Runtime Elision of Transactional Barriers for Captured Memory

Abstract

Even though Software Transactional Memory (STM) is a promising approach to simplify concurrent programming, current STM implementations incur into significant overheads that render them impractical for many real-sized programs. These overheads are often attributed to over-instrumentation made by STM engines that protect every memory access with a call to an STM memory access function (i.e. *STM barriers*).

Yet, these STM barriers are useless when accessing transaction-local memory (i.e. *captured memory*), because this memory is not visible to other transactions until the successful commit of its allocating transaction. Although many techniques (both at runtime and at the compiler level) have been proposed to elide unnecessary STM barriers, none is able to remove STM Barriers for captured memory efficiently.

In this work we propose a new technique that can identify captured memory accurately in managed environments, while having a low runtime overhead. We implemented our proposal in a well known STM framework (Deuce) and we tested it in STMBench7 with two different STMs: TL2 and LSA. In both STMs the performance improved dramatically (43 times and 38 times, respectively) leading them to outperform the coarse-grain lock strategy and in the case of LSA approximating the performance of the medium-grain lock strategy. Moreover, running the STAMP benchmarks with our approach shows improvements of 5 times in the best case for the Vacation application and without any performance penalty in the worst case of the Ssca2.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming - Parallel Programming

General Terms Performance, Transactions

Keywords Software Transactional Memory, Runtime Optimizations

1. Introduction

Some researchers (e.g. [7]) question the usefulness of Software Transactional Memory (STM), because most STM implementations fail to demonstrate applicability to real-world problems: In many cases, the performance of an STM on a real-world–sized benchmark is significantly lower than the sequential version of the benchmark, or even than the version using coarse-grain locks. The loss of performance is often attributed to the *over-instrumentation* [21] made on these benchmarks by overzealous STM compilers that protect each and every memory access with a barrier that calls back to the STM runtime.

In fact, over-instrumentation is one of the major sources of overhead in concurrent applications that are synchronized with STMs and, thus, several researchers proposed optimization techniques to elide useless barriers—for instance, to elide barriers when accessing transaction local memory [10] (i.e. *captured memory*). Many of the existing approaches (e.g. [3, 6, 14, 21]) decompose the STM's API in heterogeneous parts that allow the programmer to convey application-level information about the behavior of the memory locations to the instrumentation engine.

Yet, this approach contrasts with one of the main advantages of an STM, which is to provide a transparent synchronization API, meaning that programmers just need to specify which operations are atomic, without knowing which data is accessed within those operations. That is the approach taken by Deuce [15], which provides a simple API based on an `@Atomic` annotation to mark methods that must have a transactional behavior.

So, approaches such as those proposed in [1, 10], which perform runtime or static *capture analysis* to identify whether memory locations being accessed are captured (transaction-local), are better suited to the overall goal of STMs. Yet, none of these approaches accomplishes the performance speedups that are shown in the results presented by the proposals based in heterogeneous APIs.

In this paper we propose a new technique for runtime capture analysis in managed environments that, to the best of our knowledge, is the first one to achieve performance results similar to those obtained with heterogeneous APIs, but without reducing the transparency of an STM. We implemented our technique in the well-known Deuce STM framework for the Java environment, incorporating our capture analysis approach in both the LSA [18] and TL2 [8] STMs supported by Deuce.

Running two distinct real-world–sized benchmarks—the STMBench7 [13] and the Vacation [5]—with our approach in either of the STMs of Deuce shows huge performance gains, making them perform better than the coarse-grain lock approach.

A distinctive aspect of our approach is that it performs a lightweight analysis when compared to other capture analysis implementations (e.g. [10]) and it has almost no overhead when the benchmark presents no opportunities for optimizations such as on the Kmeans and Ssca2 benchmarks from STAMP [5]. The key idea of our solution consists in recording on every transactional object the identity of its allocating transaction—its *owner* transaction. Later, and before performing an STM barrier, we can verify if the accessed object is in captured memory comparing its owner with the identity of the transaction that is accessing that object.

The remainder of this paper is organized as follows. In Section 2, we give an overview of the key aspects of Deuce. In Section 3, we describe our runtime technique for capture analysis, and then, in Section 4, we explain how we have integrated this solution

in Deuce. In Section 5, we describe our performance evaluation. In Section 6, we discuss related work on different approaches taken to reduce STM compiler over-instrumentation. Finally, in Section 7, we conclude and discuss some future work.

2. Deuce Overview

Deuce is a Java STM framework that provides a bytecode instrumentation engine implemented with ASM [4]. Its two major goals are: (1) to be able to integrate the implementation of any synchronization technique, and in particular different STMs; and (2) to provide a transparent synchronization API, meaning that the end user programmer just needs to be concerned with the specification of the *atomic methods*, i.e. which methods should execute with an atomic semantics. For this purpose, the programmer should mark those methods with an `Atomic` annotation and the Deuce engine will automatically synchronize their execution using the synchronization technique specified by the end user.

The synchronization mechanism is defined by a class that implements the `Context` interface. This interface specifies the API of the STM barriers that each STM implementation must provide and that is used by the code instrumented by Deuce to notify the STM engine whenever one of the following events occurs during the execution of the instrumented program: the begin of an atomic method (`init` event handler); the end of an atomic method (either `commit` or `rollback` event handlers); the access to a memory location made inside of a *transactional scope*—i.e. an atomic method or any method invoked in the scope of an atomic method—(`beforeReadAccess`, `onReadAccess`, and `onWriteAccess` event handlers). To use memory barriers only when accessing memory from within a transactional scope, Deuce creates a duplicate of every method—a *transactional method*—where every memory access is replaced by the invocation of the corresponding STM barrier implemented by the `Context` interface. Depending on whether a method is invoked from inside or from outside a transactional scope, then either the transactional version or the original version, will be invoked, respectively.

```
class Counter{
  private int n;

  public int next(){
    int current = n;
    current++;
    n = current;
    return current;
  }

  // Synthetic members generated by Deuce

  private static long n__ADDRESS__ = ...;

  public int next(Context c){
    c.beforeReadAccess(this, n__ADDRESS__);
    int current = c.onReadAccess(this, n, n__ADDRESS__);
    current++;
    c.onWriteAccess(this, current, n__ADDRESS__);
    return current;
  }
}
```

Figure 1. Resulting class `Counter` from the instrumentation of the Deuce engine

In Figure 1 we show an example of the resulting class after the instrumentation of an hypothetical class `Counter` by the Deuce engine. For the method `next` Deuce generated a new transactional version of this method, which receives an additional `Context` parameter. Then, every memory access from inside this method, such as reading or writing to the field `n`, is replaced by the invocation of the corresponding STM barrier. Moreover, all calls to the `Counter.next` method made within other transactional methods will be replaced with calls to this new transactional method `next`.

