Framework Support for the Efficient Implementation of Multi-version Algorithms

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

CITI — Universidade Nova de Lisboa, Quinta da Torre, 2829-516 Caparica, Portugal
{ricardo.dias,joao.lourenco}@fct.unl.pt,
t.vale@campus.fct.unl.pt

Abstract. Software Transactional Memory algorithms associate metadata with the memory locations accessed during a transaction's lifetime. This metadata may be stored in an external table and accessed by way of a function that maps the address of each memory location with the table entry that keeps its metadata (this is the out-place or external scheme); or alternatively may be stored adjacent to the associated memory cell by wrapping them together (the in-place scheme). In transactional memory multi-version algorithms, several versions of the same memory location may exist. The efficient implementation of these algorithms requires a one-to-one correspondence between each memory location and its list of past versions, which is stored as metadata. In this chapter we address the matter of the efficient implementation of multi-version algorithms in Java by proposing and evaluating a novel in-place metadata scheme for the Deuce framework. This new scheme is based in Java Bytecode transformation techniques and its use requires no changes to the application code. Experimentation indicates that multi-versioning STM algorithms implemented using our new in-place scheme are in average 6× faster than when implemented with the out-place scheme.

1 Introduction

Software Transactional Memory (STM) algorithms differ in the properties and in the guarantees they provide. Among other differences, one can refer distinct strategies used to read (visible or invisible) and update memory (direct or deferred), the consistency (opacity or snapshot isolation) and progress guarantees (solo, global and local progress), 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 [10] and Deuce [11], address the need of experimenting with new algorithms and their comparative evaluation by providing a single transactional interface over which the STM algorithms are built. However, the internal architecture each STM framework tends to favor the performance of some classes of STM algorithms and disfavor others. For instance, the Deuce framework stores the metadata in an external table and favors algorithms like TL2 [6] and LSA [14], which are resilient to the false sharing of transactional metadata (such as ownership records), and disfavor multi-version algorithms, which require unique metadata per memory location.

STM algorithms manage information per transaction (frequently referred to as the *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 keeps the information required to validate and commit the transaction, e.g., the read- and write-sets. The per memory location information, henceforth be referred as *metadata*, depends on the nature of the STM algorithm and may contain locks, timestamps, version lists. Metadata is stored either adjacent to each memory location (*in-place* scheme), or in an external table (*out-place* or *external* scheme). STM libraries for imperative languages, such as C, frequently use the out-place scheme, while those addressing object-oriented languages bias towards the in-place scheme.

The out-place scheme is implemented by using a table-like data structure that efficiently maps memory references to its metadata. Storing the metadata in such a preallocated 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 also 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 scheme is usually implemented using the *decorator* design pattern [8], by extending the functionality of an original class by wrapping it in a *decorator* class that contains the required metadata. This scheme implements a *one-to-one* relation between memory locations and metadata units, thus no false sharing occurs. It 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. Riegel et al. [15] 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 scheme. Implementing in Deuce an algorithm that requires a one-to-one relation between metadata and memory locations, such as a multi-version algorithm, requires the use of an external table to handles collisions, which significantly degrades the throughput of the algorithm.

In the remaining of this Chapter we present a novel approach to support the in-place metadata scheme that does not use the decorator pattern, and thoroughly evaluate its implementation in Deuce. This extension allows the efficient implementation of multi-version algorithms, which require a one-to-one relation between metadata and memory locations. The developed extension has the following properties:

Efficiency. The extension fully supports primitive types, even in transactional code. 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 we apply some light transformations to the non-transactional bytecode.

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 Deuce framework assumes a weak atomicity model, i.e., transactions are atomic only with respect to other transactions, and hence their execution may be interleaved with non-transactional code. Multi-version algorithms update objects (memory locations) by writing the new value to the object (memory cell) metadata (which contain the lists or past values), and therefore transactional accesses cannot see non-transactional updates, and vice-versa. We tackle this problem by proposing an algorithmic adaptation for multi-version algorithms that enables the support of a weak atomicity model for multi-version algorithms with meaningless impact in the overall performance.

This chapter follows with a description of the Deuce framework and its out-place scheme in Section 2. Section 3 describes properties of the in-place scheme, its implementation, and its limitations as an extension to Deuce. We present an evaluation of the extension's implementation using several metrics in Section 4. Section 5 describes the implementation of several state-of-the-art STM multi-version algorithms using our proposed extension. In Section 6 we show how to adapt the multi-version algorithms to support a weak-atomicity model. Finally, we present a comparison between different single- and multi-version algorithms using standard benchmarks in Section 7.

2 The Deuce Framework

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 which strategies they use to store the transactional metadata. Algorithms such as TL2 [6] or LSA [14] use an out-place scheme by resorting to a very fast hashing function and storing a single lock in each table entry. 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, false sharing may have some impact in the performance but does not affect the correctness. To implement multi-version algorithms with the out-place scheme, one has to manage collision lists in the table, which significantly degrades performance.

```java
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
}
```

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

To support the out-place scheme, 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 runtime passes the pair (O, f^o) 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.

3 Supporting the In-Place Scheme in Deuce

In our approach to extend Deuce to support the in-place scheme, 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.

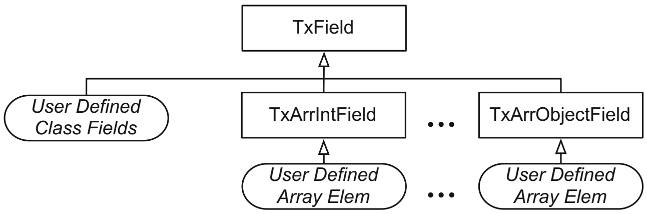

Fig. 2. Metadata classes hierarchy

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

Fig. 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
}
```

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

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 `*` ranges over the Java primitive types[1]. All the `TxArr*Field` classes extend from `TxField`, providing the STM algorithm with a simple and uniform interface for call-back functions.

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

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

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

Fig. 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;
}
```

Fig. 6. Example transformation of a class with the in-place scheme

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 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 the 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, this metadata object f^m is accessible by a single dereference operation from object O, which can be achieved by adding a new metadata field (of the corresponding 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 new 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 each 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 [1], 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. Each array cell contains a single value of a well defined type and no other information can be added to those cells. 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 also contain the additional information. This straight forward transformation has many implications in the application code, as 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.

