CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2013; **00**:1–18
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Efficient support for in-place metadata in Java software transactional memory[†]

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

CITI and Departamento de Informática, FCT—Universidade Nova de Lisboa, Portugal

SUMMARY

Software Transactional Memory (STM) algorithms associate metadata with the memory locations accessed during a transaction's lifetime. This metadata may be stored in an external table, by resorting to a mapping function that associates the address of a memory cell with the table entry containing the corresponding metadata (out-place or external strategy). Alternatively, the metadata may be stored adjacent to the associated memory cell by wrapping the cell and metadata together (in-place strategy). The implementation techniques to support 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 which suit one of the approaches, and inhibiting a fair comparison with STM algorithms suiting the other. In this paper we introduce a technique to implement in-place metadata that does not wrap memory cells, thus overcoming the bias and allowing STM algorithms to directly access the transactional metadata. The proposed technique is available as an extension to Deuce, and enables the efficient implementation of a wide range of STM algorithms and their fair (unbiased) comparison in a common STM framework. We illustrate the benefits of our approach by analyzing its impact in two popular TM algorithms with several transactional workloads, TL2 and multi-versioning, each befitting out-place and in-place respectively.

Received . . .

KEY WORDS: Software Transactional Memory, Java, Bytecode Instrumentation, Data Structures

1. INTRODUCTION

Software Transactional Memory (STM) algorithms differ in the properties and guarantees they provide. Among others differences, one can list distinct strategies used to read (visible or invisible) and update memory (direct or deferred), the consistency (opacity or snapshot isolation) and progress guarantees (blocking or non-blocking), the policies applied to conflict resolution (contention management), and the sensitivity to interactions with non-transactional code (weak or strong atomicity). Some STM frameworks (e.g., DSTM2 [1] and Deuce [2]) address the need of experimenting with new STM algorithms and their comparison, by providing a unique transactional interface and different alternative implementations of STM algorithms. However, STM frameworks tend to favor the performance for some classes of STM algorithms and disfavor others. For instance, the Deuce framework favors algorithms like TL2 [3] and LSA [4], which are resilient to false sharing of transactional metadata (such as ownership records) stored in an external table, and disfavor multi-version algorithms, which require unique metadata per memory location. This paper addresses

[*]Correspondence to: Departamento de Informática, FCT-UNL, Quinta da Torre, 2829-516 Caparica, Portugal. E-mail: joao.lourenco@fct.unl.pt

[†]This research was partially supported by the EU COST Action IC1001 (Euro-TM) and the Portuguese *Fundação para a Ciência e Tecnologia* in the research project PTDC/EIA-EIA/113613/2009 (Synergy-VM), and the research grant SFRH/BD/41765/2007.

this issue by proposing an extension to the Deuce framework that allows the efficient support of transactional metadata records per memory location.

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, and may be composed by, e.g., locks, timestamps or version lists, will henceforth be referred as *metadata*. Metadata is stored either adjacent to each memory location (*in-place* strategy), or in an external table (*out-place* or *external* strategy). STM libraries for imperative languages, such as C, frequently use the out-place strategy, while those addressing 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 such a pre-allocated table avoids the overhead of dynamic memory allocation, but incurs in the overhead for evaluating the location-to-metadata mapping function. The bounded size of the external table induces a false sharing situation, where multiple memory locations share the same table entry and hence the same metadata, in a *many-to-one* relation between memory locations and metadata units.

The in-place strategy is usually implemented by using the *decorator* design pattern [5], by extending the functionality of an original class by wrapping it in a *decorator* class that 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 heavily rewritten to use the decorator classes instead of the original ones. The *decorator* pattern based technique bears two other problems: additional overhead for non-transactional code, and multiple difficulties while working with primitive and array types. This technique implements a *one-to-one* relation between memory locations and metadata units, thus no false sharing occurs. Riegel et al. [6] briefly describe the trade-offs of using in-place *versus* out-place strategies.

Deuce 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 to intercept the read and write memory accesses by injecting call-backs to the STM algorithm. These call-backs receive the referenced memory address as argument, hence limiting the range of viable STM algorithms to be implemented by forcing an out-place strategy. To implement an algorithm in Deuce that requires a one-to-one relation between metadata and memory locations, such as a multi-version algorithm, one needs to use a external table that handles collisions, which significantly degrades the throughput of the algorithm.

This paper reports on an extension of our previous work [7], which proposes a novel approach to support the in-place metadata strategy without making use of the decorator pattern, and thoroughly evaluates its implementation in Deuce. This extension allows the efficient implementation of algorithms requiring a one-to-one relation between metadata and memory locations, such as multi-version algorithms. The developed extension has the following properties:

Efficiency The extension fully supports primitive types, even in transactional code. It does not rely on an external mapping table, thus providing fast direct access to the transactional metadata. Transactional code does not require the extra memory dereference imposed by the decorator pattern. Non-transactional code is in general oblivious to the presence of metadata in objects, hence no significant performance overhead is introduced. And we propose a solution for supporting transactional n-dimensional arrays with a negligible overhead for non-transactional code.

Flexibility The extension supports both the original out-place and the new in-place strategies simultaneously, hence it is fully backwards compatible and imposes no restrictions on the nature of the STM algorithms to be used, nor on their implementation strategies.

Transparency The extension automatically identifies, creates and initializes all the necessary additional metadata fields in objects. No source code changes are required, although some light transformations are applied to the non-transactional bytecode. The new transactional array types — that support metadata at the array cell level — are compatible with the standard arrays, therefore not requiring pre- and post-processing of the arrays when used as arguments in calls to the standard JDK 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 zero or negligible performance overhead.

Compliance The extension and bytecode transformations are fully-compliant with the Java specification, hence supported by standard Java compilers and JVMs.

The paper is structured as follows. Section 2 describes the Deuce framework and its out-place strategy. Section 3 describes properties of the in-place strategy, its implementation, and its limitations as an extension to Deuce. Section 4 describes the implementation of three multi-version algorithms; one using the out-place strategy and the other two using the in-place strategy. We evaluate our implementation with several benchmarks and algorithms in Section 5, and discuss the related work in Section 6. We finish with some concluding remarks in Section 7.

2. DEUCE AND THE OUT-PLACE STRATEGY

Algorithms such as TL2 [3] or LSA [4] 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 by two different memory locations. In these algorithms, the false sharing may wrongly cause transactions to fail and abort, hurting the system performance but never compromising their correctness.

The out-place strategy suits algorithms where metadata information does not depend on the memory locations, such as locks and timestamps, but not algorithms that need to keep location-dependent metadata information, e.g., multi-version algorithms. The out-place implementations of these algorithms require a mapping table with collision lists, which significantly degrades performance.