In truth, the barriers defined by a `Context` implementation are not directly invoked by the transactional methods of an instrumented class, as depicted in the Figure 1 (we did it just for simplification of the code) and this detail is relevant for the implementation of our solution, which we will explain in the section 4. In the deuce framework there is a class `ContextDelegator`, with just static methods and similar signatures to those ones specified by the `Context` interface—e.g. for the barrier `onWriteAccess( Object obj, int value, long add)` of the `Context` interface there is a corresponding static method `onWriteAccess( Object obj, int value, long add, Context c)`, receiving an additional parameter `Context`, in the `ContextDelegator` class. So, according to the invocation flow of the barriers, a field access is replaced by the invocation of a static method in the `ContextDelegator` class, which in turn invokes the corresponding barrier in the `Context` object, performing e.g. `c.onWriteAccess(obj, value, add)`.

Besides other reasons related to the implementation details of the Deuce, one of the goals of the `ContextDelegator` is to unify the access between objects and arrays to the same barriers implemented by a `Context`—i.e. the Deuce deals with array elements like fields of an object. So, for each barrier of each type there is also a static method in the `ContextDelegator` class for the corresponding array of that type—e.g. for the barrier `onReadAccess` of the `int` type, there is a corresponding static method `onArrayReadAccess(int[] a, int index, Context c)` which in turn invokes `c.onReadAccess(a, a[index], address)`, where the `address` is calculated from the value of the `index` parameter.

3. Capture Analysis Implementation

Dragojevic et al. [10] were the first describing a technique for runtime capture analysis. In Figure 2, we depict the code skeleton of a read and a write barrier using runtime capture analysis, which was adapted from the original proposal of Dragojevic et al. [10] to the Deuce environment. In the case of Deuce, object fields are updated in place using the `sun.misc.Unsafe` pseudo-standard internal library, and for each primitive type there is a corresponding barrier in the `Context` implementation (for simplification we just depict the code for the `int` type).

The efficiency of this solution is directly dependent of the overhead of the capture analysis, made by the `isCaptured` function. Its original implementation was made in the Intel C++ STM compiler and its algorithm is close to the memory management process. The key idea of this algorithm consists in comparing the address of the accessed Object—`ref`—with the range of memory locations allocated by the transaction. To perform this analysis, all transactions must keep a *transaction-local allocation log* for all allocated memory, including the two memory spaces: the *heap* and the *stack*. So, the efficiency of the `isCaptured` function depends of the efficiency of the search algorithm that needs to lookup the allocation log for a specific address, and ultimately depends on the number of allocations made within a transaction.

In our solution that is dependent of the characteristics and the restrictions imposed by a managed environment, we cannot explicitly manage neither the memory allocation, nor which memory space to use for new objects—i.e. reference type instances are automatic allocated in the managed heap by the JVM.

So we followed a distinct approach that consists in uniquely identifying each transaction with a *fingerprint*, which is recorded in every object instantiated by a transaction, representing its *owner*

```
public class FooStmContext implements Context{

 public int onReadAccess(Object ref, int val, long add){
  if(isCaptured(ref))
   return val;
  return onFullReadAccess(ref, val, add);
 }

 public void onWriteAccess(Object ref, int val, long add){
  if(isCaptured(ref))
   UnsafeHolder.getUnsafe().putInt(ref, add, value);
  else
   onFullWriteAccess(ref, val, add);
 }
 ...
 // There is a corresponding pair of barriers for each
 // primitive type.
}
```

Figure 2. The code skeleton of a read and a write barrier using runtime capture analysis (for simplification we omit the `beforeReadAccess` barrier, which has a similar structure to the `onReadAccess`).

transaction. A new fingerprint is initialized and stored in the `Context` object of the executing thread, when it is notified of the beginning of a new transaction. Besides that, given that Deuce uses linear nesting [2], meaning that a transaction can have only one pending child transaction within it, the fingerprint of a top-level transaction can be shared across its nested transactions. In Figure 3, we depict the code of a `Context` implementation that controls the initialization of new fingerprints. We also depict the code of the `isCaptured` method and we represent the accessed object `ref` as a reference of the type `CapturedState`, which defines the `owner` field that holds the fingerprint of the allocating transaction. If an object is instantiated out of a transactional scope then its `owner` field will be `null`.

```
protected Object trxFingerprint = null;
protected int nestedLevel = 0;


@Override
public void init(int atomicBlockId, String metainf) {
  nestedLevel++;
  if (nestedLevel == 1) trxFingerprint = new Object();
        ...
}

@Override
public boolean commit() {
  nestedLevel--;
  ...
}

@Override
public boolean rollback() {
  nestedLevel--;
  ...
}

protected boolean isCaptured(CapturedState ref){
  return ref.owner == trxFingerprint;
}
```

Figure 3. A `Context` implementation for controlling the generation of new fingerprints and the `isCaptured` algorithm. The `commit` and `rollback` events are disjoint— i.e.the transaction control flow ends with one of these events.

The `isCaptured` algorithm just needs to check if the owner of the accessed object corresponds to the same fingerprint of the executing `Context`. Every time a new top-level transaction begins, the corresponding `Context` object will get a new fingerprint. So, when a new object is published by the successful commit of its allocating transaction, then we have the guarantee that every running and newly created transaction calling the `isCaptured` method will get `false` in return, because their fingerprint cannot be the same as that one recorded on that object. We do not need to clear the context's fingerprint at the end of the top-level transaction, because a new fingerprint will be produced on the initialization of the next top-level transaction.