We address this matter by 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-instrumented legacy code. With this approach, the original array can still be retrieved with a minimal overhead by dereferencing twice the sibling `TxArr*Field` array. 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.

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, the method signature does not provide information enough to know if there is the need to revert to primitive arrays. Take, for example, the

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

⟹

```
class D {
    TxArrIntField[] a;
    TxField a_metadata;
}
class TxArrIntField {
  int[] array; //base array
  int index;
}
```

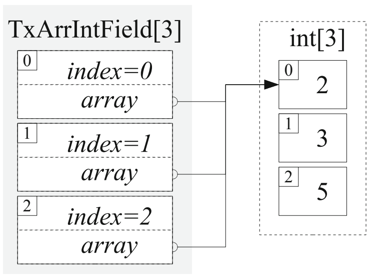

Fig. 7. Memory structure of a `TxArrIntField` array

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

⟹

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

Fig. 8. Example transformation of array access in the in-place scheme

`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*[2] 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 non-instrumented method (e.g., from the 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-instrumented 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.

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

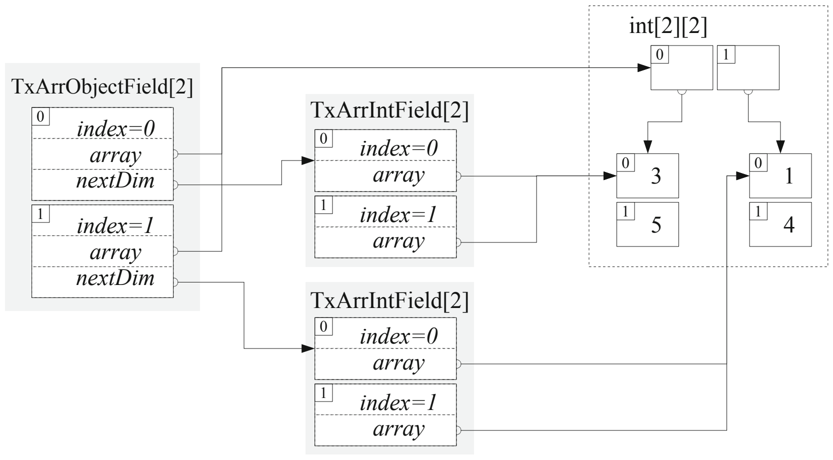

Fig. 9. Memory structure of a multi-dimensional `TxArrIntField` array

The limitations of our support for in-place metadata for single- and multi-dimensional arrays in Deuce are discussed with further detail in [5].

4 Evaluation of the In-Place Scheme

The implementation of the proposed Deuce extension, described in the previous sections, introduces more complexity to the transactional processing when comparing with the original Deuce implementation. This complexity, in the form of additional memory operations and allocations, may slowdown the performance in some cases. In our first step to assess the extension implementation performance, we evaluate the overhead of the new implementation by comparing it with the original Deuce implementation.

In a second step we evaluate the performance speedup of using our extension to implement a multi-version STM algorithm, against an implementation of the same algorithm using the original Deuce interface. We chose a well known multi-version STM algorithm, JVSTM, described in [3], and implemented two versions of the algorithm, one using the original Deuce interface and an out-place scheme (referred to as `jvstm-outplace`), and another using our new interface and extension supporting an in-place scheme (referred to as `jvstm-inplace`).

Both the overhead and speedup evaluations are preformed using several micro- and macro-benchmarks. Micro-benchmarks are composed by the Linked List, Red-Black Tree, and Skip-List data structures. Macro-benchmarks are composed by the STAMP [4] benchmark suite and the STMBench7 [9] benchmark. 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 6272 16-Core processors @ 2.1 GHz with 8×2 MB of L2 cache, 16 MB of L3 cache, and 64 GB of RAM, running Debian Linux 3.2.41 x86_64, and Java 1.7.0_21.

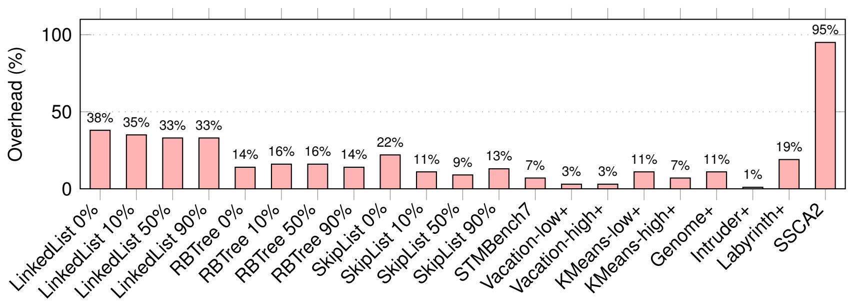

Fig. 10. Performance overhead measure of the usage of metadata objects relative to out-place TL2

In the following sections we describe in detail, and present the results, of the overhead evaluation as well as the speedup evaluation.

4.1 Overhead Evaluation

To evaluate the overhead introduced by the management of the metadata object fields and sibling arrays as required by our extension, we measured and compared the performance of two very similar implementations of the TL2 algorithm, which only differ in which API (`context` interface) is used to implement the algorithm: one uses the original API as provided by the Deuce distribution, and another (named `tl2-overhead`) uses the new interface of our modified Deuce (as described in Figure 4 in page 170). The change of API requires the additional management of metadata objects (allocation, and array manipulation), and two additional dereferences on the metadata object to obtain the field's object reference and the field offset, for each read and write operation.