Deuce provides the STM algorithms with a unique identifier for each object field, composed by the 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 the key to any map implementation that associates the object's field with the transactional metadata. Likewise for arrays, 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 Deuce supplies a single `@Atomic` Java annotation, and relies heavily on bytecode instrumentation to provide a transparent transactional interface to application developers, which are unaware of how the STM algorithms are implemented and of the strategies used to store the transactional metadata.

The performance of STM algorithms varies with both the hardware and the transactional workload, and a thorough experimental evaluation is required to assess the optimal combination of the triple hardware–algorithm–workload. Deuce is an extensible STM framework that may be used to address such comparison of different STM algorithms. However, Deuce is biased towards the out-place strategy, allowing very efficient implementations for some algorithms like TL2 and LSA, but hampering some others, like the multi-version oriented STM algorithms.

To support the out-place strategy, Deuce identifies an object's field by the object reference and the field's logical offset. This logical offset is computed 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. Within a memory transaction, when there is a read or write memory access to a field f of an object O, or to the array element $A[i]$, the run-time passes the pair (O, f^o)

```
public interface Context {
  void init(int atomicBlockId, String metainf);
  boolean commit();
  void rollback();

  void beforeReadAccess(Object obj, long field);

  int onReadAccess(Object obj, int value, long field);
  // ... onReadAccess for the remaining types

  void onWriteAccess(Object obj, int value, long field);
  // ... onWriteAccess for the remaining types
}
```

Figure 1. `Context` interface for implementing an STM algorithm.

or (A, i) respectively as the argument to the call-back function. The STM algorithm shall not differentiate between field and array accesses. If an algorithm wants 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) or (A, i). STM algorithm implementations must comply with a well defined Java interface, as depicted in Figure 1. The methods specified in the interface are the call-back functions that are injected by the instrumentation process in the application code. For each read and write of a field of an object, the methods `onReadAccess` and `onWriteAccess`, are invoked respectively. The method `beforeReadAccess` is called before the actual read of an object's field.

We have extended Deuce to support an efficient in-place strategy, in addition to the already existing out-place strategy, while keeping the same transparent transactional interface to the applications.

3. SUPPORT FOR IN-PLACE STRATEGY

In our approach to extend Deuce to support the in-place strategy, we replace the previous pair of arguments to call-back 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 any 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 2, 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` (see Figure 3). The constructor of `TxField` class receives the object reference and the logical offset of the field. All subclasses must call this constructor. For array elements, we created specialized metadata classes for each primitive type in Java, the `TxArr*Field` classes, where $\star$ ranges over the Java primitive types[†]. All the `TxArr*Field` classes extend from `TxField`, providing the STM algorithm with a simple and uniform interface for call-back functions.

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

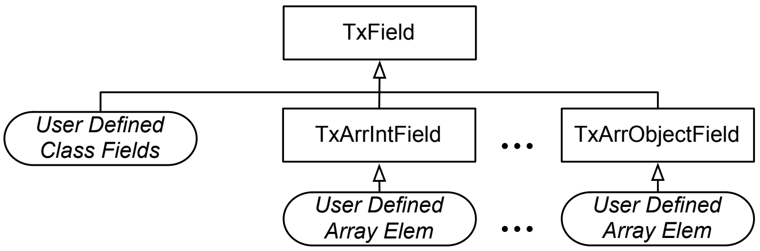

Figure 2. Metadata classes hierarchy.

```
  public class TxField {
    public Object ref;
    public final long offset;

    public TxField(Object ref, long offset) {
      this.ref = ref;
      this.offset = offset;
    }
  }
```

Figure 3. `TxField` class.

```
public interface ContextMetadata {
  void init(int atomicBlockId, String metainf);
  boolean commit();
  void rollback();

  void beforeReadAccess(TxField field);
  int onReadAccess(int value, TxField field);
  // ... onReadAccess for the remaining types

  void onWriteAccess(int value, TxField field);
  // ... onWriteAccess for the remaining types
}
```

Figure 4. `Context` interface for implementing an STM algorithm supporting in-place metadata.

We defined a new interface for the call-back methods (see Figure 4). In this new interface, the read and write call-back functions (`onReadAccess` and `onWriteAcess` respectively) receive only the metadata `TxField` object, not the object reference and logical offset of the `Context` interface. This new interface coexists with the original one in Deuce, allowing new STM algorithms to access the in-place metadata while ensuring backward compatibility.

The `TxField` class can be extended by the STM algorithm programmer to include additional information required by the algorithm for, e.g., locks, timestamps, or version lists. 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 Figure 5. The programmer may register a different metadata class for each kind of data type, either for class field types or array types. As shown in the example of Figure 5, the programmer registers the metadata implementation class `TL2IntField` for the fields of **`int`** type, by assigning the name of the class to the `fieldIntClass` annotation property.

The STM algorithm must implement the `ContextMetadata` interface (Figure 4) that includes a call-back function for the read and write operations on each Java type. These functions always receive an instance of the super class `TxField`, but no confusion arises from there, as each

```
@InPlaceMetadata(
  fieldObjectClass="TL2ObjField",
  fieldIntClass="TL2IntField",
  ...
  arrayObjectClass="TL2ArrObjectField",
  arrayIntClass="TL2ArrIntField",
  ...
)
public class TL2Context implements ContextMetadata {
  ...
}
```

Figure 5. Declaration of the STM algorithm specific metadata.

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

$\Longrightarrow$

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

Figure 6. Example transformation of a class with the in-place strategy.

algorithm 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 call-back functions. We will explain separately the management of metadata objects for class fields and for array elements.

3.1.1. Adding Metadata to Class Fields During the execution of a transaction, there must be a metadata object f^m for each accessed field f of object O. Ideally, the metadata object f^m would be accessible by a single dereference operation from object O, and this was achieved by adding a new metadata field (of the appropriate type) for each field declared in a class C. The general rule for this process can be described as: given a class C that has a set of declared fields $F = \{f_1, \ldots, f_n\}$, for each field $f_i \in F$ we add a metadata object field f^m_{i+n} to C, such that the class ends with the set of fields $F^m = \{f_1, \ldots, f_n, f^m_{1+n}, \ldots, f^m_{n+n}\}$, where the field f_i is associated with the metadata field f^m_{i+n} for any $i \leq n$. In Figure 6 we show a concrete example of the transformation of a class with two fields.

Instance and static fields are expected to have instance and static metadata fields, respectively. Thus, instance metadata fields are initialized in the class constructor, while static metadata fields are initialized in the static initializer (`static { ... }`). This ensures that whenever a new instance of a class is created, the corresponding metadata objects are also new and unique, while static metadata objects are the same in all instances. Since a class can declare multiple constructors that can call each other, using the *telescoping constructor* pattern, blindly instantiating the metadata fields in all constructors would be redundant and impose unnecessary stress on the garbage collector. Therefore, the creation and initialization of metadata objects only takes place in the constructors that do not rely in another constructor to initialize its target.