In Figure 4, we show an example of three different transactions sharing a `Counter` object instantiated by one of those transactions—transaction 1. The bar bellow each thread has a number representing the `id` of each transaction. In this example the thread `A` performs transactions 1 and 3, while the thread `B` performs transaction 2. In this case just the transactions 2 and 3 perform full barriers, while transaction 1 returns and updates the `Counter` object in place. The `Context` of the thread `A` has the same fingerprint of the object `Counter`, only during the execution of the transaction 1, avoiding in this case a full barrier. After the completion of transaction 1, no other `Context` will have the same fingerprint of the object `Counter` and all subsequent transactional accesses to this object must perform a full barrier, as happens for transactions 2 and 3.

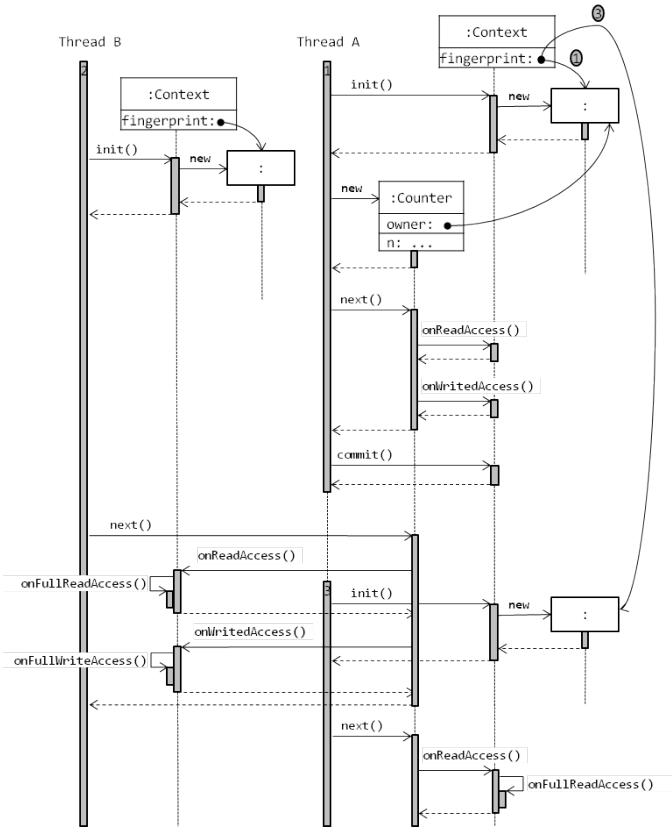

Figure 4. Three different transactions accessing a shared object `Counter` that was previously instantiated by transaction 1, which is the only one that avoids the execution of the full barriers when accessing that object. For simplification we omit the invocation of the `beforeReadAccess` barrier.

4. Deuce Adaptation

We integrated our capture analysis solution in the Deuce framework keeping the following guidelines in mind: 1) avoiding changes to the current Deuce API; 2) guarantying retro-compatibility with existing applications and STMs for the Deuce; 3) providing the ability to enhance any existing STM with the capture analysis technique without requiring either its recompilation or any modification to its source-code.

Enhancing the Deuce with the capture analysis technique requires three main changes to the Deuce core structures: 1) a *transactional class* (i.e. a class whose instances are accessed in a transactional scope) must have an additional field—`owner`—that stores the fingerprint of the transaction that instantiates it; 2) the `Context` implementation of any STM must keep a fingerprint representing the identity of the transaction in execution by a context; 3) all STM barriers defined by a `Context` implementation must perform the capture analysis dictated in the code of the figure 2. In the following we will explain the introduction of this enhancements by the same order, but first we deal with transactional objects and then with arrays.

4.1 Transactional Objects in Captured Memory

4.1.1 `CapturedState`

The entry point of the instrumentation engine is defined in the Deuce class `Agent`, which implements the interface `java.lang.instrument.ClassFileTransformer`. This class is responsible for instrumenting all the classes from a jar archive, or from the class loader, depending when the instrumentation engine is performed in offline mode (running the `main` method), or as a Java agent (running the `premain` method), as depicted in the code of the figure 5. In both cases, it is invoked the core method `transform`, which receives a `byte[]` holding the bytecodes of the original class definition and returns a new `List<ClassByteCode>`, which includes the bytecodes of the transformed class and other auxiliary classes added by the transformation. So, the standard transformation applied by Deuce is defined by the class `org.deuce.transform.asm.ClassTransformer`, which is invoked by the previous mentioned `transform` method.

```
public class Agent implements ClassFileTransformer {

  public static void main(String[] args){
    ...
    for (JarEntry nextJarEntry : ...){
      List<ClassByteCode> transformed = transform(...);
      ...
    }
  }
  public static void premain(..., Instrumentation inst) {
    inst.addTransformer(new Agent());
  }
  @Overrides
  public byte[] transform(..., byte[] bytecodes){
    List<ClassByteCode> transformed = transform(...);
    ...
  }
  List<ClassByteCode> transform(..., byte[] bytecodes){
    ClassTransformer cv = new ClassTransformer(...);
    bytecodes = cv.visit(bytecodes);
    ...
  }
}
```

Figure 5. Skeleton of the class `Agent`, with two different entry points: `main` and `premain`, depending when the instrumentation is performed in offline mode or as a Java agent.

To include the modification pointed in the first enhancement we extended the Deuce framework to receive an additional instrumentation parameter, which specifies a second transformation that should be applied to the resulting class from the standard Deuce instrumentation. In view of new features that we would like to include in the future, we consider that additional transformations could be needed after, or before, the standard Deuce instrumentation of classes. So instead of a unique transformation, we allow the specification of multiple transformations that can be applied before, or after, the standard Deuce instrumentation, through the specification of the corresponding parameters: `org.deuce.transform.pre` and `org.deuce.transform.pos`. These parameters specify classes that implement the interface `ClassEnhancer`, depicted in the figure 6.

```
public interface ClassEnhancer {
  List<ClassByteCode> visit(
    boolean offline, String className, byte[] bytecodes);
}
```

Figure 6. `ClassEnhancer interface.`

For the implementation of the first enhancement we defined the `ClassEnhancerCapturedState` that specifies a transformation responsible for replacing the top of the classes hierarchy from `Object` to the `CapturedState` class depicted in the figure 7, which adds an extra field `owner` to all transactional classes. Depending when a transactional class is instantiated outside, or inside a transactional scope, then it will be invoked the corresponding parameterless constructor, or the constructor with a `Context` parameter. The proper functioning of this enhancement admits that at runtime the `Context` object is an instance of the class `ContextFilterCapturedMem`, otherwise it will not make sense to run this feature isolated. We will explain it in the next subsection.

```
public class CapturedState {

  private final Object owner;

  public CapturedState(){
    this.owner = null;
  }
  public CapturedState(Context ctx){
   this.owner =
    ((ContextFilterCapturedMem) ctx).getTrxFingerprint();
  }
}
```

Figure 7. `CapturedState` class adds an extra field `owner` to all transactional classes.