Figure 10 depicts the average overhead introduced by the `tl2-overhead` implementation with respect to the original Deuce TL2 implementation. The Figure reports on several benchmarks, with each benchmark aggregating results from executions ranging form 1 to 64 threads. The overhead of the additional management of metadata objects and sibling arrays is in average about 20%. The benchmarks that use metadata arrays (SkipList, Kmeans, Genome, Labyrinth, SSCA2) have in general a higher overhead than those that only use metadata objects for class fields (RBTree, STMBench7, Vacation, Intruder). The micro-benchmarks (Linked List, Red-Black Tree and SkipList) were all tested in four scenarios: with a read-only workload (0% of updates), and read-write workloads with 10%, 50%, and 90% of updates. These micro-benchmarks are composed of small transactions which only perform read and write accesses to shared memory, and thus, the overhead is more visible. The higher overhead in the LinkedList micro-benchmark is due to the 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 STAMP benchmarks, show relatively low overhead,except for the SSCA2+ benchmark. These benchmarks have medium sized transactions which perform some computations with the data read from the shared memory. The SSCA2+ benchmark only preforms read and write operations over arrays, and may be considered the worst-case scenario for our extension. The STMBench7 benchmark was executed with a read-dominant workload, without long-traversals, and with structural modifications activated. In this benchmarks transactions are computationally much heavier, which hides the small overhead introduced by the management of in-place metadata.

From this results we can conclude that out new in-place scheme introduces a small overhead due to the management of in-place metadata, but it also enables the efficient implementation of single- and multi-version STM algorithms in a single STM framework. In the next sections we show the comparison of the performance of the same multi-version algorithm implemented using the original Deuce framework and our extension.

4.2 Implementing a Multi-versioning Algorithm: 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. To commit 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.

4.2.1 Out-Place Scheme

To implement JVSTM algorithm in the original Deuce framework, which only supports the out-place scheme, the *vboxes* must be stored in an external table[3]. 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

[3] We opted to use a concurrent hash table from the `java.util.concurrent` package.

```java
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
}
```

Fig. 11. `VBox` in-place implementation

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.

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 the same JVSTM algorithm using the in-place scheme (`jvstm-inplace`).

4.2.2 In-Place Scheme

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 11. 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 comparison evaluation shows that the direct access to the version list allowed by the in-place scheme will greatly benefit the performance of the algorithm. We present the comparison results in the next section by presenting the speedup of the in-place version with respect to the out-place version.

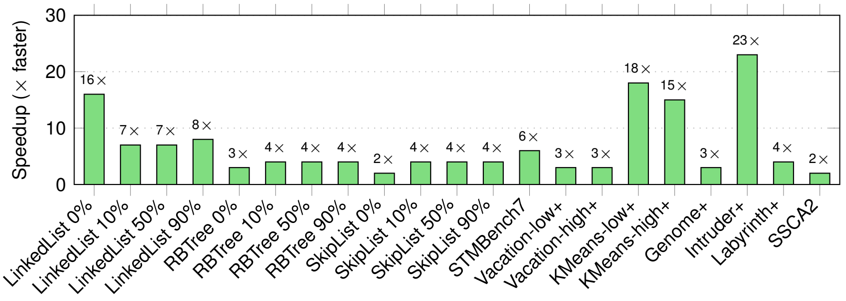

Fig. 12. In-place over Out-place scheme speedup: the case of JVSTM

4.3 Speedup Evaluation

From the evaluation of the in-place management overhead, we concluded that this scheme 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 [3], one in the original Deuce using the native out-place scheme (`jvstm-outplace`), and another in the extended Deuce using our in-place scheme (`jvstm-inplace`), as described in the previous Section.

Figure 12 depicts the average speedup of our two implementations of the JVSTM algorithm: one In-Place (`jvstm-inplace`) and another Out-Place (`jvstm-outplace`). We used the same set of benchmarks and configuration that was used for the overhead evaluation in Section 4.1. In The in-place version of the JVSTM algorithm is in average 7 times faster than its dual out-place version. The speedup observed for the micro-benchmarks, where transactions are small and contention is low, shows that the multi-versioning algorithms greatly benefit from our in-place support. In the case of the STAMP benchmarks, where transactions are submitted to workloads of intensive contention, the in-place version is much faster than the out-place approach as it avoids completely the use of a shared external table, which becomes a serious bottleneck in the presence of high contention. In the special case of KMeans and Intruder benchmarks, the overhead of managing a shared external table drastically increases the probability of transaction aborts as depicted in Figure 13, which in turn makes the transactional throughput to decrease. The STMBench7 macro-benchmark has many long-running transactions and the overall throughput for both algorithms is relatively low. Even so, the in-place algorithm is in average 6× faster.

5 State-of-the-Art Multi-version Algorithm's Implementations

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. With the support for in-place metadata we can implement and compare the state-of-the-art multi-version algorithms, both between themselves and with single-version algorithms.

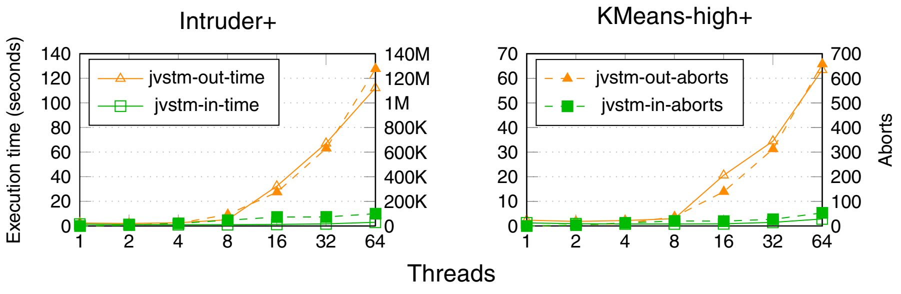

Fig. 13. Performance and transaction aborts of JVSTM-Inplace/Outplace for the Intruder and KMeans benchmarks

To support this fact, we implemented two state-of-the-art multi-version algorithms: SMV [12] and JVSTM-LockFree [7]. These algorithms are significantly different, although both are MV-*permissive* [13]. They differ on the progress guarantees, e.g., JVSTM-LockFree implements a commit algorithm that is lock-free, while SMV uses write-set locking, and also differ on the technique used to garbage collect unnecessary versions, where JVSTM-LockFree uses a custom parallel garbage collector, while SMV resorts to the JVM garbage collector by using weak-references. In the following sections we describe the implementation details of each of the above algorithms.