Opposed to the transformation approach based in the *decorator* pattern, where primitive types must be replaced with their object equivalents (e.g., in Java 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 auto-boxing and its overhead.

3.1.2. Adding Metadata to 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 and add some more information to each cell, is to change the original array to an array of objects that wrap the original value and the additional information. This straight-forward transformation has strong implications in the application, as code statements accessing the original array or array elements will now have to be rewritten to use the new array type or wrapping class respectively. This problem is even more complex if the new arrays with wrapped elements are to be manipulated by non-instrumented libraries, such as the JDK libraries, which are unaware of the new array types.

While the instrumentation process can replace the original arrays with the new arrays where needed, the straight-forward transformation approach needs to be able to revert back to the original arrays when presented with non-instrumented code. For example, consider that the application code is invoking the non-instrumented method `Arrays.binarySearch(int[], int)` from the Java platform. Throughout the instrumented code `int[]` has been replaced by a new type, which we denote `IntWrapper[]`. As the `binarySearch` method was not instrumented, the array parameter remains of type `int[]`, thus one needs to construct an `int[]` with the same state of the `IntWrapper[]`, which can then be passed as an argument to the `binarySearch` method. From the caller perspective, the non-instrumented method itself is a black box which may have modified some array cells.[‡] Hence, unless we were to build some kind of black/white list with such information for *all* methods, the values from `int[]` have to be copied back to `IntWrapper[]`. All these memory allocation and copies significantly hamper the performance when executing non-instrumented code, which should not be affected due to transactional-related instrumentation. We call this straight-forward approach the *naïve solution*.

The solution we propose is also based on changing the type of the array to be manipulated by the instrumented application code, but with minimal impact on the performance of 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 in the declaration of the base array is changed to the type of the corresponding sibling array (`TxArr*Field`), as shown in Figure 7. 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. With this approach for adding metadata support to arrays, the original array can still be retrieved with two dereferences from the sibling `TxArr*Field` array, with minimal overhead implications. Since the original array serves as the backing store, no memory allocation or copies need to be performed, even when array elements are changed by non-instrumented code. We call our proposed solution the *efficient solution*.

Non-transactional methods that have arrays as parameters are also instrumented to replace the array type by the corresponding sibling `TxArr*Field`. For non-instrumented methods, relying on the method signature is not enough to identify the need to revert to primitive arrays. Take, for example, the `System.arraycopy(Object, int, Object, int, int)` method from the Java platform. The signature refers `Object` but it actually receives arrays as arguments. We identify these situations by inspecting the type of the arguments on a *virtual stack*[§] and if an array is found, despite the method's signature, we revert to primitive arrays. The value of an array element is then obtained by dereferencing the pointer to the original array kept in the sibling, as illustrated in Figure 8. When passing an array as argument to an uninstrumented method (e.g., from the

[‡]In this example we used the `binarySearch` method which does not modify the array, but in general we do not know.

[§]During the instrumentation process we keep the type information of the operand stack.

```
class D {
  int[] a; //base array
}
```

⟹

```
class D {
  TxArrIntField[] a;
  TxField a_metadata;
}

class TxArrIntField {
  int[] array; //base array
  int index;
}
```

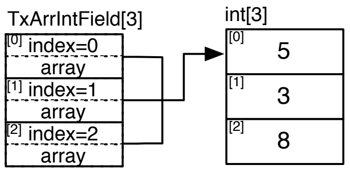

Figure 7. Memory structure of a `TxArrIntField` array.

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

⟹

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

Figure 8. Example transformation of array access in the in-place strategy.

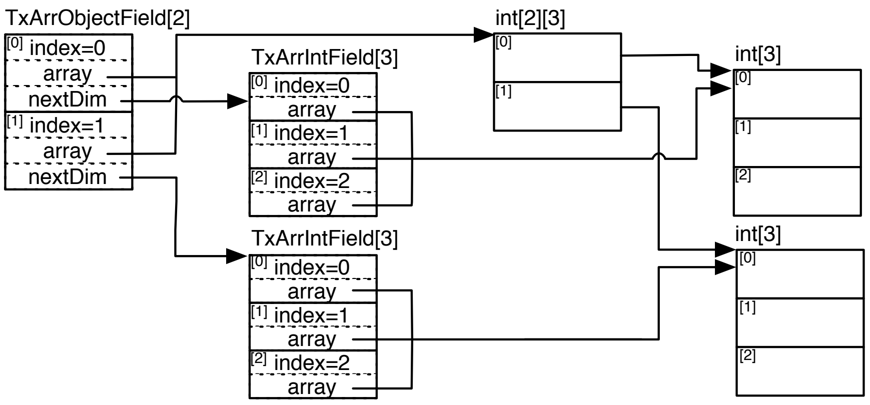

Figure 9. Memory structure of a multi-dimensional `TxArrIntField` array.

JDK library), we can just pass the original array instance. Although the instrumentation of non-transactional code adds an extra dereference operation when accessing an array, we still do avoid the auto-boxing of primitive types, which would impose a much higher overhead.

3.1.3. Adding Metadata to 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 an additional field, `nextDim`, which may be null in the case of a unidimensional 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 9 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.

Table I provides a comparison between the regular primitive arrays, used in the out-place strategy, and our instrumented arrays, used in the in-place strategy. The instrumented arrays follow the strategy described above. For accessing a cell in a n-dimensional array (Table I, second column),

Table I. Comparison between primitive and transactional arrays.

Arrays	Access n^{th} dimension	Objects	Non-transactional methods
Primitive arrays	n *derefs*	$\sum_{i=1}^{n} l_{i-1}$	—
Instrumented arrays	$2n+1$ *derefs*	$\sum_{i=1}^{n} 2l_{i-1} + (l_i \times l_{i-1})$	2 *derefs*

n (dimensions), l_i (length of i^{th} dimension)

in a primitive array it takes n object dereferences, i.e., dereferencing all intermediate dimension arrays and directly accessing the cell. With our array instrumentation it takes $2n + 1$ dereferences, introducing an extra dereference per dimension ($2n$) because each cell is now a `TxArr*Field`. Since the original array is used as the backing store, there is an additional dereference of the original array in the last dimension to access the value. Regarding the number of objects that each approach needs for an n-dimensional array (Table I, third column), for simplicity's sake let's assume that all intermediate i^{th}-dimensional arrays have the same length, l_i. Primitive arrays have l_{i-1} objects per dimension, i.e., each dimension's array cell is a reference to another array, except in the last dimension. The instrumented arrays have twice the number of arrays, i.e., $2l_{i-1}$, corresponding to the the original array (which is kept) and the sibling array, plus an extra `TxArr*Field` in every array cell ($l_i \times l_{i-1}$). When an array is to be used by a non-instrumented method (Table I, fourth column), the instrumented arrays require two dereferences to obtain the backing-store primitive array, i.e., dereferencing the sibling array followed by a dereference of a `TxArr*Field` cell, from which the `array` field can be used. These two dereferences required by our instrumented arrays contrast with the expensive memory allocation and copies necessary for the straight-forward naïve solution, described in 3.1.2.