4.1.2 Transaction Fingerprint

According to the second enhancement, a `Context` object must keep the fingerprint of the transaction that is running, for later perform the capture analysis, as depicted in the code of the figures 2 and 3. But keeping the third guideline in mind, we would like to add this feature to any existing STM for Deuce, without the need of recompiling or changing its source-code.

For that purpose we added a new execution parameter to the Deuce framework—`org.deuce.filter`—that enables the specification of a *filter context*— i.e. a class that also implements the `Context` interface and adds some functionality to any existing `Context` (following the decorator design pattern [12]). The new context `ContextFilterCapturedMem` follows this approach using the code depicted in the figure 3, which can be applied to an existing `Context` of any STM. The figure 8 depicts the code of the class `ContextFilterCapturedMem` (for simplification we omit the method `rollback` that performs a similar code to the method `commit`).

```
public class ContextFilterCapturedMem implements Context{
  protected final Context ctx;
  protected Object trxFingerprint = null;
  protected int nestedLevel = 0;

  public ContextFilterCapturedMem(Context ctx){
    this.ctx = ctx;
  }
  @Override
  public void init(int atomicBlockId, String metainf){
    nestedLevel++;
    if(init == 1) trxFingerprint = new Object();
    ctx.init(atomicBlockId, metainf);
  }
  @Override
  public boolean commit(){
    nestedLevel--;
    return ctx.commit();
  }
  ...
}
```

Figure 8. The `ContextFilterCapturedMem` class is a context decorator that could add the transaction fingerprint to any existing STM `Context` implementation.

4.1.3 Capture Analysis

We could use the previous decorator `ContextFilterCapturedMem` to include the capture analysis in each barrier, following the same approach of the `init` and `commit` methods. Yet, these barriers are common to objects and arrays accesses, as we explained in section 2 and for arrays we cannot perform the same instrumentation of transactional objects, as specified in the subsection 4.1.1, because we have no way to change their top class hierarchy and consequently we need a different capture analysis approach to deal with arrays. So we have to take a step back in the barriers invocation chain and perform the capture analysis at the `ContextDelegator` level.

Once the methods of the `ContextDelegator` class are static (not virtual) we cannot override them in the same way as we do for the `Context` interface. But we can replace the whole class with a new one preserving the same methods signatures. In fact, the instrumentation engine refers to this class by its internal name that is stored in a global constant—`CONTEXT_DELEGATOR_INTERNAL`—and if we replace it with the name of the new class containing the same methods with a new implementation, then this new class will receive the notification of the fields accesses. So, we defined a new instrumentation parameter—`org.deuce.delegator`—that specifies the name of the class that will replace the `ContextDelegator`.

We defined the class `ContextDelegatorCapturedMem` whose methods perform the capture analysis algorithm as specified in the figure 9. For simplification we have omitted the code of the write barrier that performs a similar verification to the read barrier.

As happens in the enhancement 4.1.1, now it does not make sense to use this new delegator without performing the instrumentation specified by the `ClassEnhancerCapturedState` and without the filter `ContextFilterCapturedMem`. Otherwise we could get a `ClassCastException` on the execution of a read or write barrier.

Concluding, to run an application with the Deuce framework performing the capture analysis for transactional objects we must specify the class enhancer `ClassEnhancerCapturedState` through the parameter `org.deuce.transform.pos`, the filter `ContextFilterCapturedMem` through the parameter `org.deuce.filter` and a new delegator in the parameter `org.deuce.delegator`. The expression of the figure 10 exemplifies the execution

```
public class ContextDelegatorCapturedMem{
 static public int onReadAccess(
  Object ref, int val, long add, Context c)
 {
  if(((CapturedState) ref).getOwner() ==
     ((ContextFilterCapturedMem) c).ownershipMark)
  {
    return UnsafeHolder.getUnsafe().getInt(ref, add);
  }else
  {
    return c.onReadAccess(ref, val, add);
  }
 }
 ...
}
```

Figure 9. The `ContextDelegatorCapturedMem` class can replace the original `ContextDelegator` class adding the capture analysis to every barrier.

of the `Vacation` application instrumented by Deuce and running the LSA STM with these new features.

```
java -javaagent:deuceAgent.jar
 -Dorg.deuce.transaction.contextClass=
    org.deuce.transaction.lsa.Context
 -Dorg.deuce.transform.pos=
    org.deuce.transform.asm.ClassEnhancerCapturedState
 -Dorg.deuce.filter=
    org.deuce.transaction.ContextFilterCapturedMem
 -Dorg.deuce.delegator=
    org.deuce.transaction.ContextDelegatorFilterCapturedMem
 jstamp.vacation.Vacation
```

Figure 10. Executing the `Vacation` application instrumented by Deuce and running the LSA STM with capture analysis.

To simplify the execution of the Deuce with the capture analysis we also defined a unique boolean parameter that includes the previous three features and could be specified like this: `-Dorg.deuce.capmem=true`.

4.2 Arrays in Captured Memory

4.2.1 `CapturedStateArrayBase`

In the JVM we cannot instrument the array types as we do for the other transactional classes, changing their super class to `CapturedState`. So, we have to follow a different approach, which may include in arrays an additional field `owner` that identifies its allocating transaction. For this purpose, we need to wrap each array instance inside an object with two fields: one to store the array itself (the `elements` field) and other to store the fingerprint of the allocating transaction (the `owner` field). This wrapper is defined by the class `CapturedStateArrayBase`, which inherits from the `CapturedState`. Yet, this class does not specifies the `elements` field, because we cannot define a generic array comprising arrays of primitive types. So, we have one inherited class for each primitive type and one more class for arrays of reference types. The name of these classes follow the convention `CapturedState<T>Array`.

