TribuSTM Instrumentation Rules

Tiago Vale Ricardo J. Dias
CITI — Departamento de Informática
Universidade Nova de Lisboa
t.vale@fct.unl.pt

January 30, 2012

Contents

1 Introduction

The main objective of TribuSTM is to extend the existing STM framework DeuceSTM by supporting different kinds of STM algorithms that are hard to implement in the current framework. The kind of algorithms that will benefit from the TribuSTM extension are those that typically associate transactional metadata (e.g., lock, timestamp, etc...) to each memory location by using a wrap class pattern.

In DeuceSTM framework, the only way to associate transactional metadata is to use a data structure that maps a memory location identifier to the respective metadata.

TribuSTM supports direct access to transactional metadata without the need of a data structure indirection, thus allowing a more suitable implementation of STM algorithms.

Throughout this document, everytime we feel it is needed, we will establish a mapping between the conceptual Java code presented and the actual generated bytecode. That mapping will always be of the format *<listing>* *<dot>* *<line interval>*. For example, we refer to the *lines 40 through 45 in listing 10* as **10.40-45**.

2 Metadata: the `TxField` hierarchy

To associate transactional metadata (e.g., lock, timestamp, etc...) to each object's field, a class hierarchy, rooted in `TxField`, was created which developers are supposed to extend.

The `TxField` class holds all the information used in [?] to uniquely identify the associated field: the object's reference to which it belongs and its offset.

Listing 1: `TxField` class.

```
1 class TxField {
2   Object ref;
3   final long address;
4
5   TxField(Object ref, long address) {
6     this.ref = ref;
7     this.address = address;
8   }
9 }
```

Listing 2: `TxIntField` class.

```
1 class TxIntField extends TxField {
2   TxIntField(Object ref, long address) {
3     super(ref, address);
4   }
5
6   final int read() {
7     return UnsafeHolder.getUnsafe().getInt(ref,
          address);
8   }
9
10  final void write(int value) {
11    UnsafeHolder.getUnsafe().putInt(ref,
          address, value);
12  }
13 }
```

For convenience, there are specialized classes for each Java type: `TxBooleanField`, `TxByteField`, `TxCharField`, `TxDoubleField`, `TxFloatField`, `TxIntField`, `TxLongField`, `TxObjectField` and

TxShortField. Using sun.misc.Unsafe, the methods read and write allow you to directly get and set the value of its associated field, respectively.

In listings ?? and ?? we can see real Java code for the TxField and TxIntField classes.

2.1 Arrays, a special case

Listing 3: TxArrIntField class.

```
1  public class TxArrIntField extends TxField {
2    private static long __ADDRESS__;
3    static {
4      try {
5        __ADDRESS__ = AddressUtil.getAddress(TxArrIntField.class
6            .getDeclaredField("nonTxValue"));
7      } catch (SecurityException e) {
8      } catch (NoSuchFieldException e) {
9      }
10   }
11
12   public int nonTxValue;
13
14   public TxArrIntField(Object ref, long address) {
15     super(ref, address);
16   }
17
18   public TxArrIntField() {
19     super(null, __ADDRESS__);
20     this.ref = this;
21   }
22
23   protected final int read() {
24     return nonTxValue;
25   }
26
27   protected final void write(int value) {
28     nonTxValue = value;
29   }
30 }
```

Arrays are treated slightly differently. Instead of being able to reference the associated field, each array element's metadata class will actually *store* its element value, as depicted in figure ??. Built upon the TxField class of listing ??, listing ?? shows the content of the TxArrIntField, the metadata class for an array of ints.

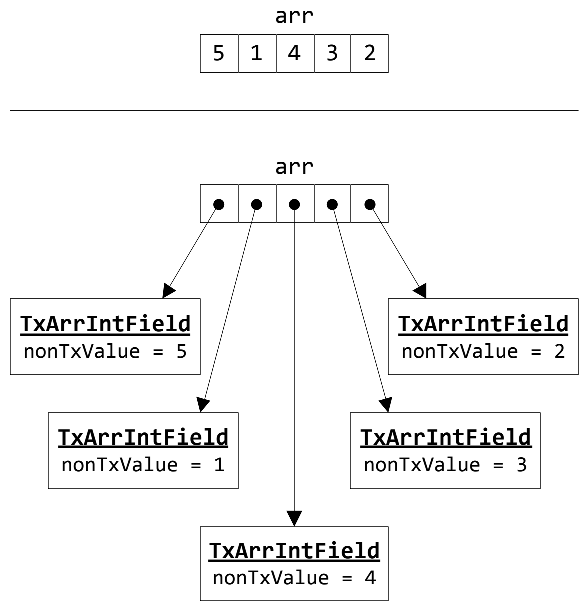

Figure 1: New kind of arrays.

Therefore, we have the `TxArrBooleanField`, `TxArrByteField`, `TxArrCharField`, `TxArrDoubleField`, `TxArrFloatField`, `TxArrIntField`, `TxArrLongField`, `TxArrObjectField` and `TxArrShortField` classes, each holding a value of the specific type.

This means that all arrays need to be replaced by it's transactional counterpart. For example, an `int[]` field must be replaced by `TxArrIntField[]`. This approach, not without severe implications which are described in the following sections, also works for multidimensional arrays.

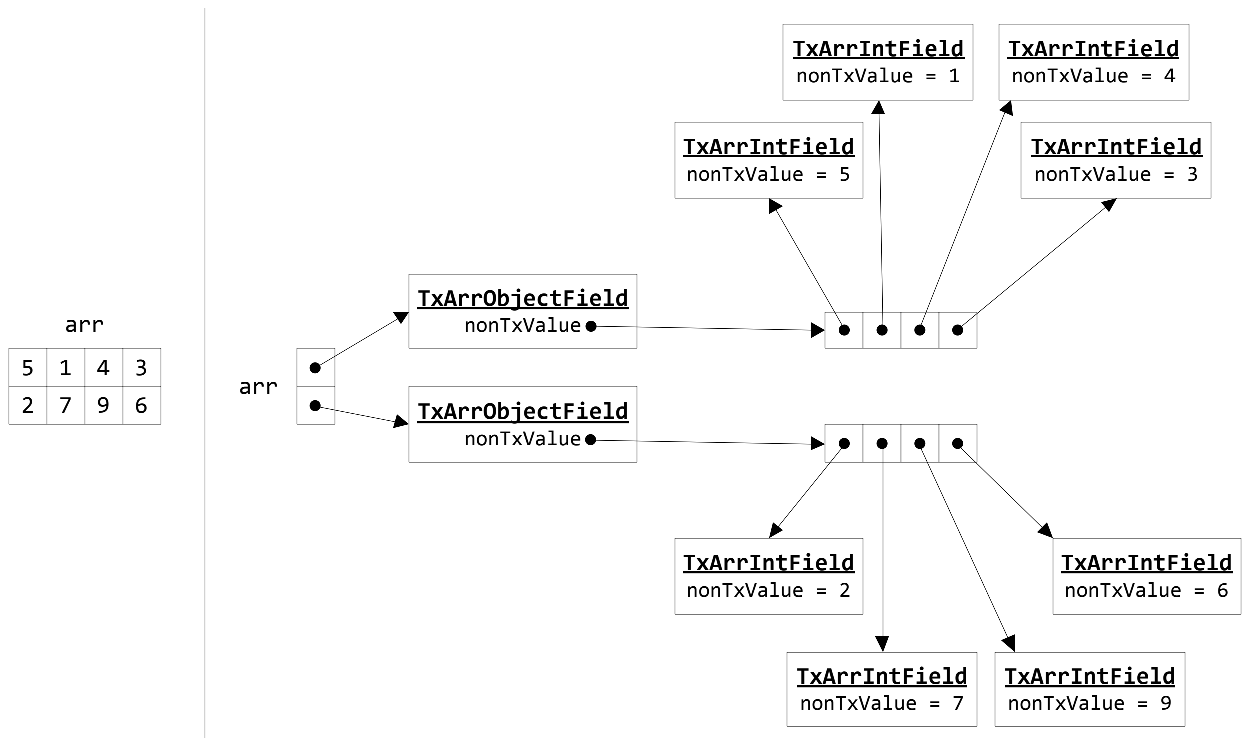

Figure 2: Multidimensional arrays.