3.2. Instrumentation Limitations

Some Java core classes, mostly in the `java.lang` package, are loaded and initialized during the JVM bootstrap. Because these classes are loaded upon JVM startup, they can either be redefined online after the bootstrap, or require an offline, static, instrumentation. Online redefinition of classes has many and strong limitations, and its support is an optional functionality for JVMs [8]. For this reason, instead of online redefinition of bootstrap-loaded classes, Deuce provides an offline instrumentation process.

Most JVMs are very sensitive with regard to the order in which classes are loaded during the bootstrap. If that order is changed due to the execution of instrumented code during the bootstrapping phase (i.e., because instrumented code may depend on certain classes that need to be loaded before the instrumented code can be executed), the JVM may crash [9]. The Deuce online instrumentation injects static fields and their initialization, which would disrupt the class loading order if done on bootstrap-loaded classes. They solve this problem in the offline instrumentation by creating a separate class to hold the fields instead. This is possible because the necessary fields are static.

The instrumentation to support the in-place metadata strategy is more complex, requiring the injection of instance fields and modifying arrays. For this reason, the instrumentation of bootstrap-loaded classes is not supported by our current instrumentation process, as these transformations disrupt the bootstrap class loading order by loading the metadata and transactional array classes.

At the moment there is no support for structural modification of arrays inside non-instrumented code, such as the java run-time library, because the solution for metadata at array element level relies on a sibling array where a structural invariant exists between the sibling array and the original array. If a non-instrumented method modifies the original array, the structural invariant is broken and both structures become different.

4. USE CASE: MULTI-VERSION ALGORITHM IMPLEMENTATION

Our main purpose for extending Deuce with support for in-place metadata was to allow the efficient implementation of a class of STM algorithms that require a *one-to-one* relation between memory locations and their metadata. Multi-version based algorithms fit into that class, as they associate a list of versions (holding past values) with each memory location.

To evaluate our extension to Deuce, we started by implementing the JVSTM multi-version algorithm as described in [10][¶]. We did two implementations of the algorithm, one using the original Deuce interface and an out-place strategy (referred to as `jvstm-outplace`), and another using our new interface and extension supporting an in-place strategy (referred to as `jvstm-inplace`). We also implemented a new multi-version algorithm, based in TL2 (referred to as `mvstm`), which has a bounded number of versions for each memory location and does not use a global lock in the commit. Instead, at commit time it uses a lock per memory location and only the write-set is locked. In the following sections we describe the implementation details of each of the above algorithms.

4.1. Implementing JVSTM

The JVSTM algorithm defines the notion of version box (*vbox*), which maintains a pointer to the head of an unbounded list of versions, where each version is composed by a timestamp and the data value. Each version box represents a distinct memory location. The timestamp in each version corresponds to the timestamp of the transaction that created that version, and the head of the version list always points to the most recent version.

During the execution of a transaction, the read and write operations are done in versioned boxes, which hold the data values. For each write operation a new version is created and tagged with the transaction timestamp. For read operations, the version box returns the version with the highest timestamp less than or equal to the transaction's timestamp. A particularity of this algorithm is that read-only transactions never abort, neither do write-only transactions. Only read-write transaction may conflict, thus aborting.

On committing a transaction, a global lock must be acquired to ensure mutual exclusion with all other concurrent transactions. Once the global lock is acquired, the transaction validates the read-set, and in case of success, creates the new version for each memory location that was written, and finally releases the global lock. To prevent version lists from growing indefinitely, versions that are no more necessary are cleaned up by a *vbox* garbage collector.

To implement the JVSTM algorithm, we need to associate a *vbox* with each field of each object. For the sake of the correctness of the algorithm, this association must guarantee a relation of *one-to-one* between the *vbox* and the object's field. We will detail the implementation of this association for both, the out-place and the in-place strategies.

4.1.1. Out-Place Strategy To implement JVSTM algorithm in the original Deuce framework, which only supports the out-place strategy, the *vboxes* must be stored in an external table[‖]. The *vboxes* are indexed by a unique identifier for the object's field, composed by the object reference and the field's logical offset.

Whenever a transaction performs a read or write operation on an object's field, the respective *vbox* must be retrieved from the table. In the case where the *vbox* does not exists, we must create one and add it into the table. These two steps, verifying if a *vbox* is present in the table and creating and inserting a new one if not, must be performed atomically, otherwise we would incur in the case where two different *vboxes* may be created for the same object's field. Once the *vbox* is retrieved from the table, either it is a read operation and we look for the appropriate version using the transaction's timestamp and return the version's value, or it is a write operation and we add an entry to the transaction's write-set.

[¶]Recent ongoing work in JVSTM algorithm [11] reports considerable performance improvements over the version we used in this paper.
[‖]We opted to use a concurrent hash table from the `java.util.concurrent` package.

```
public class VBox extends TxField {
  protected VBoxBody body;

  public VBox(Object ref, long offset) {
    super(ref, offset);
    body = new VBoxBody(read(), 0, null);
  }

  // ... methods to access and commit versions
}
```

Figure 10. `VBox` in-place implementation.

We use weak references in the table indices to reference the *vbox* objects and not hamper the garbage collector from collecting old objects. Whenever an object is collected our algorithm is notified in order to remove the respective entry from the table.

Despite using a concurrent hash map, this implementation suffers from a high overhead penalty when accessing the table, since it is a point of synchronization for all the transactions running concurrently. This implementation (`jvstm-outplace`) will be used as a base reference when comparing with the implementation of JVSTM algorithm using the in-place strategy (`jvstm-inplace`) and with the new multi-version algorithm (`mvstm`).

4.1.2. In-Place Strategy The in-place version of JVSTM algorithm makes use of the metadata classes to hold the same information as the *vbox* in the out-place variant. This will allow direct access to the version list whenever a transaction is reading or writing.

We extend the *vbox* class from the `TxField` class as shown in Figure 10.

The actual implementation creates a `VBox` class for each Java type in order to prevent the boxing and unboxing of primitive types. When the constructor is executed, a new version with timestamp zero is created, containing the current value of the field identified by object `ref` and logical offset `offset`. The value is retrieved using the private method `read()`.

The code to create these `VBox` objects during the execution of the application is inserted automatically by our bytecode instrumentation process. The lifetime of an instance of the class `VBox` is the same as the lifetime of the object `ref`. When the garbage collector decides to collect the object `ref`, all metadata objects of class `VBox` associated with each field of the object `ref`, are also collected.

