In). The JVSTM-Out implementation uses an open concurrent hash table to map each accessed memory location to its list of versions. Hence, for each memory location accessed, we perform a search in the hash table to find the respective version list. If none is found, a new one is created and

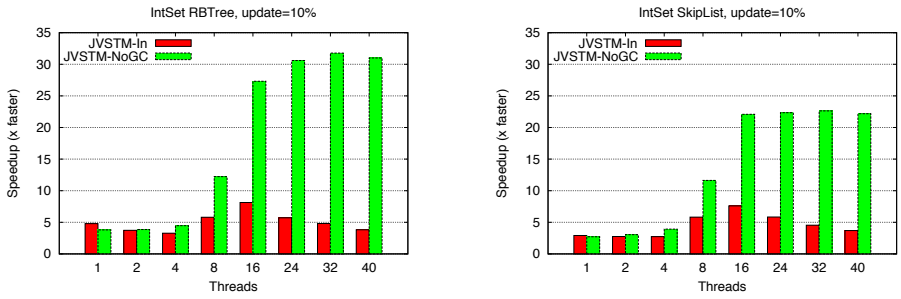

Fig. 5. Speedup of JVSTM-In and JVSTM-NoGC relative to JVSTM-Out

added to the hash map. The JVSTM-In implementation uses metadata objects containing the version lists, thus when there is an access to an object field or an array element, a direct reference to its version list is obtained from the corresponding metadata field. JVSTM-In algorithm follows the specification in [3], and although it exhibits a much better performance than the out-place version (in average it is 5× faster), it has some scalability problems due to: i) the global lock used in the commit phase, and ii) the garbage collection mechanism for *vboxes* (used by JVSTM to wrap the object and its list of versions). Hence we implemented an optimized variant of the JVSTM-In (JVSTM-NoGC), which replaces the global lock in the commit phase with separate locks for each memory locations in the transaction write-set, and which eliminates the *vbox* garbage collection mechanism by imposing an upper bound on the size for the list of versions for any memory location. In this new algorithm, transactions accessing an old version that is not available anymore are aborted and restarted.

Figure 5 depicts the speedup results when comparing the JVSTM-In and JVSTM-NoGC implementations with respect to the JVSTM-Out implementation. JVSTM-In is in average 5× faster than JVSTM-Out on both benchmarks. JVSTM-NoGC is in average 33× faster in the Red-Black Tree, and 23× faster in the Skip List. These results prove that our strategy to support in-place metadata in DeuceSTM gave it leverage to implement algorithms that need direct access to transactional metadata, thus enabling the fair comparison of a wide range of STM algorithms, including those that could not be implemented efficiently in the original DeuceSTM.

5 Related Work

Several STM algorithms were developed in the last few years, and comparing their performance always requires a great implementation effort while using the same transactional interface and programming language. Some STM frameworks address this problem and provide a uniform transactional interface front-end and a flexible run-time back-end, which is normally biased towards one of the in-place or out-place strategies.

DSTM2 [7] is a flexible STM framework for the Java language which permits the use of different synchronization techniques and recovery strategies as framework plug-ins. DSTM2 creates transactional objects using the factory pattern, and new factories can be implemented to test different properties of STM algorithms. DSTM2 only allows to implement algorithms using the in-place strategy.

DeuceSTM [8], which is the base of our work, is one of the most efficient STM frameworks available for the Java programming language. It provides a well defined interface that allows to implement several STM algorithms, and relies in Java bytecode instrumentation to intercept transaction limits and transactional memory accesses and invoke developer-defined callback functions. DeuceSTM has a strong bias towards the out-place approach.

STM algorithms such as TL2 [4], LSA [11] and SwissTM [5] are usually implemented using an out-place strategy, thus viable for use in DeuceSTM. Others such as JVSTM [3] and SMV [9] are better fit for the in-place strategy and impracticable for DeuceSTM. Our extension of DeuceSTM overcomes this limitation and allows the efficient implementation of algorithms using any of those strategies, enabling their fair comparison.