Moreover the `CapturedStateArrayBase` also defines the common behavior to all `CapturedState<T>Array` classes—the methods `arrayLength()` and `unwrapp()` (this one is just useful for accessing multidimensional arrays from non-transactional methods).

Then, taking advantage of the new Deuce enhancement infrastructure, explained in the subsection 4.1.1, we defined the enhancer `ClassEnhacerCapturedStateArray`, responsible for transforming all operations dealing with arrays in new operations manipulating instances of the `CapturedState<T>Array` classes.

This solution just have a drawback for fields of multidimensional arrays, when they are accessed out of a transactional scope. In this case the encapsulated array must be unwrapped from the `CapturedState<T>Array` instance. But, unlike unidimensional arrays, where the unwrap operation only consists in getting its `elements` field, for multidimensional arrays we need to instantiate a new array and copy the elements from each unidimensional array. Note that in Java a multidimensional array is an array of arrays and in our design of the captured memory this is represented by an instance of a `CapturedStateObjectArray`, which in turn encapsulates an array of other `CapturedState<T>Array` objects and so on. For this reason, the unwrap of a multidimensional array has a so big overhead.

4.2.2 Transaction Fingerprint

This process is availed from the filter `ContextFilterCapturedMem` already defined for transactional objects and has no differences for transactional arrays, working in the same way.

4.2.3 Capture Analysis for Arrays

The class `ClassEnhacerCapturedStateArray` is a pos enhancer, meaning that it should run after the standard instrumentation of the Deuce engine. At this phase all the array accesses have already been replaced by the invocation of the corresponding barriers in the `ContextDelegator` class.

But after the second instrumentation of the `ClassEnhacerCapturedStateArray` all the barriers that were invoked passing an array by argument are now invoked with that array wrapped inside a `CapturedState<T>Array` object. So, we need to extend the array barriers of the `ContextDelegatorCapturedMem` with new methods according to the previous invocation. For this purpose, we duplicated every array barrier with a corresponding method that replaces the array argument by a `CapturedState<T>Array`. These methods performs the capture analysis and invoke the corresponding array barrier depending whether the array is allocated, or not, in captured memory, as depicted in the example of the figure 11.

```
public class ContextDelegatorCapturedMem{
 static public int onArrayReadAccess(
  CapturedStateIntArray ref, int index, Context c)
 {
  if(ref.getOwner() ==
   ((ContextFilterCapturedMem)context).ownershipMark)
  {
   return ref.elements[index];
  }else
  {
   return onArrayReadAccess(ref.elements, index, c);
  }
  ...
}
```

Figure 11. An example of an array read barrier of the `ContextDelegatorCapturedMem` performing capture analysis.

Concluding, to run an application instrumented by the Deuce engine performing a full capture analysis for transactional objects and arrays, we should execute the expression depicted in the figure 10 but with the additional value `ClassEnhacerCapturedStateArray` separated by comma, in the parameter `org.deuce.transform.pos`. As before, we defined an extra flag—`org.deuce.capmemfull`—that includes all the required arguments to perform in this case the full capture analysis.

5. Performance evaluation

All the tests were performed on a machine with 4 AMD Opteron(tm) 6168 processors, each one with 12 cores, resulting in a total of 48 cores. The JVM version used was the 1.6.0 33-b03, running on Ubuntu with Linux kernel version 2.6.32.

To evaluate the performance benefits of our approach, we used the StmBench7 [13] and the Stamp [5] benchmarks with the LSA [18] and the TL2 [8] STMs. In all tests we evaluate the impact of our solution comparing the execution of the capture analysis when it is performed in partial mode just for transactional objects (identified by the suffix cap) and when it performs a full analysis for both objects and arrays (identified by the suffix capfull).

All the results are presented in terms of speedup in comparison to the sequential execution of the same application without instrumentation by one thread.

5.1 StmBench7 benchmark

The StmBench7 is a benchmark for performance evaluation implemented in Java that models a realistic large scale CAD/CAM application. The data structure of STMBench7 consists in a large graph of different kinds of objects and its operations manipulate large parts of this data structure. These operations vary in the length of the path that is randomly selected from the graph of objects. The main feature of STMBench7 is that it uses long transactions and large data structures, becoming a big challenge to overcome for the majority of STMs, as shows the work of of Dragojevic et al. [9], concluding that this benchmark stretches STMs too much with regard to memory requirements.

The STMBench7 benchmark provides three kinds of workloads that vary in the ratio of update operations: *read-dominated* (10% update operations), *read/write* (40% update operations), and *write-dominated* (90% update operations). We omit the results for the write-dominated workload, which are identical to the read-write workload. We can still enable or disable long transactions for each workload and in our tests the structural modifications and long traversals were disabled. In the results of the figures 12 and 13 we do not include the full capture analysis, because there is no difference to the performance of the partial anaysis, due to the absence of arrays manipulation in the StmBench7 operations.

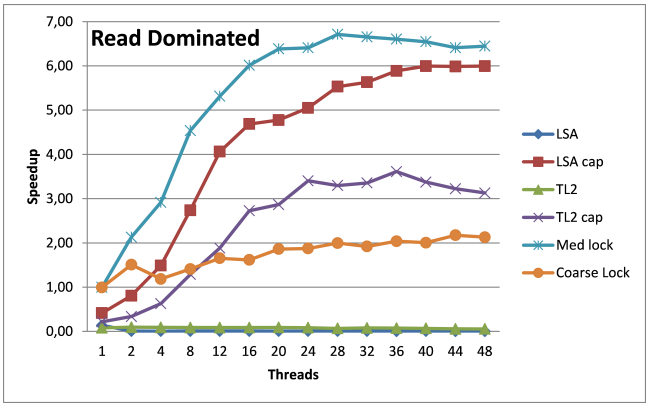

Figure 12. The StmBench7 with a read-dominated workload

As we can see in the results of the figures 12 and 13 the LSA and TL2 present no scalability in the StmBench7 when they run without the capture analysis and their performance is even worse than a sequential execution of one thread without instrumentation. On the other hand the capture analysis has an huge influence in the performance of both STMs and in the case of the LSA for the read-write workload, its performance is near of the behavior of the medium-grain lock synchronization strategy.