Multidimensional arrays are converted into nested unidimensional `TxArrObjectField` arrays, where only the "last" dimension is an array of the corresponding `TxArrField` type. Consider the variable `arr` in figure **??** as previously of type `int[2][4]` (on the left), and now of type `TxArrObjectField[2]` where each element is a `TxArrIntField[4]` object (on the right).

3 Fields

Using the class hierarchy presented in section **??**, we want to have metadata at field level granularity. Simply building upon [?], for each field in a class we insert a new one (behind the scenes) which holds the metadata object for its corresponding "brother" field. Figure **??** represents such concept. The dashed-line will be described in the following paragraph.

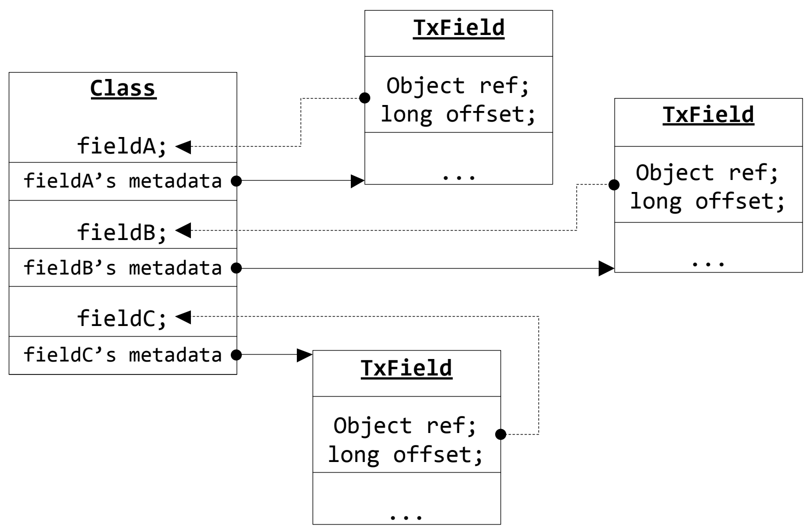

Figure 3: Field's metadata abstraction.

Regarding the Java implementation details, which can be seen in figure ??, after the class is loaded its static descriptor is created. This descriptor holds everything that is shared between class instances, therefore contains the class' static fields and their metadata pointers.

A metadata pointer is a constant field (`final public`) suffixed with `__ADDRESS__` that references the `TxField` metadata object associated with the corresponding field. This metadata is tightly coupled to the specific instance of the class – there will be as many instances of each metadata object as there are instances of the class – unless the field is `static`, in which case only one exists, in the descriptor previously presented. The metadata maintains a low-level reference to the field it's associated with. This connection is displayed with a dashed-line instead to raise awareness. This reference is based upon the base address of the field's object and the field's offset from it. Using the `sun.misc.Unsafe` class, the field can be manipulated directly from the metadata class.

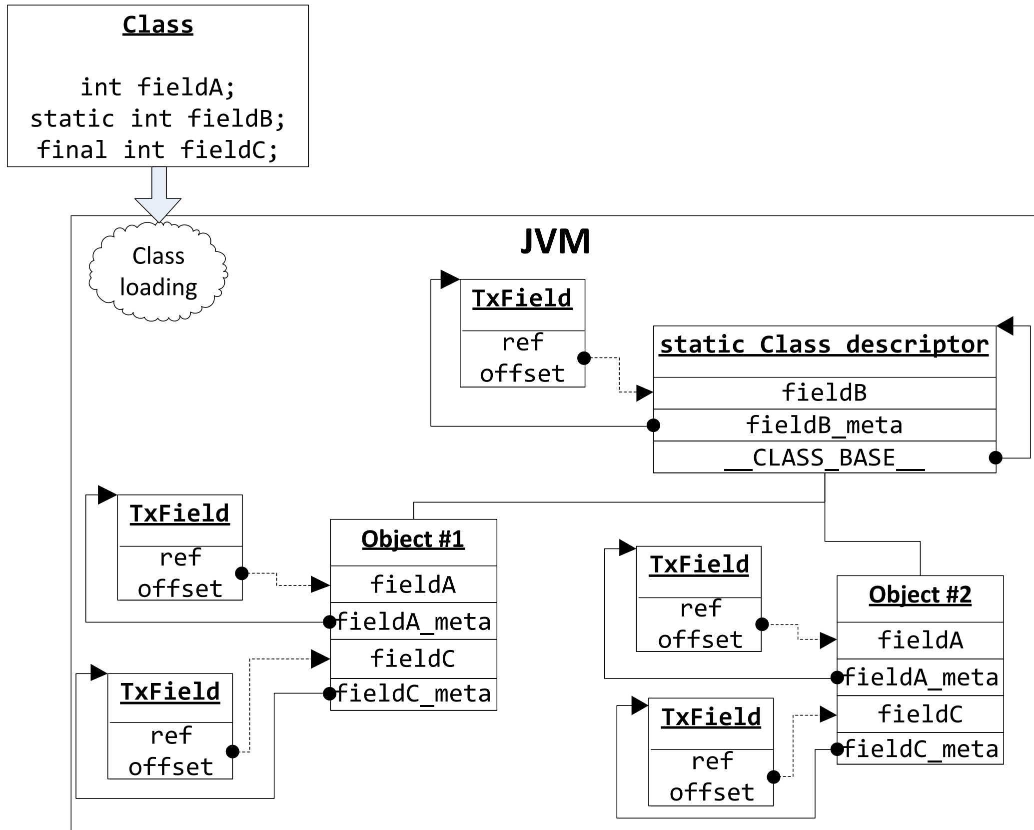

Figure 4: Java implementation details of the metadata abstraction. The `_meta` fields are equivalent to the `__ADDRESS__` fields.

Having seen what is the idea (figure ??) and how it is accomplished (figure ??), let's dissect the following real code instrumentation example. In listings ?? and ??, to the left we have the original class and to the right the instrumented result.

Listing 4: Example class.

```
1 class Example {
2   int intFieldA;
3   static int intFieldB;
4   final int intFieldC;
5   (...)
6 }
```

Listing 5: Metadata related added fields.

```
1 class Example {
2   int intFieldA;
3   final TxIntField intFieldA__ADDRESS__;
4   static final long __STATIC__intFieldA__ADDRESS__;
5
6   static int intFieldB;
7   static final TxIntField intFieldB__ADDRESS__;
8   static final long __STATIC__intFieldB__ADDRESS__;
9
10  final int intFieldC;
11  final TxIntField intFieldC__ADDRESS__;
12  static final long __STATIC__intFieldC__ADDRESS__ = -1L;
13
14  static final Object __CLASS_BASE__;
15  (...)
16 }
```

As we can see, the chosen example contains 3 fields. Their type is not really important, the modifiers are. For each field 2 new fields are injected, a `final __ADDRESS__` suffixed `TxField` and `static final __STATIC__` prefixed `long`. The `__ADDRESS__` suffixed field contains a reference to the associated meta-

data TxField object, while the __STATIC__ prefixed long's value is the associated field's offset. Therefore, fieldA__ADDRESS__ is the metadata object associated with fieldA and __STATIC__fieldA__ADDRESS__ its offset.