Our evaluation (see Section 5) shows that the direct access to the version list allowed by the in-place strategy will greatly benefit the performance of the algorithm. The evaluation of both variants of the JVSTM algorithm (`jvstm-outplace` and `jvstm-inplace`) revealed a scalability bottleneck, which motivated the development of a new multi-version algorithm (`mvstm`).

4.2. MVSTM – A New Multi-Version Algorithm

We developed and implemented a new multi-version algorithm (MVSTM) using the in-place metadata support and inspired in TL2. It defines a fixed size for the list of versions, imposing a bound in the number of versions for each memory location, and at commit time it uses a lock per memory location listed in the write-set.

The structure for each version is the same as in JVSTM. Each version is composed by a timestamp, which corresponds to the timestamp of the transaction that committed the version, and the data value. Each metadata object has a pointer to the head of a version list with a fixed size. Whenever a transaction commits a new version, and the maximum size of the version list is reached, we discard half of the older versions. This decision allows to limit the memory used by the algorithm and avoid complex garbage collection algorithms to remove old versions. The drawback of this approach is that read-only operations can now abort because they may try to read a version that was already removed.

The commit operation is similar to the TL2 algorithm. Read-only transactions may commit without any additional validation procedure, whilst read-write transactions need to lock the write-set entries and then validate their read-set. In the case of a successful validation of the read-set, the transaction applies the write-set by creating a new version for each entry in the write-set, and finally unlocks the write-set locks. This locking scheme allows two transactions to commit concurrently if their write-sets are disjoint.

4.3. Supporting the Weak Atomicity Model

Multi-version algorithms read and write the data values from and to the list of versions. This implies that all accesses to fields in shared objects must be done inside a memory transaction, and thus multi-version algorithms require a *strong atomicity* model [12].

Deuce does not provide a strong atomicity model, and hence it is possible to have non-transactional accesses to fields of objects that were also accessed inside memory transactions. This hinders the usage of multi-version algorithms in Deuce. One approach to address this problem is to rewrite the existing benchmarks to wrap all accesses to shared objects inside an atomic method, but such code changes are always a cumbersome and error prone process. We addressed this problem by proposing an adaptation for the multi-version algorithms to support the *weak atomicity* model.

When using a weak atomicity model, updates made by non-transactional code to object fields are not seen by transactional code and, on the other way around, updates made by transactional code are not seen by non-transactional code. The key idea for our proposal is to store the value of the latest version in the object's field instead of at the head of the version list. When a transaction needs to read a field of an object, it requests the version corresponding to the transaction timestamp. If it receives the head version, then it reads the value directly from the object's field instead, otherwise it reads the value from the appropriate version node.

The problem with this approach is how to guarantee atomicity when committing a new version, because now we have two steps: adding a new version node to the head of the list and updating the field's value. These two steps must appear atomic with respect to the other concurrent transactions. Our solution is to create a temporary new version with an infinite timestamp, making it invisible for other concurrent transactions, until we update the value and then change the timestamp to its proper value.

The pseudo-code of the commit algorithm of a new version is listed below. In this list, t is the timestamp of the transaction that is performing the commit, t_∞ is the highest timestamp, val is the value to be written, v_h is the pointer to the head version, and f is the object's field.

```
1. $v_n := \mathsf{create\_version}(val, t_\infty, v_h)$
2. $v_h.\,value := \mathsf{read}(f)$
3. $v_h := v_n$
4. $\mathsf{write}(\mathsf{f}, val)$
5. $v_h.\,timestamp := t$
```

The first step is to create a new version with the new value to be written in field f, an infinite timestamp, and the pointer to the current head version. Then we update the value of the head version with the current value of field f. This update is safe because until this point transactions that retrieve the head version read the value directly from field f, as described above. In the third step, we make the new version v_n the current head version and it becomes visible to all concurrent transactions. This version will never be accessed by any concurrent transaction because of the infinite timestamp. Then we can safely update the field's value in the fourth step because no concurrent transaction accesses the head version. In the last step we change the timestamp of the current head version to its proper value making accessible to concurrent transactions.

We adapted the three multi-version algorithms (`jvstm-outplace`, `jvstm-inplace` and `mvstm`) to include the new commit algorithm, which enabled the execution all benchmarks available in the Deuce framework with no modification.

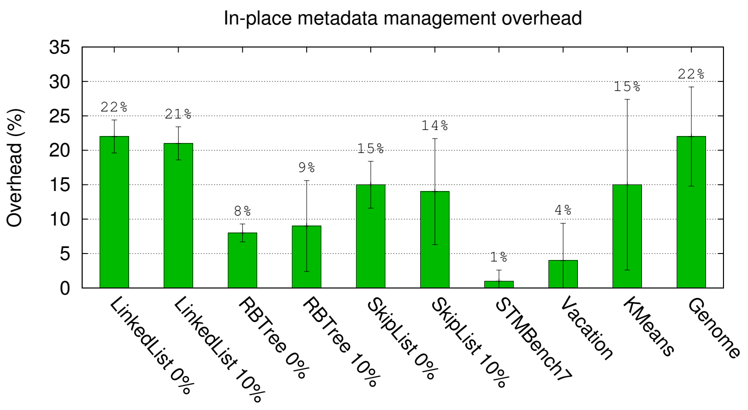

Figure 11. Performance overhead measure of the usage of metadata objects relative to out-place TL2.

5. PERFORMANCE EVALUATION

We evaluated our work in three dimensions: the performance overhead introduced by the support of the in-place strategy; and the performance improvements achieved by multi-versioning STM algorithms when using our in-place strategy; the memory consumption of some STM algorithms that use the in-place and/or out-place strategies, while running several benchmarks. To measure the transactional throughput we used the vanilla micro-benchmarks[**] available in the Deuce framework, the Vacation, KMeans and Genome benchmarks from the STAMP[††] test suite [13], and the STMBench7[‡‡] benchmark [14]. All these benchmarks were executed in our extension of Deuce with in-place metadata with no changes whatsoever, as all the necessary bytecode transformations were performed automatically by our instrumentation process.

The benchmarks were executed on a computer with four AMD Opteron 6168 12-Core processors @ 1.9 GHz with 12×512 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.

5.1. Performance Overhead

To evaluate the performance overhead of our extension, we compared the performance of the TL2 algorithm as provided by the original Deuce distribution, with another implementation of TL2 (`tl2-overhead`) using the new interface provided by our Deuce extension (Figure 4). The original Deuce interface for call-back 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 `tl2-overhead` implementation 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 make a reasonable estimation of the performance overhead introduced by the management of the metadata object fields and sibling arrays.

