Parallel Dynamic Analysis on Multicores with Aspect-Oriented Programming

Danilo Ansaloni
Faculty of Informatics
University of Lugano
Switzerland
danilo.ansaloni@usi.ch

Walter Binder
Faculty of Informatics
University of Lugano
Switzerland
walter.binder@usi.ch

Alex Villazón
Faculty of Informatics
University of Lugano
Switzerland
alex.villazon@usi.ch

Philippe Moret
Faculty of Informatics
University of Lugano
Switzerland
philippe.moret@usi.ch

ABSTRACT

In most aspects, advice are synchronously executed by application threads, which may cause high overhead if advice execution is frequent or computationally expensive. When synchronous advice execution is not a necessity, asynchronous advice execution has the potential to parallelize program and advice execution on multicores. However, asynchronous advice execution requires communication between threads, causing some overhead. In order to mitigate such overhead, we introduce buffered advice, a new AOP mechanism for aggregating advice invocations in a thread-local buffer, which is processed when it is full. For asynchronous processing of full buffers, the communication overhead is paid only once per buffer, instead of once per advice invocation. We present an enhanced AOP programming model and framework based on AspectJ, which ease the use of buffered advice and support pluggable, custom buffer processing strategies. As case study, we optimize an existing aspect for data race detection using buffered advice. A thorough evaluation with standard benchmarks confirms that the use of buffered advice yields significant speedup on multicores.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*

General Terms

Algorithms, Languages, Measurement

Keywords

Parallelization, multicores, programming models, frameworks, aspect weaving, dynamic program analysis, data race detection

AOSD '10 Rennes, St. Malo, France

1. INTRODUCTION

Aspect-oriented programming (AOP) has been successfully used for building various software-engineering tools that perform some kind of dynamic program analysis; examples include profilers [27, 4], memory leak detectors [18, 37], data race detectors [10], and testing tools [38]. Typically, such aspect-based tools intercept a high number of join points, which may cause excessive overhead. For instance, the calling context profilers presented in [4] intercept the join points before and after each method execution, or the data race detector Racer [10] intercepts lock acquisition and release, as well as field access.

Most aspect-based tools execute advice bodies upon intercepted join points in a synchronous manner. That is, each program thread synchronously invokes advice while executing woven code. In many cases, advice execution involves relatively complex and time consuming operations. For example, in the case of Racer [10], advice executed upon field access involves lookup in a thread-safe hash table as well as a set intersection operation. As the code executed and the data structures manipulated by the advice respectively by the application are usually different, synchronous advice execution may impair locality both in application and in aspect code, resulting in increased cache miss rate and therefore lower performance. Furthermore, synchronous advice execution does not leverage idle CPU cores.

For many aspect-based software-engineering tools, synchronous advice execution is not a requirement; often, the computation performed in advice can be handled asynchronously, possibly in parallel with program execution using idle CPU cores. However, in practice, asynchronous advice execution at the granularity of individual advice invocations rarely pays off, because the communication overhead due to passing the relevant context information from one thread to another may outweigh the benefits thanks to parallelization of program and advice code. Communication between threads involves access to a shared data structure (e.g., a shared queue for passing the context information of invoked advice); thread-safety of the shared data structure incurs some overhead [12].

In this paper we introduce *buffered advice* in order to mitigate the communication overhead incurred by asynchronous advice execution. Advice invocations are aggregated in thread-local storage, until it pays off to execute the aggregated workload asynchronously. Upon invocation by woven code, buffered advice only store the relevant context information (e.g., references to static or dynamic join point instances, receiver or owner object, etc.) in a thread-local

buffer. When the buffer is full, the workload conveyed in the buffer is processed, that is, the respective advice bodies are executed using the buffered context information.