Although in section ?? it is said that the offset has migrated into the metadata object, a static value is still kept for performance reasons, because its calculation is a heavy operation and the resultant value is constant. So you can interpret the __STATIC__ field as a sort of *cache* which will be used when instantiating new metadata for new objects.

If the original field is final, like intFieldC, the value of __STATIC__intFieldC__ADDRESS__ will be -1L and intFieldC__ADDRESS__ will not be initialized, and therefore will be null. These are our means of knowing if a field is final in runtime.

The ref field in TxField metadata associated with static fields, such as fieldB, cannot reference any concrete object instance. Instead it references the static __CLASS_BASE__ field, which points to the *static class descriptor* where all static class fields reside. All other instance-specific metadata will reference its concrete owner object.

3.1 Arrays

As said in section ??, besides the instrumentation already mentioned in section ??, the actual type of array variables must be changed.

Listing 6: Example class.

```
1 class ArrayExample {
2   int[] intArrField;
3   int[][] intMatrxField;
4   (...)
5 }
```

Listing 7: Modified array fields.

```
1 class ArrayExample {
2   TxArrIntField[] intArrField;
3   final TxObjectField intArrField__ADDRESS__;
4   static final long __STATIC__intArrField__ADDRESS__;
5
6   TxArrObjectField[] intMatrxField;
7   final TxObjectField intMatrxField__ADDRESS__;
8   static final long __STATIC__intMatrxField__ADDRESS__;
9
10  static final Object __CLASS_BASE__;
11  (...)
12 }
```

The example in listings ?? and ?? is very similar to the previous. intMatrxField's type will be changed to TxArrObjectField[] and each element will be of type TxArrIntField[].

4 Methods

In the previous sections we presented the metadata class hierarchy, and the transformations done regarding class fields. Some of these fields are static and need to be initialized.

In Java, the special method <clinit> is called when a class is loaded. Its purpose is to initialize everything that is not instance-dependent, but shared among all of them.

4.1 <clinit>

The static fields added by the transformation in listing ?? need to be initialized. This is accomplished by injecting code in the class initialization method, <clinit>.

Listing 8: Example class.

```
1 class Example {
2   int intFieldA;
3   static int intFieldB;
4   final int intFieldC;
5   (...)
6 }
```

Listing 9: static fields' initialization.

```
1 long addr;
2 Field f;
3
4 __CLASS_BASE__ = AddressUtil.staticFieldBase(Example.class,
     "intFieldB");
5
6 f = Example.class.getDeclaredField("intFieldA");
7 __STATIC_intFieldA__ADDRESS__ = AddressUtil.getAddress(f);
8
9 f = Example.class.getDeclaredField("intFieldB");
10 addr = AddressUtil.getAddress(f);
11 intFieldB__ADDRESS__ = new TxIntField(__CLASS_BASE__, addr);
```

In listings ?? and ?? we have the initializations performed by the `<clinit>` method. To the left we have the example class, and to the right the injected code.

The `__CLASS_BASE__` field is initialized using the `AddressUtil` utility class. The `staticFieldBase` method receives the class and the name of a `static` field, and returns the address of the *static class descriptor* using `sun.misc.Unsafe`.

Then we initialize `intFieldA`'s offset *cache*, `__STATIC__intFieldA__ADDRESS__`, using the standard reflection method `getDeclaredField` to obtain the field data structure. This data structure is then passed to the `AddressUtil` helper class that calls `sun.misc.Unsafe` in order to obtain the field's offset.

`intFieldB`'s process is analogous. But since the field is `static`, we have to instantiate the metadata as said in section ??. Therefore, we need the reference to the *static class descriptor* and the field's address, `__CLASS_BASE__` and `addr`, respectively.

In listing ?? we have the actual generated bytecode for this initializations.

Listing 10: static metadata initialization generated bytecode.

```
static {};
  Code:
   0:  ldc Example
   2:  ldc intFieldB : String
   4:  invokestatic AddressUtil.staticFieldBase(Class, String) : Object
   7:  putstatic Example.__CLASS_BASE__ : Object
   10: ldc Example
   12: ldc intFieldA : String
   14: invokevirtual Class.getDeclaredField(String) : Field
   17: invokestatic AddressUtil.getAddress(Field) : long
   20: putstatic Example.__STATIC__intFieldA__ADDRESS__ : long
   23: new TxIntField
   26: dup
   27: getstatic Example.__CLASS_BASE__ : Object
   30: ldc Example
   32: ldc intFieldB : String
   34: invokevirtual Class.getDeclaredField(String) : Field
   37: invokestatic AddressUtil.getAddress(Field) : long
   40: invokespecial TxIntField."<init>"(Object, long) : void
   43: putstatic Example.intFieldB__ADDRESS__ : TxIntField
   46: return
```

Just as `<clinit>` must initialize the `static` fields, the instance fields also need to be initialized. These initializations are injected at the end of the constructor method, `<init>`.

4.2 <init>

In Java, the class self-titled constructor methods are renamed to `<init>` as the source code is compiled to bytecode. The compilation also assures that there exists at least one `<init>` method, the default parameterless construtor, if the programmer doesn't specify one. That said we can safely append the instance metadata initialization to the `<init>` method.

Listing 11: Example class.

```
1 class Example {
2   int intFieldA;
3   static int intFieldB;
4   final int intFieldC;
5   (...)
6 }
```

Listing 12: Instance fields' initialization.

```
1 public Example() {
2   (...)
3   intFieldA__ADDRESS__ = new TxIntField(this,
        Example.__STATIC__intFieldA__ADDRESS__);
4 }
```

In examples **??** and **??** we have the Java equivalent of what will be injected in the same example class. We already saw in listing **??** how the `__ADDRESS__` field is instantiated. In this case, the `TxField`'s `ref` object is `this` because it is an instance-dependent field, and the offset is the `__STATIC__` *cache* field initialized in `<clinit>`.

The constructors are also duplicated as described in section **??**. Listing **??** shows the actual appended bytecode.

Listing 13: Generated bytecode for instance metadata initialization.

```
public Example();
  Code:
   (...)
   9:  aload_0
   10: new TxIntField
   13: dup
   14: aload_0
   15: getstatic Example.__STATIC__intFieldA__ADDRESS__ : long
   18: invokespecial TxIntField."<init>"(Object, long) : void
   21: putfield intFieldA__ADDRESS__ : TxIntField
   24: return
```

This two subsections cover the initialization of all metadata objects, either `static` or instance-dependent. We will now see how is a non-*@Atomic* annotated method modified and duplicated to obtain the transactional effect.

4.3 Non-*@Atomic*

A non-*@Atomic* method is a regular Java method; it does not trigger events upon memory access and thus does not impose any transactional overhead. Still, calls to this method can be done within a transactional context. It is then desirable to have a duplicate method which triggers those events.

This duplicate method will have an extra parameter, an `IContext` object, which represents the current transaction. It will "wrap" all class members access with their corresponding *hook functions* in the `ContextDelegator` class and replace all function calls with their respective duplicate calls. Ideally, we would like to know in compile-time if a field is `final`, because those fields do not need to be wrapped.