Figure 11 depicts the performance overhead, and its standard deviation, of `tl2-overhead` with respect to the original Deuce TL2 implementation for several benchmarks, with executions ranging form 1 to 48 threads. In average, the overhead of the additional management of metadata objects and sibling arrays is about 13%. The benchmarks that use metadata objects for arrays (SkipList, KMeans, Genome) have in general higher overhead than the benchmarks that only use metadata

[**]LinkedList, RBTree, and SkipList. Run parameters: `-i 16384` (initial size) `-r 262144` (range).
[††]Run parameters: Vacation `-q90 -u90 -r32768 -t262144 -n8` (vacation-high+); KMeans `-m40 -n40 -t0.001 -irandom-n16384-d24-c16` (kmeans-low+); Genome `-g512 -s32 -n32768` (genome+).
[‡‡]Run parameters: `-w r -g stm --no-traversals`.

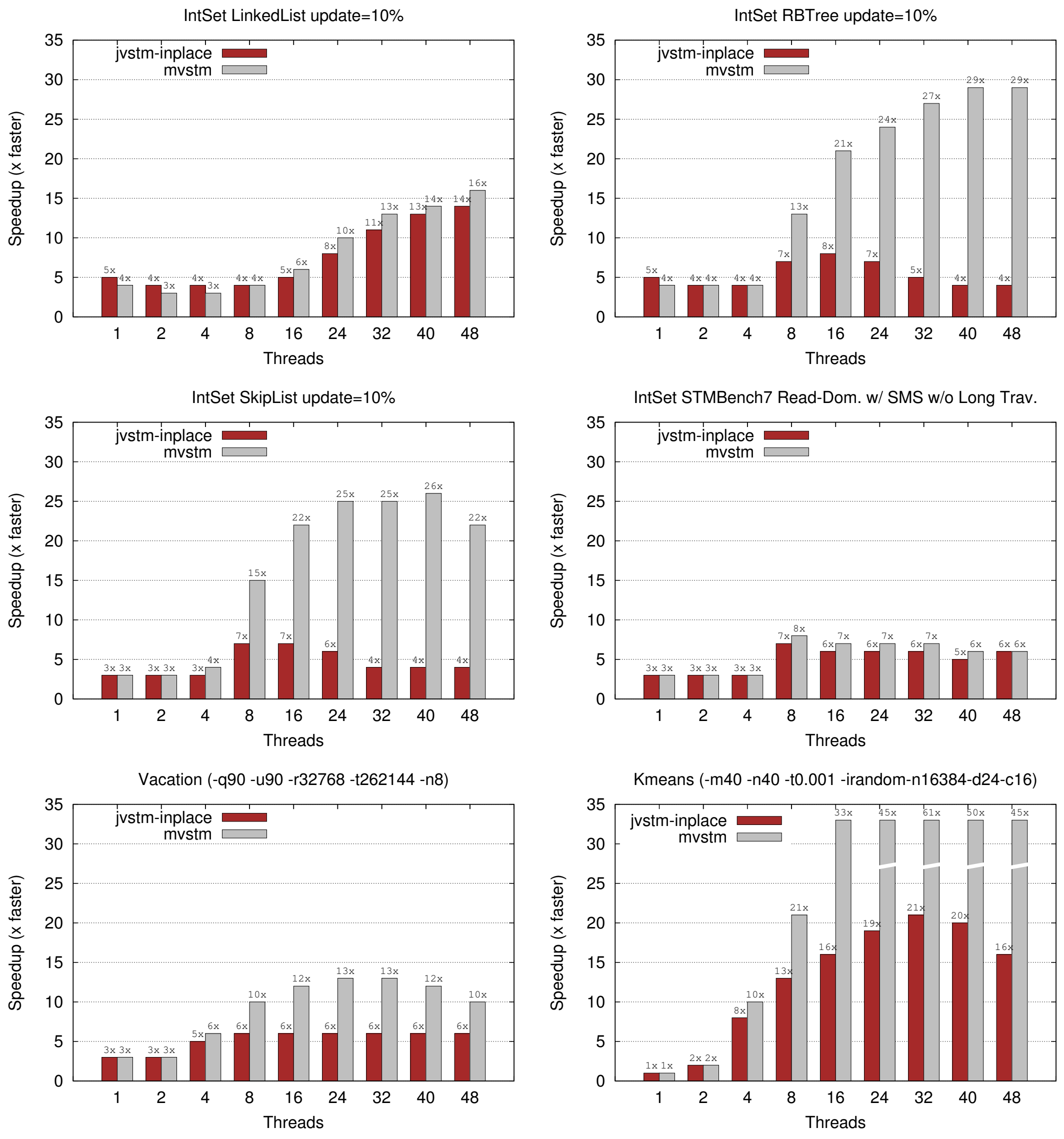

Figure 12. Speedup of `jvstm-inplace` and `mvstm` relative to `jvstm-outplace`.

objects for class fields (RBTree, STMBench7, Vacation). The LinkedList benchmark does not use metadata objects for arrays and still has a high overhead. This benchmark has long running transactions that perform a very large number of read operations, and our extension requires an external table lookup and an additional object dereference to retrieve the metadata object for each memory read operation. The transactions in STMBench7 are computationally heavier, which hides the small overhead introduced by the management of in-place metadata.

The micro-benchmarks were all tested in two scenarios: with a read-only workload (0% of updates), and a read-write workload (10% of updates). Although the differences in the average overhead are negligible for all three micro-benchmarks, there is a clear trend for a larger standard deviation in read-write workloads. This trend is justified by the higher variability in the behavior of the read-write transactions, due to the conflicts and changes in the state of the program.

5.2. Performance

From the evaluation of the in-place management overhead, we concluded that this strategy is a viable option for implementing algorithms biased to in-place transactional metadata. Hence, we implemented and evaluated two versions of the JVSTM algorithm as proposed in [10], one in the original Deuce using the native out-place strategy (`jvstm-outplace`), and another in the extended Deuce using our in-place strategy (`jvstm-inplace`), as described in Section 4.1. We also implemented and evaluated the new multi-version algorithm (`mvstm`), as described in Section 4.2.

Figure 12 depicts the speedup of our implementation of the `jvstm-inplace` and `mvstm` algorithms relative to our implementation of the `jvstm-outplace` algorithm. The speedup observed for the micro-benchmarks, where transactions are small and contention is low, shows that the multi-versioning algorithm greatly benefit from our in-place support. This benefit is even more evident for the `mvstm` algorithm, which scales very well with the number of threads.

In the Vacation and KMeans macro-benchmarks, the in-place multi-version algorithms perform much better than the out-place multi-version algorithm, and also scale well with the number of threads. The STMBench7 macro-benchmark has many long-running transactions and the overall throughput for all the algorithms is relatively low. Even so, the in-place algorithms are in average $5\times$ and $6\times$ faster for the `jvstm-inplace` and `mvstm` algorithms respectively.