Buffered advice offers two important advantages: First, it allows for different buffer processing strategies in a flexible way. For example, a full buffer may be synchronously processed by the thread that has filled the buffer, or the full buffer may be passed to another thread in a pool of dedicated processing threads. In the latter case, buffered advice can be processed in parallel with program code on idle CPU cores. Second, upon invocation by woven code, buffered advice avoids any manipulation of the aspect's data structures. Hence, execution of application code is less disrupted by buffered advice invocation, which helps reduce the negative impact of advice invocation on locality in the woven code. Conversely, as the bodies of buffered advice are executed altogether when a buffer is processed, locality may be improved in the aspect code, if all buffered advice invocations execute the same code and operate on the same data structures of the aspect.

In this paper, we introduce a novel framework for buffered advice execution. It reconciles a simple and convenient programming model for expressing buffered advice, flexibility thanks to customizable processing of full buffers, as well as portability and compatibility with some prevailing AOP frameworks for the Java Virtual Machine (JVM). Regarding the programming model, the AspectJ language is extended with an annotation to mark buffered advice. A small API consisting of two interfaces and one class allows the aspect programmer to use different strategies for processing full buffers and to control the buffer size. Thanks to automated code generation, details regarding concrete buffer implementations are hidden from the programmer. Our framework is compatible with AspectJ; after aspect weaving with a standard AspectJ weaver, our framework applies code transformations to the woven code and generates a buffer implementation specialized for the woven aspect.

Our framework integrates support for *invocation-local variables*, a mechanism for passing data in local variables between advice woven in the same method body [36]. Buffered advice may read invocation-local variables, which can be part of the context information stored in the buffer. Hence, it is possible to pass arbitrary state, which is maintained by standard[1] advice, to buffered advice. Thus, invocation-local variables enable aspects that compose both standard and buffered advice.

The original, scientific contributions of this paper are twofold.

1. We introduce a novel AOP feature, buffered advice, as well as a framework for weaving buffered advice. Our framework integrates well with some existing AspectJ weavers for the JVM, such as ajc [16], MAJOR [37, 38], and HotWave [36].

2. We present a detailed case-study, where we adapt Racer [10], an existing aspect-based data race detector, to use buffered advice. A thorough performance evaluation using standard benchmarks confirms that using buffered advice significantly reduces the overhead caused by the aspect on a modern multicore machine.

This paper is structured as follows: Section 2 gives a short overview of invocation-local variables. Section 3 shows how buffered advice are specified using annotations. Section 4 explains the API for customized buffer processing. Section 5 details the weaving process, involving the automated generation of a buffer implementation. Section 6 presents our case study, aspect-based data race detection with Racer [10]. Section 7 provides a detailed performance evaluation for our case study. Section 8 compares our approach with related work. Section 9 discusses the pros and cons of buffered advice. Finally, Section 10 concludes this paper.

[1] In this paper, "standard advice" stands for "non-buffered advice", that is, synchronously invoked advice as supported by standard AspectJ.

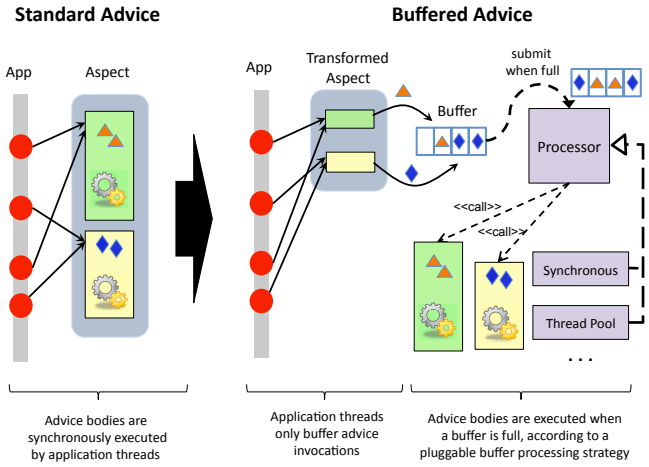

Figure 1: Standard advice execution (left) versus buffered advice execution (right)

2. BACKGROUND: INVOCATION-LOCAL VARIABLES

In this section, we briefly review *invocation-local variables* [36], which allow passing arbitrary state information (which is maintained by standard advice) to buffered advice. Invocation-local variables are important, because they offer a mechanism to buffer auxiliary information, which is not provided as context information by AspectJ.