5.1 SMV – Selective Multi-Versioning STM

The SMV algorithm described in [12] is an MV-*permissive* multi-version algorithm, which uses the JVM garbage collector to automatically collect unreachable versions. The implementation of this algorithm in our extension of Deuce was based on the original source code released by the authors[4]. The original algorithm is object-based, opposite to Deuce, and our extension, which only supports word-based STMs, and hence we adapted the SMV algorithm to work as a word-base STM.

The transactional metadata required by SMV can be depicted in Figure 14. This is a direct adaptation of the `SMVAdapterLight` class provided by the original source code. Also, we used the same source code that implements the behavior of read- and update-transactions with minimal changes. We did this by implementing our extension's interface `ContextMetadata` as an adapter of the original source code, each transactional operation (read, write, commit, abort) is forward to the original implementation.

The change from an object-based to a word-based approach only required minimal changes on the read and write procedures. In the case of a read operation, instead of returning an object, is returned a field's value. And in the case of a write operation, instead of cloning the object to be written and storing in the transaction's write-set, the tentative value of a field is stored in the write-set. The overall adaptation of the original source code to our framework was very easy and fast, which proves the flexibility of our support for implementing different STM algorithms.

[4] `http://tx.technion.ac.il/ dima39/sourcecode/ SMVLib-29-06-11.zip`

```
public class SMVObjAdapter extends TxField {
    public volatile Object latest;
    public int creatorTxnId;
    public final AtomicInteger version = new AtomicInteger(1);
    public volatile WeakReference<VersionHolder> prev =
                new WeakReference<VersionHolder>(null);
    // ... public methods
}
```

Fig. 14. SMV transactional metadata class

```
public class VBoxAdapter extends TxField {
    protected VBox<Object> vbox;
    // ... public methods
}
```

Fig. 15. JVSTM-LockFree transactional metadata class

5.2 JVSTM Lock Free

The JVSTM-LockFree [7] is an adaptation of the original JVSTM algorithm [3], which enhances the commit procedure using a lock-free algorithm, instead of using a global lock, and also improves the garbage collector algorithm by the use of a parallel collecting approach. Once again, we based our implementation in the original source code[5].

We created a metadata object containing a reference to a vbox, as implemented originally by the JVSTM-LockFree algorithm. We show the object metadata implementation in Figure 15. The context class was implemented as an adapter to the original implementation of the read-only and update transactions. Actually, we used the JVSTM-LockFree implementation as an external library (JAR file), and the Deuce context class only forwards the transactional calls to the external library. This approach was possible because there was no need to make any changes to the JVSTM-LockFree algorithm, for it to work in our framework extension.

6 Supporting Efficient Non-transactional code

Multi-version algorithms read and write the data values from and into 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 [2]. Deuce does not provide a strong atomicity model as memory accesses done outside of transactions are not instrumented, 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. This problem can be circumvented by rewriting the existing benchmarks to wrap all accesses to shared

[5] `https://github.com/inesc-id-esw/jvstm`

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 to the multi-version algorithms that makes them compatible with the *weak atomicity* model.

When using a weak atomicity model with a multi-version scheme, 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 solution is to store the value of the latest version in the object's field instead of in the node 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, otherwise it reads the value from the version node.

The main issue with this approach is how to guarantee the atomicity of the commit of 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 be atomic with respect to the other concurrent transactions. Our solution is to create a temporary new version with an infinite timestamp, making it unreachable for other concurrent transactions, until we update the value and then change the timestamp to its proper value. The algorithmic adaptation that we propose is not intended to support a workload of intertwined non-transactional and transactional accesses, but rather a phased workload where non-transactional code does not execute concurrently with transactional code. Many of the transactional benchmarks we used exhibit such a phased workload, because the data structures are initialized in the program startup using non-transactional code. After this initialization, the transactional code can now operate over the data previously installed by non-transactional code. After the transactional processing, non-transactional code may also post-process the data, such as in a case of a validation procedure.

6.1 Read Access Adaptation

In a multi-version scheme, read-only transactions always search for a correct version to return its value. Each version container holds the timestamp (or version number) and the respective value. When the transaction finds the correct version, it returns the value contained in the version.

To support non-transactional accesses mixed with a multi-version scheme, the latest value of an object's field is stored in-place, and therefore the head version might not have the correct value because of a previous non-transactional update. The read procedure of a multi-version transaction must be adapted to reflect the new location of the latest value. When a transaction queries for a version, and receives the head version, corresponding to the latest value, it has to return the value directly from the object's field. The pseudo-code of this adaptation is presented below, where the additional operations are denoted in underline.

```
1. $\underline{val := \mathsf{read}()}$
2. $ver := \mathsf{find\_version}()$
3. $\mathsf{return} \begin{cases} \underline{val} & \underline{\text{if } \mathsf{is\_head\_version}(ver)} \\ ver.val & \text{otherwise} \end{cases}$
```

The $\mathsf{read}()$ function returns the value from the object's field, the $\mathsf{find\_version}$ function retrieves the corresponding version according to the transaction timestamp, and the $\mathsf{is\_head\_version}$ function asserts if version *ver* is the head version. This small change introduces the additional shared memory access performed in step 1. The correctness of this adaptation can only be assessed with the explanation of the commit adaptation, which guarantees that whenever the $\mathsf{is\_head\_version}$ function returns true the value *val* is correct.

6.2 Commit Adaptation

The commit operation is typical composed by a validation phase and write-back phase. In the write-back phase, for each new value present in the write-set, a new version is created and is stored as the head version. The write-back phase must be atomic, and this can be achieved using a global lock (JVSTM), a write-set entry locking (SMV), or even a lock-free algorithm (JVSTM-LockFree).