Typically, the STMBench7 operations focus on object manipulation along the graph of objects, which is tackled through an intensive use of object iterators. These iterators are transaction local

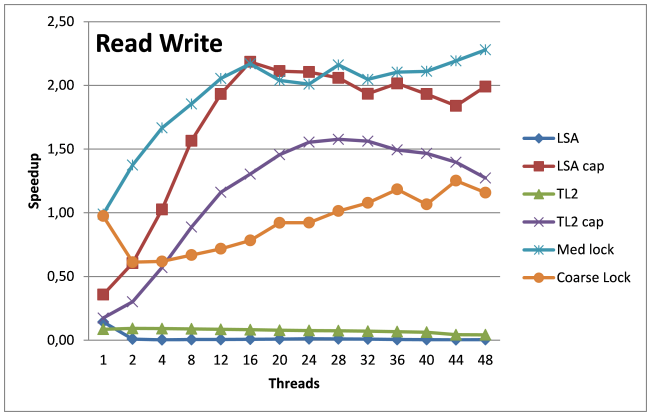

Figure 13. The StmBench7 with a read-write workload

and they have a big overhead when they are instrumented and performed using STM barriers. For this reason our capture analysis solution has a colossal influence in the performance speedup of the Deuce in the StmBenhch7.

Finally performing the capture analysis, the LSA is almost 40% better than the TL2 at its peak of performance. This happens because the LSA takes advantage of read-only transactions. This is another feature that is not correctly accomplished without the capture analysis, because all transactions behave like read-write, once the instanciated iterators perform internal updates that promote the read-only transactons to the read-write mode. For this purpose we had to include a minor change in the StmBench7, splitting the definition of the generic atomic block of the Deuce, in two functions: one for the execution of read-write transactions and other for read-only. We need this modification because the LSA keeps a log that registers the kind of transaction performed by every atomic function. So, once an atomic function performs a read-write transaction, then the LSA will no longer try to perform a read-only transaction for that function.

5.2 STAMP benchmarks

STAMP is a benchmark suite that attempts to represent real-world workloads in eight different applications. Unlike STMBench7, the STAMP applications are configurable and allow us to vary the level of contention, size of transactions, the percentage of writes, among other parameters. In all tests we ran the benchmarks with the highest contention and the largest data set configuration described in [5].

We tested four STAMP benchmarks: K-Means, Ssca2, Intruder and the Vacation. The original implementation of the STAMP is available as a C library and these four benchmarks are the only ones available for Java in the public repository of the Deuce and running with correct results. The K-Means implements k-means clustering, the Ssca2 is comprised of four kernels that operate on a large, directed, weighted multi-graph (this benchmarks just allows the execution of a number of threads multiple of two), the Intruder scans network packets for matches against a known set of intrusion signatures and the Vacation simulates an on-line travel reservation system.

The results of the figures 14 and 15 confirm the analysis of Dragojevic et al. [10] about the opportunities for barrier elision available in each benchmark of the STAMP suite. According to their work the K-Means and Ssca2 applications do not access transaction local memory and in these cases we do not have opportunities for eliding barriers with our capture analysis technique. Yet, the results of the figures 14 and 15 still show that our technique for capture analysis has almost no overhead in performance when running in its partial mode, targeting only the transactional objects and ignoring arrays.

The full capture analysis has a big overhead in the Ssca2 because of the multi-dimensional arrays fields that are accessed out of a transactional scope, requiring the unwrap of the array encapsulated in a `CapturedState<T>Array` object. For this reason we did not perform the full capture analysis for the Ssca2 benchmark.

The performance speedup verified in the Intruder and the Vacation also confers to the results of [10] that evidence some opportunities for elision of transaction local barriers. From our analysis the Intruder instantiates an auxiliary linked list and a `byte[]` that can be elided with our capture analysis technique. On the other hand the Vacation performs three different kinds of operations, each of them including an initialization phase (non transactional) and the execution phase. The initialization instantiates several arrays with data for the parametrization of the actions performed by each transaction. To be considered as transaction local, we need to move the initialization of these parametrized arrays to inside the atomic function that defines the execution of the operation. We just run the Vacation with this modification when performing the full capture analysis, where the array barriers for the parametrized arrays are completely elided. For the other two cases, we keep the Vacation as its original definition, because the overhead of the arrays barriers in the initialization phase compromises to much the performance of this benchmark.

6. Related Work

The compiler over-instrumentation is one of the main reasons for the STM overheads and an obstacle to the scalability of programs synchronized with STMS. The use of unnecessary barriers on transaction-local memory access has an huge contribution to this behavior and in the past few years several solutions have been proposed to mitigate this problem.

One of the first contributions of Harris et al. [14], propose a direct access STM with a new decomposed interface that is used in the translation of the atomic blocks and exposed to the compiler, giving new opportunities for optimization. Analyzing the same problems, Yoo et al. [21] propose a new `tm_waiver` annotation to mark a function or block that would not be instrumented by the compiler for memory access—*waivered* code. In [17], they also rely in the programmer the responsibility of declaring which functions could avoid the instrumentation through the use of the annotation `tm_pure`.

The same approach has been followed in managed runtime environments, such as the work of Beckman et al. [3] that proposes the use of access permissions, via Java annotations, which can be applied to references, affecting the behavior of the object pointed by that reference. Carvalho et. al. [6] also proposed the use of Java annotations to identify the object fields and arrays that could be accessed directly, avoiding the STM barriers.

Using a different approach, Dragojevic et al. [10] propose a new technique for capture analysis. They provide this feature at runtime and also in the compiler using pointer analysis, which determines whether a pointer points to memory allocated inside the current transaction. Similar optimizations also appear in Wang et al. [20], and Eddon and Herlihy [11], which apply fully interprocedural analyses to discover thread-local data.

Afek et al. [1] integrates static analysis in Deuce to eliminate redundant reads and writes operations in transactional methods, including accesses to transaction local-data. Yet, the results presented in their work are far from the performance speedups of our solution.