Invocation-local variables enable data passing in local variables between advice that are woven in the same method body. The scope of an invocation-local variable is one invocation of a woven method. Invocation-local variables are accessed through public static fields that have `@InvocationLocal` annotations.

Aspects are first woven with the standard AspectJ weaver ajc [16]. Afterwards, program transformations inline those advice bodies that access invocation-local variables and map the invocation-local variables to local variables. Advice inlining is slightly complicated, because advice methods may take some arguments representing context information, such as static or dynamic join point instances. Our inlining algorithm determines the number and types of the advice arguments from the advice method's signature. The bytecodes that access the static fields corresponding to invocation-local variables are simply replaced with bytecodes for loading/storing from/in the local variables.

Since advice accessing invocation-local variables are inlined in woven methods, non-singleton aspect instances using `per*` clauses (e.g., per-object or per-control flow aspect association) are not supported in conjunction with invocation-local variables. In addition, the inlined advice must not access/invoke any non-public fields/methods of the aspect.

Each invocation-local variable is initialized at the beginning of a woven method with the value stored in the corresponding static field in the aspect, which is assigned only during execution of the aspect's static initializer. The Java memory model [13, 12] ensures that the value assigned by the static initializer is visible to all threads. As an optimization, the initialization of local variables

(a) Aspect with standard advice:

```
public aspect LogAspect {
  after() returning() : call(* *.*(..)) && !within(Log*) {
    Log.callWithArgs(thisJoinPoint, Thread.currentThread());
  }

  before(Object owner) : get(* *) && target(owner) &&
                         !within(Log*) {
    Log.fieldGet(owner, thisJoinPointStaticPart,
                 Thread.currentThread());
  }
}


// Slow logging operations (e.g., writing to a file or to a database)
public class Log {
  // Logs a method call; jp provides class name, method name, method
  //    signature, and the actual arguments; t is the calling thread
  public static void callWithArgs(JoinPoint jp, Thread t) { ... }

  // Logs read access to an instance field; owner is the object holding the accessed
  //    field; jpsp provides class and field name; t is the reading thread
  public static void fieldGet(Object owner,
                      JoinPoint.StaticPart jpsp, Thread t) { ... }
}
```

(b) Optimized aspect with buffered advice:

```
public aspect LogAspectBuf {
  @InvocationLocal
  public static Thread t;

  // Initialize the invocation-local variable t at the beginning of
  //    each method, constructor, or static initializer
  before() : ( execution(* *.*(..)) ||
               preinitialization(*.new(..)) ||
               staticinitialization(*)
             ) && !within(Log*) {
    t = Thread.currentThread();
  }

  @Buffered
  after() returning() : call(* *.*(..)) && !within(Log*) {
    Log.callWithArgs(thisJoinPoint, t);
  }

  @Buffered
  before(Object owner) : get(* *) && target(owner) &&
                         !within(Log*) {
    Log.fieldGet(owner, thisJoinPointStaticPart, t);
  }
}
```

Figure 2: Logging example: (a) aspect with standard advice, versus (b) aspect with buffered advice

corresponding to invocation-local variables in woven methods is skipped, if it can be statically determined that the first access is always a write.

We have implemented the program transformations in support of invocation-local variables as a separate weaver module that is compatible with ajc. We have successfully integrated that module both in MAJOR [37] and in HotWave [36].

3. SPECIFYING BUFFERED ADVICE

Figure 1 compares the execution of standard advice (left) with the execution of buffered advice (right). Upon invocation, buffered advice stores the relevant context information in a thread-local buffer. The stored information includes standard AspectJ context accessed in the buffered advice (e.g., static or dynamic join point instances, the currently executing object (`this`), the target object (`target`), the arguments (`args`), etc.), as well as invocation-local variables that are read in the buffered advice. Buffered advice must not write to any invocation-local variable, because the buffered advice body is executed later when the buffer is processed, possibly within a different calling context and possibly by a different thread.