Our adaptation only makes changes to the write-back phase. In each iteration of the write-back phase, a new version is installed as the head version of the version list associated with the object's field being written. The version contains the commit timestamp, which defines the commit ordering, and the new value. Additionally, to support the weak-atomicity model, we also need to write the new value directly to the object's field. The problem that arises with this additional operation is that concurrent transactions need to see the update on the version list, and the update of the object's value as a single operation. The key idea to solve this problem is to create a version with a temporary infinite timestamp, which will prevent concurrent transactions from accessing the head version, and consequently the object's field value.

Below we present the pseudo-code of the adaptation to the commit of a new version, where t_c is the timestamp of the transaction that is performing the commit, t_∞ is the highest timestamp, *val* is the value to be written, and ver_h is the pointer to the head version. For the sake of simplicity, we assume that these steps execute in mutual exclusion with respect to other concurrent commits (in Section 6.2.3 we explain how to apply these steps to a lock-free context as in the JVSTM-LockFree algorithm).

```
1. $\underline{ver_h.value := \mathsf{read}()}$
2. $ver_n := \mathsf{create\_version}(new\_val, \underline{t_\infty}, ver_h)$
3. $ver_h := ver_n$
4. $\underline{\mathsf{write}(new\_val)}$
5. $\underline{ver_h.timestamp := t_c}$
```

Once again, the additional changes are denoted in underline. The first step is to update the value of the head version with the current value of the object's field. This update is safe because until this point transactions that retrieve the head version read the value directly from the object's field, as described in the previous section. Then we create a new version with an infinite timestamp and the new value to be written in the object's field, and the pointer to the current head version. In the third step, we make the new version ver_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 object's field value in the fourth step

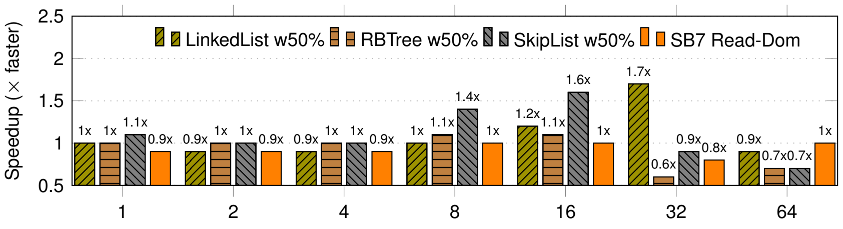

Fig. 16. Performance comparison between original JVSTM and adapted JVSTM

because no concurrent transaction gets the head version (the head version still has an infinite timestamp up to this point). In the last step we change the timestamp of the current head version to its proper value making accessible to concurrent transactions.

The adaptation of the commit operation introduces three new shared memory accesses, where two of them are write accesses. Thus, this adaptation is expected to slightly lower the throughput of the multi-version algorithm. We applied this adaptation to the multi-version algorithms that we described previously, and compared the performance of both versions of each. In the next section we report the experience of adapting each algorithm.

6.2.1 JVSTM

The JVSTM algorithms preform the commit operation in mutual exclusion with other concurrent committing transactions. The adaptation of these algorithms to support a weak-atomicity model is straightforward. The changes that we presented in the previous section to modify the read and commit operation can be applied directly to both implementations. Moreover, the Deuce framework already provides the memory value when a read access is issued (see Figure 4 in page 170), which simplifies the first step of the read procedure described in Section 6.1.

Figure 16 depicts the performance comparison between the original and adapted versions of JVSTM. The comparison is done by showing the relative performance of the adapted version over the original version. The adapted version of JVSTM shows a performance very similar to the original versions. Sometimes, the adapted version can even outperform the original version. This is due to the specificity of the Deuce framework that already provides the memory value for each read access callback. In the case of the adapted version, most of the times that value is used, opposed to the original version where the value is always obtained by dereferencing a version container.

6.2.2 SMV

The SMV algorithm defines a different memory layout for the version list. In SMV, the value of the latest version is stored outside of the version list, which reassembles our adaptation proposal of storing the latest value directly on the memory location. To apply the support for a weak-atomicity model, we simply moved the value of the latest version from an auxiliary variable (used in SMV original implementation) directly to the associated memory location.

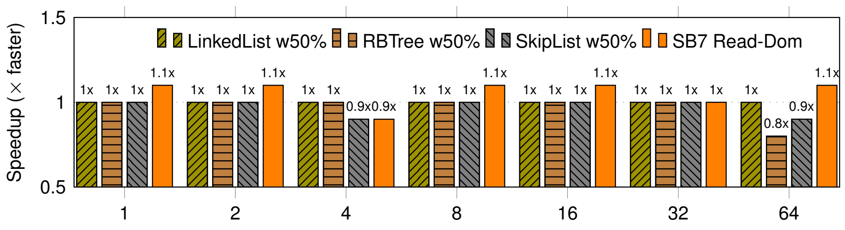

Fig. 17. Performance comparison between original SMV and adapted SMV

This modification has consequences in the commit operation, which must also be adapted to atomically update the latest version information and the memory location value. The first step in the SMV commit operation is to move the latest value and timestamp to a newly created version container and add it to the head of the version list. We change this step by using the latest value stored in memory. In the last step of the SMV commit operation the variable containing the latest value is updated with the new tentative value. We changed this step by writing the tentative value directly to memory. The changes made to the SMV algorithm are minimal and thus we expect that the performance differences between the two versions to be also minimal. The results depicted in Figure 17 confirm our expectations, showing minimal differences between the original version and adapted version.

6.2.3 JVSTM-LockFree

The JVSTM-LockFree implements a lock free commit operation. The assumption to apply the adaptation for the commit procedure, presented in Section 6.2, is that the commit should be done in mutual exclusion. This assumption is true for the previous algorithms but not for the JVSTM-LockFree. In this algorithm, the commit of a single version can be done by more than one thread at the same time by resorting to atomic primitives such as compare-and-swap.

The adaptation of the read procedure is straightforward as in the JVSTM algorithm. The adaptation of the commit procedure is rather complex and requires additional atomic operations to ensure the correctness of the algorithm. Figure 18 depicts a simplified version of the original commit. The method `commit` preforms a compare-and-swap to install the new version. Other threads may be executing the same method for the same *vbox*, but only one of them will install the new version. Further details on how the JVSTM-LockFree commit algorithm works can be found in [7].