Listing 14: Non-*@Atomic* original.

```
1  class Example {
2    int intFieldA;
3    static int intFieldB;
4    final int intFieldC;
5
6    public void addToA(int val) {
7      intFieldA = intFieldA + val;
8    }
9
10   public void incA() {
11     addToA(1);
12   }
13 }
```

Listing 15: Non-*@Atomic* duplicated.

```
1  public void addToA(int val, IContext ctx) {
2    int tmp;
3    int readA;
4
5    if (intFieldA__ADDRESS__ != null) {
6      ContextDelegator.beforeReadAccess(
7        intFieldA__ADDRESS__, ctx);
8      readA = ContextDelegator.onReadAccess(
9        intFieldA, intFieldA__ADDRESS__, ctx);
10   } else {
11     readA = intFieldA;
12   }
13
14   tmp = readA + val;
15
16   if (intFieldA__ADDRESS__ != null)
17     ContextDelegator.onWriteAccess(tmp,
           intFieldA__ADDRESS__, ctx);
18   else
19     intFieldA = tmp;
20 }
21
22 public void incA(IContext ctx) {
23   addToA(1, ctx);
24 }
```

The example in listings ?? and ?? show the generated duplicate methods for both a method with and without parameters, `addToA(int)` and `incA()`, respectively. Both are transactional methods, and as such they receive an extra parameter of the type `IContext`, as previously said.

`incA()` is a very simple method, all it does is call `addToA(1)` and as such, the only thing to do in `incA(IContext)` is to call the duplicated `addToA(int, IContext)` instead. During the transactional environment, replacing all calls with their transactional duplicates will keep the call flow within the transaction boundaries and semantics.

Regarding `addToA(int)`, there are few more complex changes to make. It's one-line body reads `intFieldA`'s and `val`'s value, and then modifies `intFieldA`'s. Therefore we will need to have the *STM* algorithm intercept both the read and write accesses to `intFieldA` if needed. This is accomplished by replacing the `intFieldA` read access on line 6 of `addToA(int)` with the guarding `if` on lines 5-12 in `addToA(int, IContext)`. It checks if the field is `final` by verifying the nullity of the metadata object, and if not calls the `beforeReadAccess` and `onReadAccess` hook methods of `ContextDelegator`. The write access substitution is analogous, calling `onWriteAccess` if the field is not `final`.

Listing ?? shows the actual generated bytecode, derived from the concept in listing ??.

Please refer to the line mapping syntax described in section ?? for the following line mappings. They will be appearing again in subsequent sections.

In `addToA(int, IContext)`, lines ??.5-9 map to lines ??.3-6 and ??.15-33, while ??.11 maps to ??.9. ??.14 maps to ??.36-37, and ??.16-17 maps to ??.41-44 and ??.53-59. Finally ??.18 maps to ??.47.

Listing 16: Generated bytecode for duplicate methods.

```
public void addToA(int, IContext);
  Code:
    0:  aload_0
    1:  aload_0
    2:  dup
    3:  getfield intFieldA__ADDRESS__ : TxIntField
    6:  ifnonnull 15
    9:  getfield intFieldA : int
    12: goto 36
    15: dup
    16: getfield intFieldA__ADDRESS__ : TxIntField
    19: aload_2
    20: invokestatic ContextDelegator.beforeReadAccess:(TxField, IContext) : void
    23: dup
    24: getfield intFieldA : int
```

```
    27: dup_x1
    28: pop
    29: getfield intFieldA__ADDRESS__ : TxIntField
    32: aload_2
    33: invokestatic ContextDelegator.onReadAccess:(int, TxField, IContext) : int
    36: iload_1
    37: iadd
    38: dup_x1
    39: pop
    40: dup_x1
    41: getfield intFieldA__ADDRESS__ : TxIntField
    44: ifnonnull 53
    47: putfield intFieldA : int
    50: goto 62
    53: dup_x1
    54: pop
    55: getfield intFieldA__ADDRESS__ : TxIntField
    58: aload_2
    59: invokestatic ContextDelegator.onWriteAccess:(int, TxField, IContext) : void
    62: return

public void incA(IContext);
  Code:
    0:  aload_0
    1:  iconst_1
    2:  aload_1
    3:  invokevirtual addToA(int, IContext) : void
    6:  return
```

Sections ?? and ?? have already emphasized that arrays will need special care throughout all transformations. In the following section we will describe the modifications required by all methods which depend on arrays.

4.4 Arrays

As described in section ??, array types are replaced. This requires substantial modifications in methods which either receive or return arrays:

1. Any method that receives one or more array parameters, or returns, must have its signature modified to reflect the new type;
2. The `getfield` and `putfield` accesses to, and `checkcast` with, an array field must be updated;
3. The `aload` and `astore` instruction families need to be replaced by `aaloading` the `TxArrField` at the specified index, and a `getfield` or `putfield` to its `nonTxValue` field, respectively;
4. All `newarray` and `anewarray` instructions must be replaced by `anewarray` of the corresponding `TxArrField`, and subsequent initialization of all elements.
5. The `multianewarray` instructions require more work, as each of the nested unidimensional array elements must be initialized. This is delegated to a generated `final static private` method.

We will now see some examples, and mind that some names might be abbreviated for the sake of readability.

Listing 17: Example class.

```
1 class ArrayExample {
2   int[] intArrField;
3   int[][] intMatrxField;
4
5   public int[][] getMatrx() {
6     return intMatrxField;
7   }
8 }
```

Listing 18: Array return modification.

```
1 public TxArrObjectField[] getMatrx() {
2   return intMatrxField;
3 }
```

Listings ?? and ?? exemplify item 1 of the above list, regarding a method that returns an array. In this case, the method returned an `int[][]` and therefore must be changed to `TxArrObjectField[]`.

Listing ?? shows the generated bytecode. Even though the returned value is `intMatrxField`, it's safe, wouldn't even compile otherwise, to change the return type from `int[][]` to `TxArrObjectField[]` because `intMatrxField`'s type has been changed accordingly as described in section ??.

Listing 19: Generated bytecode for the method returning an array.

```
public TxArrObjectField[] getMatrx();
  Code:
    0:  aload_0
    1:  getfield intMatrxField : TxArrObjectField[]
    4:  areturn
```

Still on item 1, in the example on listings ?? and ?? we have a method that received an array as a parameter and had to be updated accordingly, in this case a transformation of parameter `arr`'s type from `int[]` to `TxArrIntField[]`. Once again, this transformation is not only safe but necessary, as all array creations and initializations are replaced by their transactional counterpart as seen in listings ?? and ??. Also worth referencing is the call of a Java plaform method, `System.arraycopy`. These methods do not understand our transactional array class hierarchy and therefore will need special care, which will later be discussed in section ??. In this particular case, no problem occurs because of method overloading.

The generated bytecode regarding the array parameter can be seen in listing ??.

Listing 20: Example class.

```
1 class ArrayExample {
2   int[] intArrField;
3   int[][] intMatrxField;
4
5   public void copyToArr(int[] arr) {
6     System.arraycopy(arr, 0, intArrField,
          0, arr.length);
7   }
8 }
```

Listing 21: Array parameter modification.

```
1 public void copyToArr(TxArrIntField[] arr)
2 {
3   System.arraycopy(arr, 0, intArrField, 0,
        arr.length);
4 }
```

Listing 22: Generated bytecode for the method with array parameter.

```
public void copyToArr(TxArrIntField[]);
  Code:
    0:  aload_1
    1:  iconst_0
    2:  aload_0
    3:  getfield intArrField : TxArrIntField[]
    6:  iconst_0
    7:  aload_1
    8:  arraylength
    9:  invokestatic System.arraycopy(Object, int, Object, int, int) : void
    12: return
```

In list item number 3, and listings ?? and ??, we address the array access problematic. Since transactional array elements are now objects (see sections ?? and ??), all accesses to array positions must be changed. Arrays are accessed with the `aload` and `astore` instruction families for reading and writing, respectively (`iaload`, `baload`, `iastore`, `bastore`, etc.), and as such these need to be replaced with `getfield` and `putfield` to the `nonTxValue` field of the `TxArrField` specific object hold by the array – see section ??.

Once again, listing ?? shows the generated bytecode.