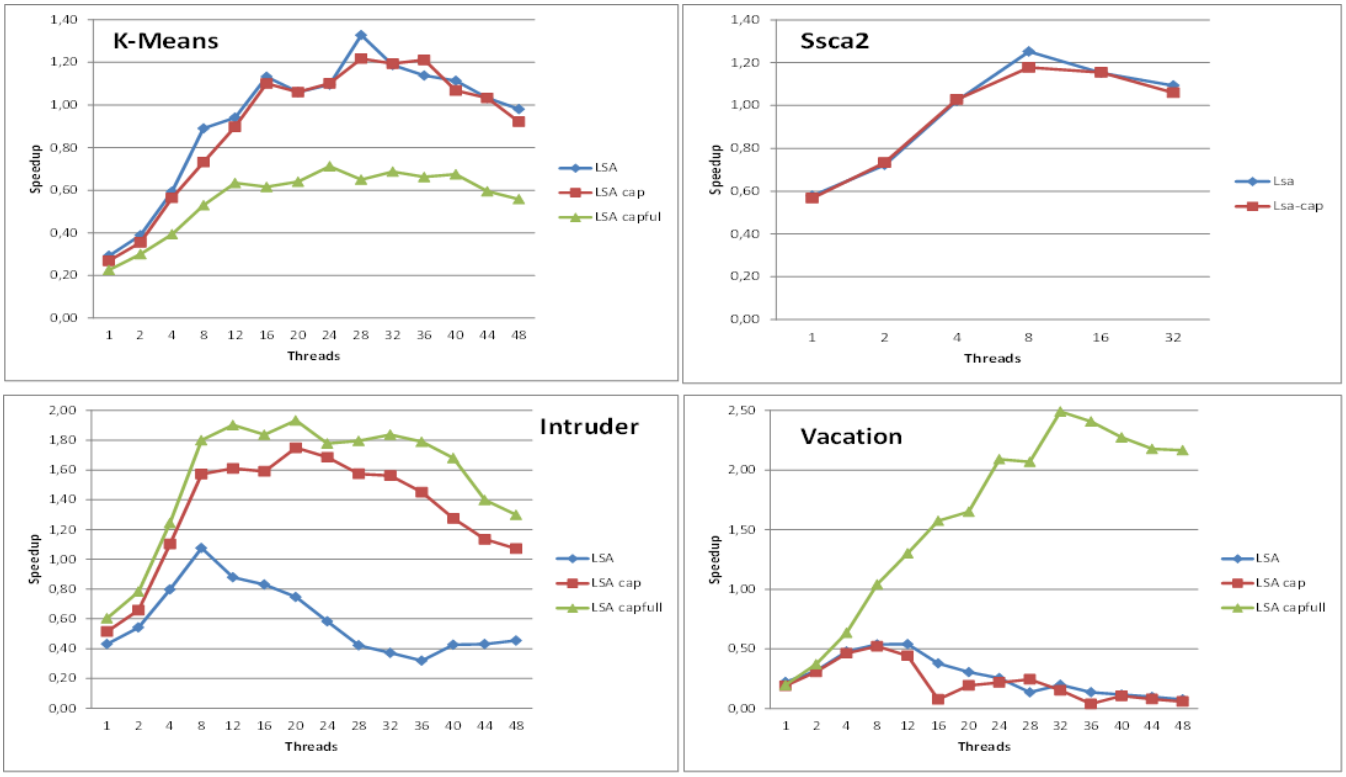

Figure 14. The STAMP benchmarks with the LSA STM

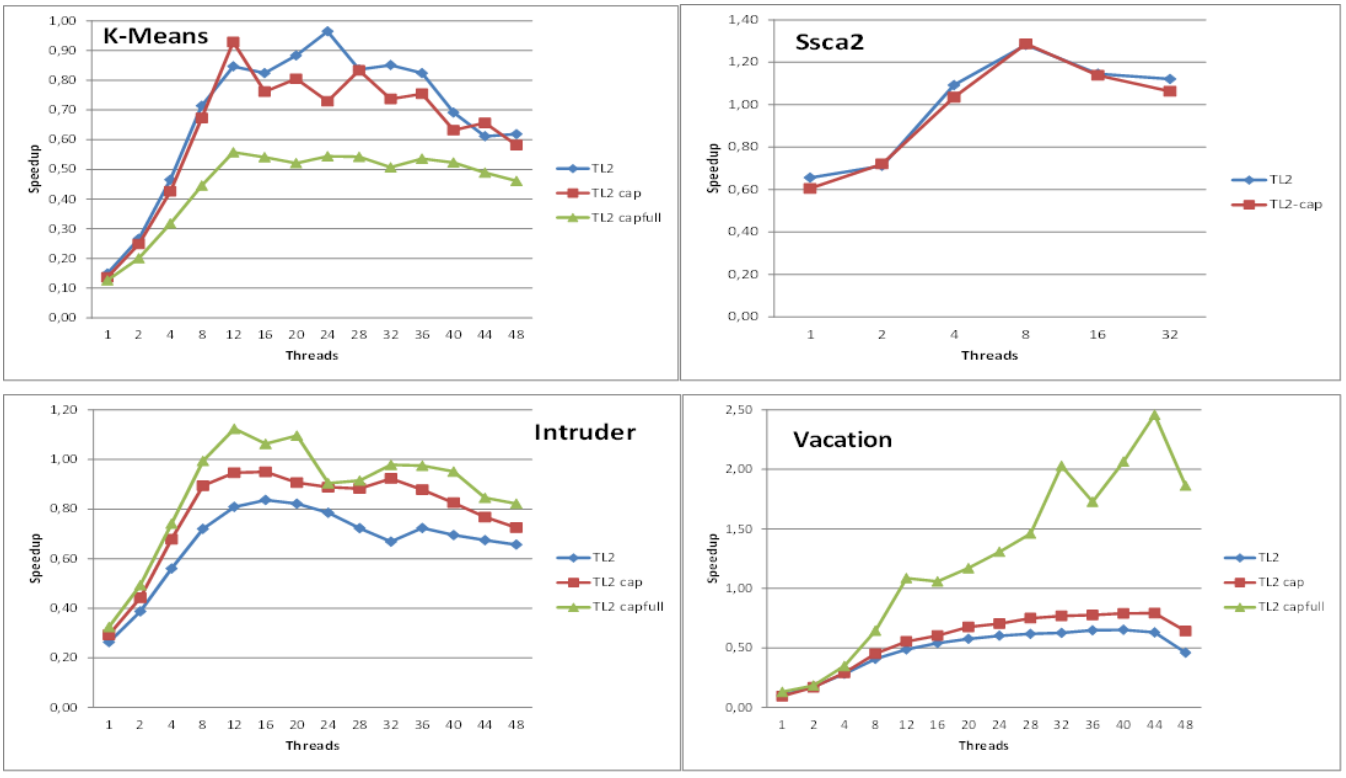

Figure 15. The STAMP benchmarks with the TL2 STM

7. Conclusions and Future Work

STMs are often criticized for introducing unacceptable overheads when compared with either the sequential version or a lock-based version of any realistic benchmark. Our experience in testing STMs with several realistic benchmarks, however, is that the problem stems from having instrumentation on memory locations that are not actually shared among transactions.