Figure 19 depicts the adapted version of the JVSTM-LockFree commit algorithm to support a weak-atomicity model. The new algorithm has roughly three times more operations than the original one. We explain this adapted version by describing how each step of the adaptation described in Section 6.2 is related to the code listed in the Figure 19.

The first step $ver_h.value := \mathsf{read}()$ is preformed by lines 5 and 7-9. The update of the head version's value (line 8) is done inside a conditional statement because other concurrent thread may had already preformed the same update. The creation of a new version in the second step $ver_n := \mathsf{create\_version}(new\_val, t_\infty, ver_h)$ is preformed in

```
public void commit(Object newValue, int txNumber) {
    Version currHead = this.head;
    Version existingVersion = currHead.getVersion(txNumber);
    if (existingBody.version < txNumber) {
        Version newVer = new Version(newValue, txNumber, currHead);
        compare_and_swap(this.head, currHead, newVer);
    }
}
```

Fig. 18. JVSTM-LockFree original commit operation

```
public void commit(Object newValue, int txNumber) {
    Version currHead = this.head;
    Version existingVersion = currHead.getVersion(txNumber);

    Object latest = read(memory_location);
    if (existingVersion == currHead
            && existingVersion.version < txNumber) {
        if (this.head == existingVersion) {
            currHead.value = latest;
        }
        Version newVer = new Version(newValue, Integer.MAX_VALUE,
                currHead);
        if (compare_and_swap(this.head, currHead, newVer)) {
            existingVersion = newVer
        } else {
            existingVersion = this.head;
            Version tmpVer = existingVersion.getVersion(txNumber);
            if (tmpVer.version == txNumber) {
                existingVersion = tmpVer;
            }
        }
        if (existingVersion.version == Integer.MAX_VALUE) {
            compare_and_swap(memory_location, latest, newValue);
        }
        existingVersion.version = txNumber;
    }
    else {
        if (existingVersion.version < txNumber) {
            existingVersion = currHead;
            if (existingVersion.version == Integer.MAX_VALUE)
                compare_and_swap(memory_location, latest, newValue);
            existingVersion.version = txNumber;
        }
    }
}
```

Fig. 19. JVSTM-LockFree adapted commit operation

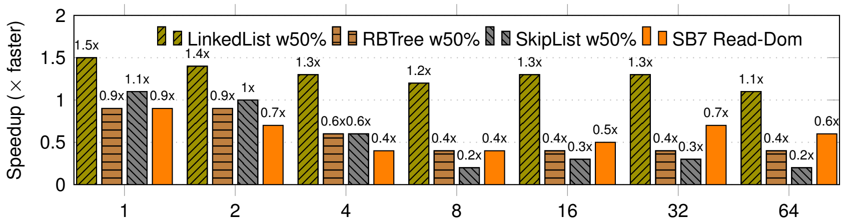

Fig. 20. Performance comparison between original JVSTM-LockFree and adapted JVSTM-LockFree

line 10. The publication of the new version in the third step $ver_h := ver_n$ is preformed in lines 11-19. In this step we preform a compare-and-swap, as in the original algorithm, to publicize the new version, but if other concurrent thread already publicize the new version, then we need to get a pointer to the new version. This is done in lines 14 to 18. Using this pointer we can preform the final fourth and fifth steps $\mathsf{write}(\mathit{new\_val})$ and $ver_h.timestamp := t_c$, which are done in lines 20-23. The writing of the new value directly to memory (line 21) is done using a compare-and-swap atomic operation to prevent lost updates. The update of the version number (line 23) is safe because we always have a pointer to the correct version container. These last two steps are also preformed in lines 28-31, in the case when a thread attempting to commit finds out, in line 6, that other concurrent thread already publicized the new version, and therefore it helps finishing the commit. Another source of overhead is caused by a limitation of the compare-and-swap operation, which can only be preformed for reference and integer types. Thus, for other primitive type such as **`float`**, or **`byte`**, the compare-and-swap operations preformed in lines 21 and 29, must be substituted by some mutual exclusion block. Fortunately the use of compare-and-swap non-supported types in the benchmarks is rare.

Figure 20 presents the results of comparing the adapted version over the original version of JVSTM-LockFree. In the case of the LinkedList micro-benchmark, the transactions generate small write-sets (the add and remove operations only write to a single object), and typically the commit rate is low due to the long duration of the lookup of a node, which is linear with the size of the list. As so, the adapted version outperforms the original version, due to the read accesses that use value directly from memory and are immediately provided by the Deuce framework. In the case of the SkipList and RBTree micro-benchmarks, the adapted commit overhead is more notorious when the contention increases with the number of threads. These benchmarks generate a high rate of commit operations, although still with small write-sets per transaction. In the STMBench7 benchmark, known to generate very large read- and write-sets, the adapted version can only achieve half the performance of the original version. The results confirm our performance expectations, and also confirm that the overhead introduced by adapting a multi-version algorithm to support a weak-atomicity model is almost nil for algorithms that preform the commit of versions in mutual exclusion, and has a considerable cost otherwise.

7 Performance Comparison of STM Algorithms

In this chapter we presented an extension of the Deuce framework to support the efficient implementation of STM algorithms that require a one-to-one relation between memory locations and transactional metadata, being multi-version algorithms an instance of this class of algorithms. We evaluated the extension considering the implications in both performance and memory consumption. The results were very satisfactory and thus we implemented two state-of-the-art multi-version algorithms (SMV and JVSTM-LockFree).