Anjo et al. [2] developed a specialized transactional array targeting specifically the JVSTM framework, achieving considerable performance improvements in read-dominant workloads that use arrays. Our approach when extending DeuceSTM aimed at providing an efficient implementation for transactional arrays that is backwards compatible, where no autoboxing is required and whose values are kept in their original primitive format and are accessible to both transactional and non-transactional code.

All the static optimizations proposed by Afek et al. [1] are orthogonal to our work and can also be applied to algorithms using the new in-place strategy, thus increasing the overall performance.

6 Concluding Remarks

To the best of our knowledge, the extension of DeuceSTM as described in this paper creates the first Java STM framework providing a balanced support of both in-place and out-place strategies. This is achieved by a transformation process of the program bytecode that adds new metadata objects for each class field, and that includes a customized solution for N-dimensional arrays that is fully backwards compatible with primitive type arrays. The creation or structural modification of arrays are not supported outside instrumented code, which is oblivious to `TxArr*Field` and metadata.

In the current state of the proposed extension every field is subjected to this transformation, hence there will be a considerable increase in the application's memory footprint. For example, for the Red-Black Tree benchmark with 50 000 elements in a read-only workload, the GNU `time` command reported 196 MB of memory used when using the out-place version of the TL2 algorithm, and 248 MB when using the in-place version. This memory overhead can be minimized by doing code analysis to discover the fields that are not accessed within

transactions, and skipping the creation and initialization of the metadata associated with those fields, which will never be needed.

We evaluated our system by measuring the overhead introduced by our new in-place interface with respect to the TL2 algorithm implementation provided in DeuceSTM distribution package as reference. Although we can observe a light slowdown in our new implementation of arrays, we would like to reinforce that our solution has no limitations whatsoever concerning the type of the array elements, the number of dimensions, fits equally algorithms biased towards in-place or out-place strategies, and all bytecode transformations are done automatically requiring no changes to the source code. We also evaluated the effectiveness of the new in-place interface by comparing the performance of two multi-version STM implementations: one using the newly proposed in-place strategy, and another using an out-place strategy resorting to an external mapping table. The version using the new in-place strategy was in average 5× faster than the one using the out-place strategy. The optimized version of the JVSTM algorithm using the in-place strategy was in average 33× faster than the out-place version.

References

1. Afek, Y., Korland, G., Zilberstein, A.: Lowering STM overhead with static analysis. In: Proc. 23rd Int. Workshop on Languages and Compilers for Parallel Computing (Oct 2010)
2. Anjo, I., Cachopo, J.: Lightweight transactional arrays for read-dominated workloads. In: Proc. 11th Int. Conf. on Algorithms and Architectures for Parallel Processing. pp. 1–13. Springer-Verlag, Berlin, Heidelberg (2011)
3. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Sci. Comput. Program. 63, 172–185 (Dec 2006)
4. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Proc. 20th Int. Symp. on Distributed Computing. LNCS, vol. 4167, pp. 194–208. Springer (Sep 2006)
5. Dragojević, A., Guerraoui, R., Kapalka, M.: Stretching transactional memory. In: Proc. Int. Conf. on Programming Language Design and Implementation. pp. 155–165. ACM (2009)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (1994)
7. Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. In: Proc. 21st conference on Object-Oriented Programming Systems, Languages, and Applications. pp. 253–262. ACM (2006)
8. Korland, G., Shavit, N., Felber, P.: Noninvasive concurrency with Java STM. In: Proc. MultiProg 2010: Programmability Issues for Heterogeneous Multicores (2010)
9. Perelman, D., Byshevsky, A., Litmanovich, O., Keidar, I.: SMV: Selective multiversioning STM. In: Proc. 25th Int. Symp. on Distributed Computing. LNCS, vol. 6950, pp. 125–140. Springer (2011)
10. Riegel, T., Brum, D.B.D.: Making object-based STM practical in unmanaged environments. In: Proc. of the 3rd Workshop on Transactional Computing (2008)
11. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: Proc. 20th Int. Symp. on Distributed Computing. LNCS, vol. 4167, pp. 284–298. Springer (Sep 2006)