Several techniques have been proposed to elide useless STM barriers in programs automatically instrumented by STM compilers. From our analysis, the main contributions in this field follow three distinct approaches: 1) runtime capture analysis; 2) compiler static analysis to elide redundant operations; 3) decomposition of the STM APIs allowing the programmers to convey the knowledge about the blocks of instructions, or memory locations that should not be instrumented.

The later approach is more efficient and has proved bigger speedups in the STM performance, but has the inconvenient of reducing the transparency of the STMs APIs. Yet, from our knowledge, none of the existing solutions has demonstrated performance speedups with the same magnitude of the results that we present here for the StmBench7 and Vacation applications.

Our solution can solve one of the major bottlenecks that brakes the performance in many realistic applications and simultaneously preserve the transparency of an STM API, such as the Deuce framework. Although our technique adds a minor overhead in memory space to all transactional locations and also adds an extra indirection for arrays, however we still get a huge speedup in the Vacation and the StmBench7 benchmarks and in the later case, performing close to the medium-grain lock strategy.

In this work we did not include information about thread local objects, but the same approach could be followed to keep track of this information, which could be useful to optimize other tasks of an STM.

This work confirms our expectations and the importance of an STM engine having access to accurate information about the kind of locality of the transactional objects. Integrating this feature natively in a managed runtime could suppress the overheads of our solution and provide an important knowledge to the STM runtimes.

References

[1] Yehuda Afek, Guy Korland, and Arie Zilberstein, *Lowering stm overhead with static analysis*, LCPC2010, 2010.

[2] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha, *Memory models for open-nested transactions*, Proceedings of the 2006 workshop on Memory system performance and correctness (New York, NY, USA), MSPC '06, ACM, 2006, pp. 70–81.

[3] Nels E. Beckman, Yoon Phil Kim, Sven Stork, and Jonathan Aldrich, *Reducing stm overhead with access permissions*, IWACO (New York, NY, USA), ACM, 2009, pp. 2:1–2:10.

[4] Walter Binder, Jarle Hulaas, and Philippe Moret, *Advanced java bytecode instrumentation*, Proceedings of the 5th international symposium on Principles and practice of programming in Java (New York, NY, USA), PPPJ '07, ACM, 2007, pp. 135–144.

[5] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun, *STAMP: Stanford transactional applications for multi-processing*, IISWC '08, September 2008.

[6] Fernando Miguel Carvalho and João Cachopo, *STM with transparent API considered harmful*, ICA3PP '11, Springer, 2011.

[7] C. Cascaval, C. Blundell, M. M. Michael, H. W. Cain, S. Chiras P. Wu, and S. Chatterjee., *Software transactional memory: why is it only a research toy?*, vol. 51(11), Commun. ACM, 2008, pp. 40–46.

[8] D. Dice, O. Shalev, and N. Shavit, *Transactional locking II*, DISC, Proceedings of the 20th International Symposium on Distributed Computing, 2006, pp. 194–208.

[9] Aleksandar Dragojevic, Rachid Guerraoui, and Michal Kapalka, *Dividing Transactional Memories by Zero*, 2008, Transact, Salt Lake City, Utah, USA, 23.02.2008.

[10] Aleksandar Dragojevic, Yang Ni, and Ali-Reza Adl-Tabatabai, *Optimizing transactions for captured memory*, Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures (New York, NY, USA), SPAA '09, ACM, 2009, pp. 214–222.

[11] Guy Eddon and Maurice Herlihy, *Language support and compiler optimizations for stm and transactional boosting*, Proceedings of the 4th international conference on Distributed computing and internet technology (Berlin, Heidelberg), ICDCIT'07, Springer-Verlag, 2007, pp. 209–224.

[12] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides, *Design patterns: Abstraction and reuse of object-oriented design*, Proceedings of the 7th European Conference on Object-Oriented Programming (London, UK, UK), ECOOP '93, Springer-Verlag, 1993, pp. 406–431.

[13] Rachid Guerraoui, Michal Kapalka, and Jan Vitek, *Stmbench7: a benchmark for software transactional memory*, SIGOPS Oper. Syst. Rev. **41** (2007), 315–324.

[14] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi, *Optimizing memory transactions*, PLDI '06, ACM, 2006, pp. 14–25.

[15] Guy Korland, Nir Shavit, and Pascal Felber, *Noninvasive concurrency with java stm*, MultiProg 2010, 2010.

[16] Virendra Jayant Marathe and Mark Moir, *Toward high performance nonblocking software transactional memory*, PPoPP '08, ACM, 2008, pp. 227–236.

[17] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian, *Design and implementation of transactional constructs for c/c++*, Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications (New York, NY, USA), OOPSLA '08, ACM, 2008, pp. 195–212.

[18] Torvald Riegel, Pascal Felber, and Christof Fetzer, *A lazy snapshot algorithm with eager validation*, 2006, pp. 284–298.

[19] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg, *McRT-STM: a high performance software transactional memory system for a multi-core runtime*, PPoPP '06, ACM, 2006, pp. 187–197.

[20] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai, *Code generation and optimization for transactional memory constructs in an unmanaged language*, Proceedings of the International Symposium on Code Generation and Optimization (Washington, DC, USA), CGO '07, IEEE Computer Society, 2007, pp. 34–48.

[21] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee, *Kicking the tires of software transactional memory: why the going gets tough*, SPAA '08, ACM, 2008, pp. 265–274.