Listing 23: Example class.

```
1 class ArrayExample {
2   int[] intArrField;
3   int[][] intMatrxField;
4
5   public void addToArr(int i, int val) {
6     intArrField[i] = intArrField[i] + val;
7   }
8 }
```

Listing 24: Modified array accesses.

```
1 public void addToArr(int i, int val) {
2   intArrField[i].nonTxValue =
        intArrField[i].nonTxValue + val;
3 }
```

Listing 25: Generated bytecode for the modified array accesses.

```
public void addToArr(int, int);
  Code:
    0:  aload_0
    1:  getfield intArrField : TxArrIntField[]
    4:  iload_1
    5:  aload_0
    6:  getfield intArrField : TxArrIntField[]
    9:  iload_1
    10: aaload
    11: checkcast TxArrIntField
    14: getfield TxArrIntField.nonTxValue : int
    17: iload_2
    18: iadd
    19: dup_x2
    20: pop
    21: aaload
    22: checkcast TxArrIntField
    25: swap
    26: putfield TxArrIntField.nonTxValue : int
    29: return
```

Next, in listings **??** and **??**, we see the instrumentation necessary to deal with array initializations. Reminding that with arrays metadata and actual data are embedded, the initializations have to the replaced by `for` cycles that initialize each and every position of the array. For each array, as many as its dimensions of chained `for`s will be needed, so one `for` for `intArrField` and two for `intMatrxField`. For each type of multiple dimension arrays an helper method is created, because we cannot access the size of each dimension in the stack. As such, we create a method that will consume *number of dimensions* `int` elements from the stack, allowing us to know the dimensions and initialize the matrix. An example of such is `TxArrIntField_multiarray(int, int, IContext)` which can create and initialize any two-dimensional array of integers.

Listing 26: Original array methods.

```
1 class ArrayExample {
2   int[] intArrField;
3   int[][] intMatrxField;
4
5   public ArrayExample() {
6     intArrField = new int[10];
7     intMatrxField = new int[10][10];
8   }
9 }
```

Listing 27: Modified array methods.

```
1  class ArrayExample {
2    TxArrIntField[] intArrField;
3    TxArrObjectField[] intMatrxField;
4
5    public ArrayExample() {
6      intArrField = new TxArrIntField[10];
7      for (int i = 0; i < 10; i++)
8        intArrField[i] = new TxArrIntField();
9
10     intMatrxField = TxArrIntField_multiarray(10,
         10, null);
11   }
12
13   private static TxArrObjectField[]
       TxArrIntField_multiarray(int d1, int d2,
       IContext ctx) {
14     TxArrObjectField[] tmp0 = new
         TxArrObjectField[d1];
15     for (int i=0; i < d1; i++) {
16       tmp0[i] = new TxArrObjectField();
17       TxArrIntField[] tmp1 = new
           TxArrIntField[d2];
18       for (int j=0; j < d2; j++) {
19         tmp1[j] = new TxArrIntField();
20       }
21       tmp0[i].nonTxValue = tmp1;
22     }
23
24     return tmp0;
25   }
26 }
```

Listing ?? shows the actual generated bytecode regarding what was just discussed in listing ??. Lines ??.6-8 of `ArrayExample()` map to ??.4-39, and line ??.10 to ??.42-51.

As for `TxArrIntField_multiarray(int, int, IContext)`, lines ??.14-16 and ??.21 map to ??.0-22 and ??.58, while ??.17-20 map to ??.23-45.

Listing 28: Generated bytecode for the modified array methods.

```
public ArrayExample();
  Code:
    0:  aload_0
    1:  invokespecial Object."<init>"() : void
    4:  aload_0
    5:  bipush 10
    7:  anewarray TxArrIntField
    10: dup
    11: dup
    12: iconst_0
    13: dup_x1
    14: swap
    15: arraylength
    16: if_icmpge 38
    19: dup_x1
    20: new TxArrIntField
    23: dup
    24: invokespecial TxArrIntField."<init>"() : void
    27: aastore
    28: swap
    29: dup_x1
    30: dup_x1
    31: swap
    32: iconst_1
    33: iadd
    34: dup_x1
    35: goto 14
    38: pop2
```

```
    39: putfield intArrField : TxArrIntField[]
    42: aload_0
    43: bipush 10
    45: bipush 10
    47: aconst_null
    48: invokestatic ArrayExample.TxArrIntField_multiarray(int, int, IContext) :
        TxArrObjectField[]
    51: putfield intMatrxField : TxArrObjectField[]
    54: (...) // Metadata initialization.

private static final TxArrObjectField[] TxArrIntField_multiarray(int, int, IContext);
  Code:
    0:  iload_0
    1:  anewarray TxArrObjectField
    4:  dup
    5:  dup
    6:  iconst_0
    7:  dup_x1
    8:  swap
    9:  arraylength
    10: if_icmpge 71
    13: dup_x1
    14: new TxArrObjectField
    17: dup_x2
    18: dup
    19: invokespecial TxArrObjectField."<init>"() : void
    22: aastore
    23: iload_1
    24: anewarray TxArrIntField
    27: dup
    28: dup
    29: iconst_0
    30: dup_x1
    31: swap
    32: arraylength
    33: if_icmpge 57
    36: dup_x1
    37: new TxArrIntField
    40: dup_x2
    41: dup
    42: invokespecial TxArrIntField."<init>"() : void
    45: aastore
    46: pop
    47: swap
    48: dup_x1
    49: dup_x1
    50: swap
    51: iconst_1
    52: iadd
    53: dup_x1
    54: goto 31
    57: pop2
    58: putfield TxArrObjectField.nonTxValue : Object
    61: swap
    62: dup_x1
    63: dup_x1
    64: swap
    65: iconst_1
    66: iadd
    67: dup_x1
    68: goto 8
    71: pop2
    72: areturn
```

So far we have seen what is needed to be done regarding arrays in our own classes. But sometimes – or actually, most of the time – we rely on methods and objects from the Java platform. These do not understand our kind of transactional arrays which have values and metadata embedded. Also, due to

licensing restrictions, these cannot be instrumented and changed accordingly. So, how do we cope the calls to Java platform methods with our arrays? That is the question answered in the next section.

4.4.1 Java Platform methods

We still need to modify the calls to methods from Java Platform classes [?], and these cannot be instrumented. To overcome this, we generate a `final static private` method that will wrap the call.

Its responsibility is to make a "standard" array copy of the `TxArray` being used, use it as an argument for the wrapped method call, and finally copy the values back into the `TxArray`.

One final caution: the `public static void main(String[])` entry point must not have its parameter type modified. The conversion of `String[]` to `TxArrObjectField[]` must be injected in the beginning of the method's code instead.

Listing 29: Example class.

```
1 class ArrayExample {
2   int[] intArrField;
3   int[][] intMatrxField;
4
5   public int existsInArr(int val) {
6     return java.util.Arrays.binarySearch(
7       intArrField, val);
8   }
9 }
```

Listing 30: Java method wrapped call.

```
1 public int existsInArr(int val) {
2   return java_util_Arrays_binarySearch(
3     intArrField, val, null);
4 }
5
6 private static final int
      java_util_Arrays_binarySearch(
7   TxArrIntField arr, int val,
8   IContext ctx)
9 {
10   int tmp[] = new int[arr.length];
11   for (int i = 0; i < arr.length; i++)
12     tmp[i] = arr[i].nonTxValue;
13
14   int ret =
        java.util.Arrays.binarySearch(tmp,
        val);
15
16   for (int i = 0; i < arr.length; i++)
17     arr[i].nonTxValue = tmp[i];
18
19   return ret;
20 }
```

The example in listings ?? and ?? intend to search for an element using the `binarySearch(int[], int)` method from `java.util.Array`. That method can only work with an array of primitive `int`s, therefore a conversion from `TxArrIntField[]` to `int[]` is necessary. For that purpose, the `private static final` generated method, named after the full name of the Java platform method, creates a temporary array, copies all elements primitive value to it, calls the Java method with the temporary array and copies the values again back to our `TxArray`.