Given this support for very different classes of STM algorithms, we may now aim at a fair comparison of their performance, i.e., compare the algorithms implemented in the same framework and with the same benchmarks. In this section we show the direct comparison between several out-place and in-place STM algorithms. The list of STM algorithms chosen for comparison are TL2, JVSTM, JVSTM-LockFree, and SMV. In the case of TL2 we use two versions: the out-place version (TL2-Outplace) which is distributed with Deuce, and an in-place version (TL2-Inplace) which we implemented in our extension. The in-place version moves the locks from the external lock table to the transactional metadata, and completely avoids the false-sharing on locks. In the case of multi-version algorithms our measurements were conducted under two settings. The first setup consisted on executing the (unmodified) benchmarks combined with the weak-atomicity-adapted multi-version algorithms. In the second setup, we executed a modified version of the micro-benchmarks and STMBench7 combined with the original multi-version algorithms that do not support weak-atomicity. In the comparison results, we will only use the best of the results of the original and the adapted versions of each multi-version algorithm. As in the extension evaluation, the benchmarks were executed on a computer with four AMD Opteron 6272 16-Core processors @ 2.1 GHz with 8×2 MB of L2 cache, 16 MB of L3 cache, and 64 GB of RAM, running Debian Linux 3.2.41 x86_64, and Java 1.7.0_21.

Figure 21 shows the results of the execution of the micro-benchmarks Linked List, Red-Black Tree, and Skip List. The Linked List benchmark is characterized by transactions with large read-sets and by a high abort rate. In this benchmark the algorithms do not scale well with the increase in the number of threads. The single-version algorithms TL2-Outplace and TL2-Inplace exhibit better performance. These algorithms have very efficient implementations and the read accesses are very lightweight. Additionally, in the case of read-only transactions, each read access is checked for consistency but the transaction can safely commit without further verification. To support multiple versions per memory location, the multi-version algorithms add a high number of extra computations when reading a value from a memory location, with the benefit of avoiding spurious transaction aborts and hence avoid the re-execution of transactions. Although, in the micro-benchmarks this possible benefit is not observed. In the Red-Black Tree and Skip List benchmarks, transactions are very small and fast, and have a low conflict probability, except in the Red-Black Tree when tree rotations are preformed. These benchmarks hide even more the advantages of multi-version algorithms when compared with single-version algorithms. The poor performance of SMV when compared to the

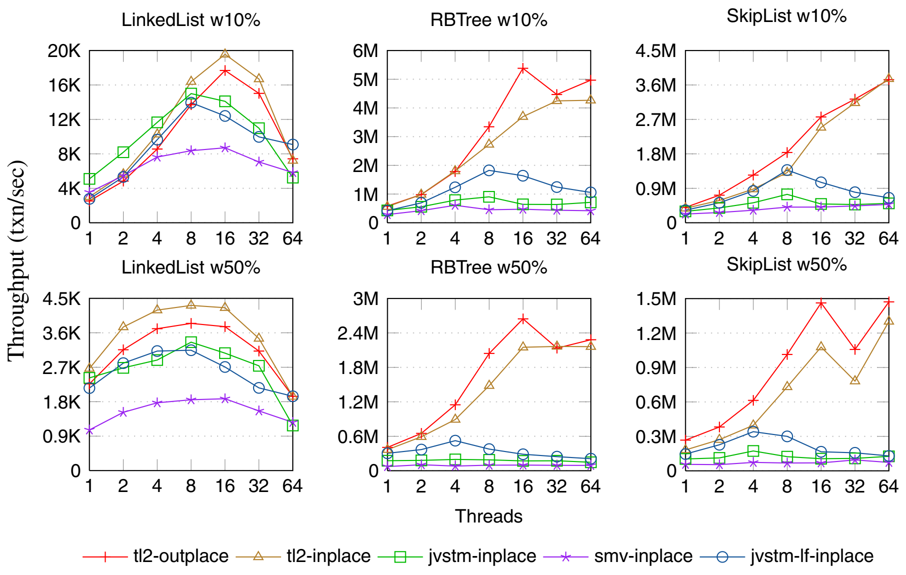

Fig. 21. Micro-benchmarks comparison

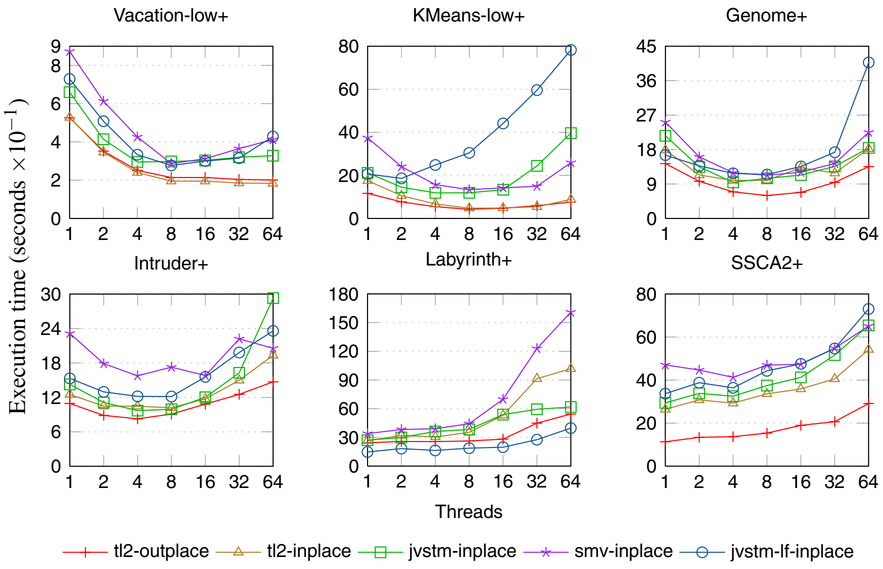

Fig. 22. STAMP benchmarks comparison

other multi-version algorithms is due to the strain imposed on the Java garbage collector: the micro-benchmarks generate millions of transactions per second, generating a lot of activity of the Java garbage collector.