These results also show that the `mvstm` algorithm clearly outperforms both the `jvstm-outplace` and the `jvstm-inplace` algorithms, mainly due to the way versions are managed in `mvstm`, eliminating the need of a garbage collector for old versions and the associated overhead.

These results prove that our strategy to support in-place metadata in Deuce gave it leverage to implement algorithms that need a one-to-one relation between memory locations and transactional metadata, thus enabling the fair comparison of a wider range of STM algorithms, including those that could not be implemented efficiently in the original Deuce framework. In Figure 13 we show an absolute performance comparison, between the TL2 and LSA (out-place) algorithms and the three multi-version algorithms: `jvstm-outplace`, `jvstm-inplace`, and `mvstm`. In all benchmarks, `jvstm-outplace` clearly under-performs all the other algorithms, with no scalability and low throughput, confirming our claims that it is not possible to implement efficiently a multi-version algorithm in the original Deuce. On the other hand, the two multi-version algorithms implemented with the in-place support (`jvstm-inplace` and `mvstm`) clearly compete with the very performant TL2 in some workloads, evidencing that a good infrastructure support for those algorithms was a key requirement for their comparative evaluation.

5.3. Memory Consumption

Figure 14 depicts the maximum memory used for each benchmark that we executed. The difference between the memory used by the `tl2` algorithm (out-place) and the `tl2-overhead` algorithm (in-place) is very low for benchmarks that do not create many objects during their execution (LinkedList, Vacation, KMeans, and Genome). In benchmarks that create many objects (RBTree, SkipList, and STMBench7), `tl2-overhead` uses about $3\times$ more memory than `tl2`.

In all benchmarks, `jvstm-outplace` and `jvstm-inplace` always use a large amount of memory. This is due to the JVSTM algorithm properties, which do not limit the size of the version lists. Additionally, in the case of `jvstm-outplace`, the external table used to hold the *vboxes* also requires a large amount of memory. In the SkipList benchmark, `mvstm` has the largest memory footprint, even higher than both `jvstm` variants. This result is due to the poor performance of `jvstm` variants, which limits their memory usage during the benchmark. On the other hand, `mvstm` performs very well, and due to the intensive use of array structures, the consumed memory is also very high. The `mvstm` algorithm has generally a much lower memory usage pattern, confirming that bounding the size of version lists is a good design choice for multi-version algorithms, with advantages for both performance and memory usage.

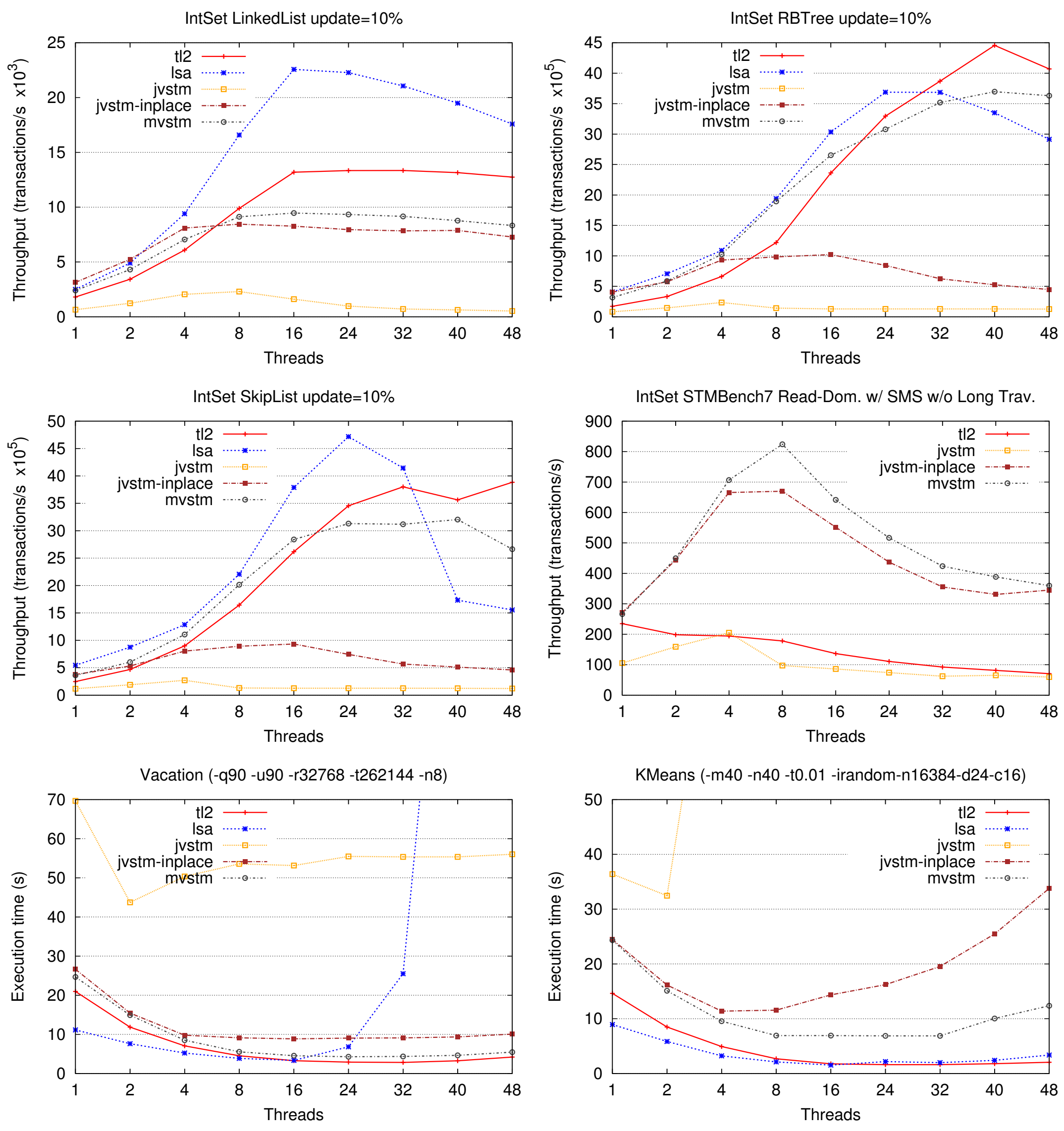

Figure 13. Performance comparison between TL2, LSA, and the multi-version algorithms.

6. 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, but are normally biased towards either the in-place or the out-place strategy.

DSTM2 [1] 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.