Listing ?? shows the lengthy generated bytecode for this procedure.

Lines ??.10-12 in `java_util_Arrays_binarySearch(TxArrIntField, int, IContext)` map to ??.0-23, while ??.16-17 maps to ??.44-75.

Listing 31: Generated bytecode for the Java platform method call.

```
public int existsInArr(int);
  Code:
   0:  aload_0
   1:  getfield intArrField : TxArrIntField[]
   4:  iload_1
   5:  aconst_null
   6:  invokestatic java_util_Arrays_binarySearch(TxArrIntField[], int, IContext) : int
   9:  ireturn

private static final int java_util_Arrays_binarySearch(TxArrIntField[], int, IContext);
  Code:
   0:  aload_0
   1:  dup
   2:  arraylength
   3:  newarray int
```

```
5:  swap
6:  dup_x1
7:  swap
8:  dup_x2
9:  dup_x1
10: iconst_0
11: dup_x1
12: swap
13: arraylength
14: if_icmpge 36
17: dup_x2
18: dup_x1
19: aaload
20: getfield TxArrIntField.nonTxValue : int
23: iastore
24: iconst_1
25: iadd
26: dup_x2
27: pop
28: dup2_x1
29: dup2_x1
30: pop
31: dup2
32: pop
33: goto 12
36: pop2
37: pop2
38: dup
39: astore_3
40: iload_1
41: invokestatic java.util.Arrays.binarySearch(int[], int) : int
44: aload_0
45: checkcast TxArrIntField[]
48: aload_3
49: checkcast int[]
52: iconst_0
53: dup_x1
54: dup_x1
55: pop
56: dup_x1
57: arraylength
58: if_icmpge 86
61: swap
62: dup2_x1
63: dup2_x1
64: pop2
65: swap
66: dup2_x2
67: aaload
68: checkcast TxArrIntField
71: dup_x2
72: pop
73: swap
74: iaload
75: putfield TxArrIntField.nonTxValue : int
78: iconst_1
79: iadd
80: dup2_x1
81: dup_x2
82: pop2
83: goto 56
86: pop2
87: pop
88: ireturn
```

5 New array implementation

The current array implementation has some drawbacks, specially when dealing with Java platform methods – see section ??. The problem lies on the fact that all arrays are converted to our `TxArray` classes, which Java platorm methods are of course unaware of. When calling a method, the `TxArray` must be converted back to the original Java array type. A subsequent conversion is needed, constructing the `TxArray` again back from the original array type.

This is terrible. It involves a lot of memory copy and allocation, and could be completely avoided if the method doesn't modify the array. The purpose of the new array implementation is to tackle this issue.

5.1 The `TxArrField` hierarchy

The main problem is that we need, when calling Java platform methods, to recreate the original array type. This is a consequence of the way we replaced the original arrays with our `TxArrField` structure, since it doesn't rely on the original arrays by embedding the value of each element directly into the `TxArrField` class. For example, an `int[]` is replaced by a `TxArrIntField[]` with each integer value stored directly inside each `TxArrIntField` object.

Instead of storing the actual values and becoming independent of the original array, the `TxArrField` structure could rely on the array to hold its values and simply maintaining a reference to it, as in figure ??.

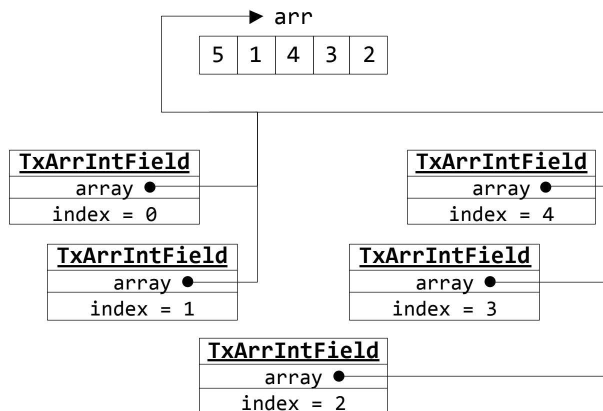

Figure 5: New array structure.

This means that now the original array instead of replaced, cohexists with the `TxArrIntField` structure with the latter poiting to the first.

Listing ?? shows the new `TxArrField` template code, in this case `TxArrIntField`.

Listing 32: `TxArrIntField` class.

```
1  public class TxArrIntField_new
2  extends      TxField
3  {
4    final static private int ARR_BASE  = AddressUtil.arrayBaseOffset(int[].class);
5    final static private int ARR_SCALE = AddressUtil.arrayIndexScale(int[].class);
6
7    public int[] array;
8    public int   index;
9
10   public TxArrIntField_new(int[] arr, int idx) {
11     super(arr, ARR_BASE + ARR_SCALE * idx);
12     array = arr;
13     index = idx;
```

```
14    }
15 }
```

The approach to multidimensional arrays is based on the same idea, but applying the array chaining of figure ??. To use the same approach the `TxArrObjectField` code will have some minor additions compared to the `TxArrIntField` in listing ??, specifically the `nextDim` field. This field behaviour is similar to the `nonTxValue` in listing ??, but the purpose is to chain the array dimensions instead of embedding the element's value.

Consider the following multidimensional array, `cube`.

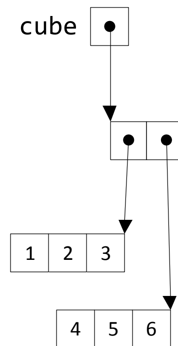

Figure 6: A simple array.

Upon our conversion, `cube` will consist of the following structure in figure ??, where the rightmost array is the original array presented in figure ??. *TxAOF* and *TxAIF* are the `TxArrObjectField` and `TxArrIntField` classes, respectively.