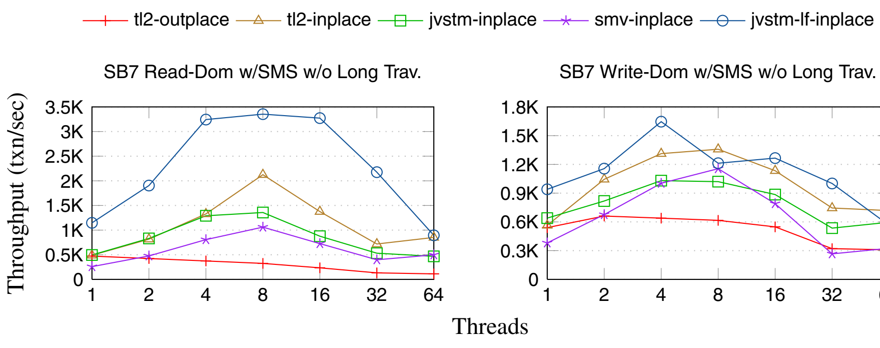

Fig. 23. STMBench7 comparison

The comparison results for the STAMP benchmarking suite are depicted in Figure 22. In these results the y-axis represents execution time and therefore lower values are better. The benchmarks in this suite exhibit very different workloads, some of them even generate such high contention that hinders the scaling for all of the tested algorithms. The benchmarks KMeans, Genome, and Intruder, exposes the corner cases of the *adapted* JVSTM-LockFree algorithm, hence its performance is strongly penalized. We believe that the original JVSTM-LockFree algorithm would perform much better than the adapted version in these particular benchmarks. The TL2 based algorithms overall exhibit a very good performance. In the Labyrinth benchmark the multi-version algorithm JVSTM-LockFree presents a very good result. This algorithm has a low abort rate when compared with the other algorithms, which allows it to not waist so much work in transaction restarts. In the SSCA2 benchmark all the in-place algorithms suffer from the high overhead of transactional metadata management shown in Figure 10 of Section 4.1.

In Figure 23 we show the results for the STMBench7 benchmarks. This benchmark generate CPU-intensive transactions with large read-sets and write-sets. This benchmarks allows to exploit the benefits of multi-version algorithms which can avoid spurious aborts and thus achieve better performance than single-version algorithms. The JVSTM-Lockfree algorithm achieves a good performance, higher than the remaining algorithms, confirming the advantages of using an MV-*permissive* algorithm in this kind of workload. In this benchmark, there is a significant performance difference between the out-place and in-place versions of TL2 algorithm. The out-place version does not even scale with the number of threads. The reason of this behavior may be due to cache locality issues. The in-place version is much more cache-friendly than the out-place version. The in-place version has a high probability of having the metadata in the same cache line as the memory location. This does not happen in the out-place version, and in the special case of STMBench7, where transactions perform a large number of reads and writes, the out-place version must read many entries from the external lock table, which may not fit in the cache and requiring much more page transfers from main memory to the cache. In the write-dominated workload of STMBench7, all algorithms have similar performance with the exception of TL2-Outplace. Although almost all transactions are read-write, the multi-version algorithms can still compete with the single-version TL2-Inplace algorithm, and JVSTM-LockFree almost always exhibit the best performance.

8 Concluding Remarks

In this chapter we presented an extension of Deuce that provides a performance-wise support for implementing STM multi-version algorithms. This is achieved by a transformation process of the program Java 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 the proposed system by measuring the overhead introduced by the new in-place scheme with respect to the original Deuce implementation. Although we can observe a light slowdown caused by the in-place metadata management, the slowdown is quickly absorbed by the performance gains achieved when using the in-place scheme to store the STM algorithms metadata.

The new efficient implementation support for STM multi-version algorithms allowed to implement two state-of-the-art multi-version algorithms SMV and JVSTM-LockFree. Moreover, we present the first performance comparison between the two.

Finally, we proposed an algorithmic adaptation for multi-version algorithms to support the weak-atomicity model as provided in the Deuce framework. We reported the experience of adapting several state-of-the-art multi-version algorithms and evaluate their performance. In general, multi-version algorithms can be adapted to support the weak-atomicity model without a performance penalty, except the case of the algorithms that implement a lock-free commit operation.

Acknowledgments. 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 grants SFRH/BD/41765/2007 and SFRH/BD/84497/2012.

References

1. Bloch, J.: Effective Java, 2nd edn. Addison-Wesley (2008)
2. Blundell, C., Lewis, E.C., Martin, M.M.K.: Deconstructing transactions: The subtleties of atomicity. In: Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking, (WDDD) (2005)
3. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Sci. Comput. Program. 63(2), 172–185 (2006)
4. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: 4th IEEE International Symposium on Workload Characterization (IISWC). IEEE (2008)
5. Dias, R.J., Vale, T.M., Lourenço, J.M.: Efficient support for in-place metadata in java software transactional memory. Concurrency and Computation: Practice and Experience 25(17), 2394–2411 (2013)

6. Dice, D., Shalev, O., Shavit, N.N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
7. Fernandes, S.M., Cachopo, J.A.: Lock-free and scalable multi-version software transactional memory. In: 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 179–188. ACM (2011)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (1994)
9. Guerraoui, R., Kapalka, M., Vitek, J.: STMBench7: A benchmark for software transactional memory. In: 2nd EuroSys Conference (EuroSys), pp. 315–324. ACM (2007)
10. Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. In: 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 253–262. ACM (2006)
11. Korland, G., Shavit, N., Felber, P.: Deuce: Noninvasive software transactional memory. Transactions on HiPEAC 5(2) (2010)
12. Perelman, D., Byshevsky, A., Litmanovich, O., Keidar, I.: SMV: Selective multi-versioning STM. In: Peleg, D. (ed.) DISC 2011. LNCS, vol. 6950, pp. 125–140. Springer, Heidelberg (2011)
13. Perelman, D., Fan, R., Keidar, I.: On maintaining multiple versions in STM. In: 29th Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 16–25. ACM (2010)
14. Riegel, T., Fetzer, C., Felber, P.: Snapshot isolation for software transactional memory. In: 1st ACM SIGPLAN Workshop on Transactional Computing (TRANSACT) (2006)
15. Riegel, T., Brum, D.B.D.: Making object-based STM practical in unmanaged environments. In: 3rd ACM SIGPLAN Workshop on Transactional Computing (TRANSACT) (2008)