Deuce [2], 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

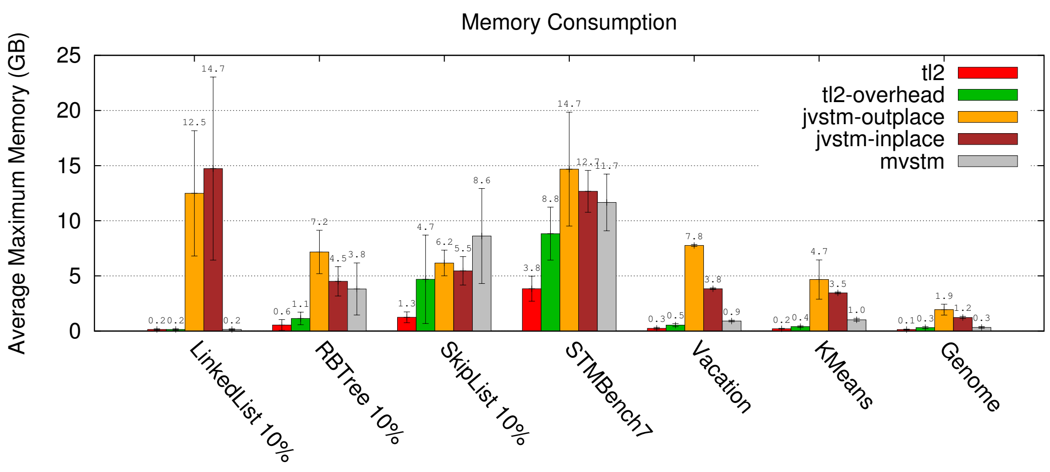

Figure 14. Maximum memory usage for each benchmark.

several STM algorithms, and relies on Java bytecode instrumentation to intercept transaction boundaries and transactional memory accesses and invoke developer-defined call-back functions. Deuce has a strong bias towards the out-place approach.

STM algorithms such as TL2 [3], LSA [4] and SwissTM [15] are usually implemented using an out-place strategy, thus viable for use in Deuce. Others, such as JVSTM [10, 11] and SMV [16] are better fit for the in-place strategy and impracticable for Deuce. Our extension of Deuce overcomes this limitation and allows the efficient implementation of algorithms using any of those strategies and their fair comparison.

Anjo et al. [17] 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 Deuce 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. [18] are orthogonal to our work and can also be applied to algorithms using the new in-place strategy, thus increasing the overall performance.

7. CONCLUDING REMARKS

To the best of our knowledge, the extension of Deuce 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.

We evaluated our system by measuring the overhead introduced by our new in-place strategy with respect to the TL2 algorithm implementation provided in Deuce 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 its 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 three multi-version STM algorithms: two of them using the newly proposed in-place strategy, and the other using an out-place strategy resorting to an external mapping table. The results show that, by using the in-place strategy, multi-version algorithms can now be fairly compared with other STM algorithms such as TL2, which was not possible when using the original Deuce framework.

A preliminary version of the work described in this paper was published in the Euro-Par 2012 conference [7].

ACKNOWLEDGEMENT

We would like to thank João Cachopo, of IST-UTL, for providing the hardware resources for the execution of the benchmarks reported in this paper.

References

1. Herlihy M, Luchangco V, Moir M. A flexible framework for implementing software transactional memory. *Proc. 21st conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM, 2006; 253–262, doi:http://doi.acm.org/10.1145/1167473.1167495.
2. Korland G, Shavit N, Felber P. Noninvasive concurrency with Java STM. *Proc. MultiProg 2010: Programmability Issues for Heterogeneous Multicores*, 2010.
3. Dice D, Shalev O, Shavit N. Transactional locking II. *Proc. 20th Int. Symp. on Distributed Computing*, *LNCS*, vol. 4167, Springer, 2006; 194–208, doi:http://dx.doi.org/10.1007/11864219_14.
4. Riegel T, Felber P, Fetzer C. A lazy snapshot algorithm with eager validation. *Proc. 20th Int. Symp. on Distributed Computing*, *LNCS*, vol. 4167, Springer, 2006; 284–298, doi:http://dx.doi.org/10.1007/11864219_20.
5. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.
6. Riegel T, Brum DBD. Making object-based STM practical in unmanaged environments. *Proc. of the 3rd Workshop on Transactional Computing*, 2008.
7. Dias RJ, Vale TM, Lourenço JM. Efficient support for in-place metadata in transactional memory. *Euro-Par 2012 Parallel Processing*, *Lecture Notes in Computer Science*, vol. 7484, Kaklamanis C, Papatheodorou T, Spirakis P (eds.). Springer Berlin / Heidelberg, 2012; 589–600.
8. Oracle. java.lang.instrument.Instrument. `http://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html` Nov 2012.
9. Binder W, Hulaas J, Moret P. Advanced Java bytecode instrumentation. *Proceedings of the International Symposium on Principles and Practice of Programming in Java (PPPJ)*, 2007; 135–144.
10. Cachopo J, Rito-Silva A. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.* Dec 2006; **63**:172–185, doi:10.1016/j.scico.2006.05.009.
11. Fernandes SM, Cachopo Ja. Lock-free and scalable multi-version software transactional memory. *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, ACM: New York, NY, USA, 2011; 179–188, doi:10.1145/1941553.1941579.
12. Blundell C, Lewis EC, Martin MMK. Deconstructing transactions: The subtleties of atomicity. *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*. 2005.
13. Cao Minh C, Chung J, Kozyrakis C, Olukotun K. STAMP: Stanford transactional applications for multi-processing. *IISWC '08: Proc. IEEE Int. Symp. on Workload Characterization*, 2008.
14. Guerraoui R, Kapalka M, Vitek J. Stmbench7: a benchmark for software transactional memory. *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, ACM: New York, NY, USA, 2007; 315–324, doi:10.1145/1272996.1273029.
15. Dragojević A, Guerraoui R, Kapalka M. Stretching transactional memory. *Proc. Int. Conf. on Programming Language Design and Implementation*, ACM, 2009; 155–165, doi:http://doi.acm.org/10.1145/1542476.1542494.
16. Perelman D, Byshevsky A, Litmanovich O, Keidar I. SMV: Selective multi-versioning STM. *Proc. 25th Int. Symp. on Distributed Computing*, *LNCS*, vol. 6950, Springer, 2011; 125–140.
17. Anjo I, Cachopo J. Lightweight transactional arrays for read-dominated workloads. *Proc. 11th Int. Conf. on Algorithms and Architectures for Parallel Processing*, Springer-Verlag: Berlin, Heidelberg, 2011; 1–13, doi: http://dl.acm.org/citation.cfm?id=2075462.2075464.
18. Afek Y, Korland G, Zilberstein A. Lowering STM overhead with static analysis. *Proc. 23rd Int. Workshop on Languages and Compilers for Parallel Computing*, 2010, doi:10.1109/IPDPS.2010.5470446.