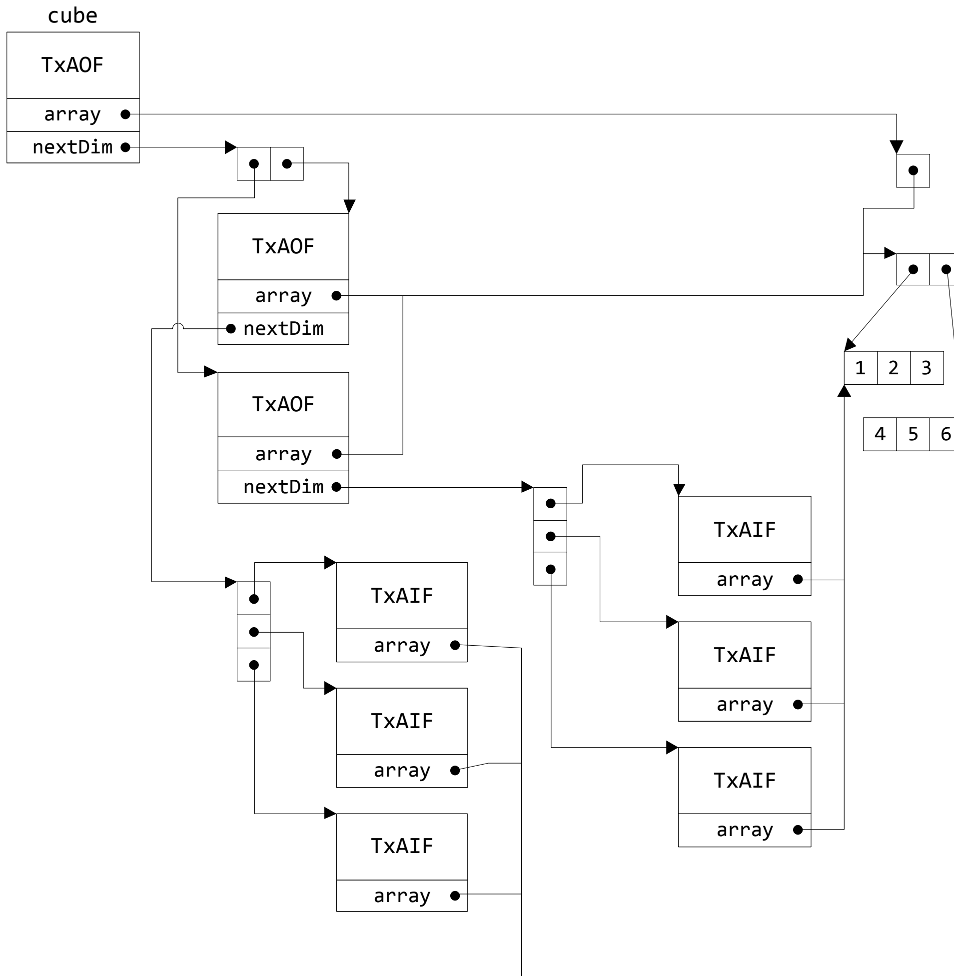

Figure 7: New multidimensional arrays.

In the first dimension, the `nextDim` field will point to the next array holding the metadata objects for the second dimension and the `array` field will point directly to the original array.

The second dimension's metadata objects `nextDim` field will point to the array holding the last dimension's metadata objects, the type of which corresponds to the original array type. In this case, that type is `int[][][]` and therefore the last dimension's metadata objects are `TxArrIntFields`. The `array` field of the second dimension's metadata objects will point to the second dimension of the original array.

Listing ?? shows `TxArrObjectField_new`'s code.

Listing 33: `TxArrObjectField` class.

```
1 public class TxArrObjectField_new
2 extends       TxField
3 {
4   final static private int ARR_BASE  = AddressUtil.arrayBaseOffset(Object[].class);
5   final static private int ARR_SCALE = AddressUtil.arrayIndexScale(Object[].class);
6
7   static public long __ADDRESS__;
8   static {
9     try {
10      __ADDRESS__ = AddressUtil.getAddress(
11            TxArrObjectField_new.class.getDeclaredField("nextDim"));
12    } catch (Exception e) {
```

```
13        __ADDRESS__ = -1L;
14      }
15    }
16
17    public Object[] array;
18    public int      index;
19    public Object   nextDim;
20
21    public TxArrObjectField_new(Object[] arr, int idx) {
22      super(arr, ARR_BASE + ARR_SCALE * idx);
23      array = arr;
24      index = idx;
25    }
26
27    public TxArrObjectField_new(Object[] arr, int idx, Object dummy) {
28      super(null, __ADDRESS__);
29      this.ref = this;
30    }
31  }
```

The first thing to be noticed, compared with `TxArrIntField_new` in listing ??, is the `__ADDRESS__` field which is analogous to the previously described fields of the same name.

Of importance are the two constructors. The first constructor is to be used when instantiating objects to be used in a unidimensional array context because, as you can see, the low level pointer pair given to the `super` constructor references the original array's element. The second constructor is to be used when instantiating objects to be used in a multidimensional array context. Since constructors always have the same name, its signature needs to be different, so the last parameter is exactly that – a dummy parameter whose purpose is only signature changing. As espected, in this case the low level pointer pair is this object itself and `nextDim`'s offset.

The decision to have two constructors instead of one (which could have a `boolean` as parameter to distinguish both situations, for example) is related to performance. We always try to remove as many overhead from the runtime as possible.

Having seen what is the design concept of the new array implementation, and its new `TxArrFields` classes, we will now see what changes are made to the instrumented program's code.

5.2 Generated code

In section ??, the method modifications needed by the original array implementation are described. For the new implementation some are the same, some are slighty different:

1. Any method that receives one or more array parameters, or returns, must have its signature modified to reflect the new type;
2. The `getfield` and `putfield` accesses to, and `checkcast` with, an array field must be updated;
3. The `aload` and `astore` instruction families need to be replaced by `aaload`ing the `TxArrField` at the specified index, a `getfield` to its `array` and `index` fields and an `aload` or `astore`, respectively. If in a multidimensional array context, instead of getting/updating the `array` at `index`, the `nextDim` field is used instead;
4. To all `newarray` and `anewarray` instructions a new `anewarray` of the corresponding `TxArrField` is appended, and subsequent initialization of all elements.
5. The `multianewarray` instructions require more work, as each of the nested unidimensional array elements must be initialized. This is delegated to a generated `final static private` method.

The first two items remain the same, so no examples need to be laid out.

Regarding the third item, array accesses, even though it is slighty different, the idea is pretty much the same. But since the element's value is not on the previous `nonTxValue` field but on the array itself, we load it through the `array` field. Listings ??, ?? and ?? show the old array implementation code, the new implementation code, and the generated bytecode, respectively.

Listing 34: Old array access.

```
1 class ArrayExample {
2   TxArrIntField[] intArrField;
3
4   public void addToArr(int i, int val) {
5     intArrField[i].nonTxValue =
          intArrField[i].nonTxValue + val;
6   }
7 }
```

Listing 35: New array access.

```
1 public void addToArr(int i, int val) {
2   intArrField[i].array[i] =
        intArrField[i].array[i] + val;
3 }
```

Listing 36: Generated bytecode for the new array accesses.

```
public void addToArr(int, int);
  Code:
   0:  aload_0
   1:  getfield arrFieldA : TxArrIntField_new[]
   4:  iload_1
   5:  aload_0
   6:  getfield arrFieldA : TxArrIntField_new[]
   9:  iload_1
   10: aaload
   11: checkcast TxArrIntField_new
   14: dup
   15: getfield TxArrIntField_new.array : int[]
   18: swap
   19: getfield TxArrIntField_new.index : int
   22: iaload
   23: iload_2
   24: iadd
   25: dup_x2
   26: pop
   27: aaload
   28: checkcast TxArrIntField_new
   31: dup
   32: getfield TxArrIntField_new.array : int[]
   35: swap
   36: getfield TxArrIntField_new.index : int
   39: dup2_x1
   40: pop2
   41: iastore
   42: return
```

Regarding the array initialization, listings **??** and **??** show the differences between the old and new implementation. The main difference is that we do not replace the original array instantiation, but instantiate the `TxArray` afterwards. The array is then used to construct the correspondent `TxArrField`, along with the respective index.

The `TxArrIntField_multiarray(int[][], IContext)` helper method initializes the multidimensional `TxArray` and is responsible for its chaining through the `nextDim` field.

Listing 37: Old array initialization.

```
1  class ArrayExample {
2    TxArrIntField[] intArrField;
3    TxArrObjectField[] intMatrxField;
4
5    public ArrayExample() {
6      intArrField = new TxArrIntField[10];
7      for (int i = 0; i < 10; i++)
8        intArrField[i] = new TxArrIntField();
9
10     intMatrxField =
           TxArrIntField_multiarray(10, 10,
           null);
11   }
12
13   private static TxArrObjectField[]
         TxArrIntField_multiarray(int d1, int
         d2, IContext ctx) {
14     TxArrObjectField[] tmp0 = new
           TxArrObjectField[d1];
15     for (int i=0; i < d1; i++) {
16       tmp0[i] = new TxArrObjectField();
17       TxArrIntField[] tmp1 = new
             TxArrIntField[d2];
18       for (int j=0; j < d2; j++) {
19         tmp1[j] = new TxArrIntField();
20       }
21       tmp0[i].nonTxValue = tmp1;
22     }
23
24     return tmp0;
25   }
26 }
```

Listing 38: New array initialization.

```
1  class ArrayExample {
2    TxArrIntField[] intArrField;
3    TxArrObjectField[] intMatrxField;
4
5    public ArrayExample() {
6      int[] unused1 = new int[10];
7      intArrField = new TxArrIntField[10];
8      for (int i = 0; i < 10; i++)
9        intArrField[i] = new
             TxArrIntField(unused1, i);
10
11     int[][] unused2 = new int[10][10];
12     intMatrxField =
           TxArrIntField_multiarray(unused2,
           null);
13   }
14
15   private static TxArrObjectField[]
         TxArrIntField_multiarray(int[][]
         matrx, IContext ctx) {
16     int d1 = matrx.length;
17     TxArrObjectField[] tmp0 = new
           TxArrObjectField[d1];
18     for (int i=0; i < d1; i++) {
19       tmp0[i] = new TxArrObjectField(matrx,
             i);
20       int d2 = matrx[i].length;
21       TxArrIntField[] tmp1 = new
             TxArrIntField[d2];
22       for (int j=0; j < d2; j++) {
23         tmp1[j] = new
               TxArrIntField(matrx[i], j);
24       }
25       tmp0[i].nextDim = tmp1;
26     }
27
28     return tmp0;
29   }
30 }
```

The actual generated bytecode is in listing ??, which is derived from the concept in listing ??. Regarding `ArrayExample()`, lines ??.6-9 map to ??.4-55 and ??.11-12 to ??.58-71.

In `TxArrIntField_multiarray(int[][], IContext)`, ??.16-19 and ??.25 map to ??.0-26 and ??.70-75 and finally ??.20-23 to ??.27-63.

Listing 39: Generated bytecode for the new array initialization.

```
public array.ArrayExample();
  Code:
    0:  aload_0
    1:  invokespecial Object."<init>"() : void
    4:  aload_0
    5:  bipush 10
    7:  newarray int
    9:  dup
    10: arraylength
    11: anewarray TxArrIntField_new
    14: dup2
    15: swap
    16: iconst_0
    17: swap
    18: dup_x1
    19: arraylength
    20: swap
    21: dup_x1
    22: if_icmple 51
    25: dup_x2
```

```
   26: dup_x1
   27: new TxArrIntField_new
   30: dup_x2
   31: dup_x2
   32: pop
   33: invokespecial TxArrIntField_new."<init>"(int[], int) : void
   36: aastore
   37: iconst_1
   38: iadd
   39: dup_x2
   40: pop
   41: swap
   42: dup_x2
   43: swap
   44: dup_x2
   45: dup_x2
   46: pop
   47: swap
   48: goto 17
   51: pop2
   52: pop
   53: swap
   54: pop
   55: putfield arrFieldA : TxArrIntField_new[]
   58: aload_0
   59: bipush 10
   61: bipush 10
   63: multianewarray int[][], 2
   67: aconst_null
   68: invokestatic TxArrIntField_new_multiarray(int[][], IContext) :
       TxArrObjectField_new[]
   71: putfield arrFieldB : TxArrObjectField_new[]
   74: (...) // Metadata initialization

private static final TxArrObjectField_new[] TxArrIntField_new_multiarray(int[][],
    IContext);
  Code:
   0:  aload_0
   1:  arraylength
   2:  anewarray TxArrObjectField_new
   5:  astore_2
   6:  iconst_0
   7:  istore_3
   8:  iload_3
   9:  aload_2
   10: arraylength
   11: if_icmpge 84
   14: aload_2
   15: iload_3
   16: new TxArrObjectField_new
   19: dup
   20: aload_0
   21: iload_3
   22: aconst_null
   23: invokespecial TxArrObjectField_new."<init>"(Object[], int, Object) : void
   26: aastore
   27: aload_0
   28: iload_3
   29: aaload
   30: arraylength
   31: anewarray TxArrIntField_new
   34: astore 4
   36: iconst_0
   37: istore 5
   39: iload 5
   41: aload 4
   43: arraylength
   44: if_icmpge 70
   47: aload 4
   49: iload 5
```

```
51: new TxArrIntField_new
54: dup
55: aload_0
56: iload_3
57: aaload
58: iload 5
60: invokespecial TxArrIntField_new."<init>"(int[], int) : void
63: aastore
64: iinc 5, 1
67: goto 39
70: aload_2
71: iload_3
72: aaload
73: aload 4
75: putfield TxArrObjectField_new.nextDim : Object
78: iinc 3, 1
81: goto 8
84: aload_2
85: areturn
```

Regarding this generated bytecode, is important to understand how the local variables are organized. The following enumeration maps the local variable index to what resides inside it, in the bytecode in listing

0. Original multidimensional array;
1. `IContext`;
2. `TxArrObjectField[]`, the first dimension
3. `int`, the first dimension's cycle counter variable – i;
4. `TxArrIntField[]`, the second and last dimension
5. `int`, the second dimension's cycle counter variable – j.

You can see the pattern. If we were to have a tridimensional array of `int`s instead, we'd have:

0. Original multidimensional array;
1. `IContext`;
2. `TxArrObjectField[]`, the first dimension
3. `int`, the first dimension's cycle counter variable;
4. `TxArrObjectField[]`, the second dimension
5. `int`, the second dimension's cycle counter variable.
6. `TxArrIntField[]`, the third and last dimension
7. `int`, the third dimension's cycle counter variable.

As said in the beginning of this section, the way Java platform methods are handled in the old implementation was the main reason to rethink the array implemetation. The fact that we completely replaced the original arrays brought big overhead when calling these methods, because we had to recreate the original array from the `TxArray`, pass it along to the Java platform method, and afterwards copy all elements back to the `TxArray`.

With the new implementation, the original array still exists, accessible through the `array` field, so the only thing we need to do is pass it to the Java platform method.

Listings **??**, **??** and **??** show the old implementation, the new implementation, and the generated bytecode. As we can see, there is a huge improvement regarding memory allocation and copy.

Listing 40: Old Java method wrapped call.

```
1 class ArrayExample {
2   int[] intArrField;
3
4   public int existsInArr(int val) {
5     return java_util_Arrays_binarySearch(
6       intArrField, val, null);
7   }
8
9   private static final int
        java_util_Arrays_binarySearch(
10    TxArrIntField arr, int val, IContext
        ctx)
11  {
12    int tmp[] = new int[arr.length];
13    for (int i = 0; i < arr.length; i++)
14      tmp[i] = arr[i].nonTxValue;
15
16    int ret =
          java.util.Arrays.binarySearch(tmp,
          val);
17
18    for (int i = 0; i < arr.length; i++)
19      arr[i].nonTxValue = tmp[i];
20
21    return ret;
22  }
23 }
```

Listing 41: New Java method wrapped call.

```
1 private static final int
      java_util_Arrays_binarySearch(
2 TxArrIntField arr, int val, IContext ctx)
3 {
4   return java.util.Arrays.binarySearch(
5     arr.array, val);
6 }
```

Listing 42: Generated bytecode for the new Java platform method call.

```
public int existsInArr(int);
  Code:
   0:  aload_0
   1:  getfield arrFieldA : TxArrIntField_new[]
   4:  iload_1
   5:  aconst_null
   6:  invokestatic java_util_Arrays_binarySearch(TxArrIntField_new[], int, IContext) :
       int
   9:  ireturn

private static final int java_util_Arrays_binarySearch(TxArrIntField_new[], int,
    IContext);
  Code:
   0:  aload_0
   1:  iconst_0
   2:  aaload
   3:  getfield TxArrIntField_new.array : int[]
   6:  dup
   7:  astore_3
   8:  iload_1
   9:  invokestatic Arrays.binarySearch(int[], int) : int
  12:  ireturn
```

6 Performance evaluation

t.vale: TODO.