In many cases, it is necessary that the parts of buffered objects, which are accessed in buffered advice bodies, are effectively immutable, in order to preserve unchanged information in the buffer for the execution of the buffered advice bodies. It is up to the programmer to ensure that relevant buffered data is effectively immutable. Some context information provided by AspectJ, which is automatically buffered when accessed in buffered advice, is effectively immutable (e.g., static join point instances), whereas other context information is mutable in general (e.g., dynamic join point instances, receiver or owner object, etc.). Our framework does not automatically create any deep copies of buffered objects.

A buffer is submitted to a processor when it is full; the processor then iteratively executes the bodies of the advice invocations conveyed in the buffer, passing the context information stored in the buffer as input data. The processor is a pluggable component that implements a particular processing strategy. For example, submitted buffers may be synchronously processed by the calling thread, or they may be passed to a thread pool executor for asynchronous processing.

The `LogAspect` example in Figure 2(a) logs method calls after normal completion of the callee (including class name, method name, method signature, actual arguments, and calling thread), as well as read access to instance fields (including field owner object, class and field name, and reading thread). As logging may be slow, involving some I/O, `LogAspect` may cause excessive overhead.

If the actual logging can be performed at a later moment, buffered advice allows us to aggregate logging requests in a thread-local buffer for later processing when the buffer is full. Figure 2(b) illustrates the optimized `LogAspectBuf` using buffered advice. The advice for logging are annotated with `@Buffered`, indicating that upon advice invocation, the relevant context information shall be buffered for later execution of the corresponding advice bodies.

Below we show the definition of the `@Buffered` annotation. The associated retention policy ensures that `@Buffered` annotations are preserved in the bytecode of compiled aspect classes, which are subsequently inspected by our weaver for buffered advice.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.CLASS)
public @interface Buffered {}
```

All AspectJ context information accessed in the buffered advice is automatically stored in the buffer (`thisJoinPoint` for the first advice body, respectively `owner` and `thisJoinPointStaticPart` for the second advice body). The current thread must be also stored in the buffer, since a different thread may later process the buffer (depending on the concrete processor implementation). Therefore, we store the current thread in the invocation-local variable `t` at the beginning of each method, constructor, or static initializer. As the buffered advice read the invocation-local variable, the value of `t` is part of the context information to be stored in the buffer.

Before discussing how `LogAspectBuf` is transformed and woven by our framework in Section 5, we explain how custom processors are plugged into our framework in the next section.

4. API

Our framework provides a small, flexible API for customizing buffer processing and managing buffer size. As illustrated in Figure 3, the API consists of two interfaces and one class.

The interface `Buffer` represents buffers that are submitted to a processor, represented by the `Processor` interface. For an as-

```
public interface Buffer {
  void process();
}

public interface Processor {
  void submit(Buffer buf);
}

public final class Control {
  public static Processor getProcessor() { ... }

  public static int getDefaultBufferSize() { ... }
  public static void setDefaultBufferSize(int size) { ... }
  public static int getBufferSize(Thread t) { ... }
  public static void setBufferSize(Thread t, int size) { ... }

  public static void submitCurrentBuffer() { ... }
  public static void submitTerminatedThreadBuffers() { ... }
}
```

Figure 3: Framework API

```
public class SynchronousProc implements Processor {
  public void submit(Buffer buf) { buf.process(); }
}


public class ThreadPoolProc implements Processor {
  private static final int NR_THREADS = ... ;

  private final Executor exec =
    Executors.newFixedThreadPool(NR_THREADS);

  public void submit(final Buffer buf) {
    exec.execute(new Runnable() {
                   public void run() { buf.process(); }
                 } );
  }
}
```

Figure 4: Two example Processor implementations

pect using buffered advice, our framework automatically generates a `Buffer` implementation, which is transparent to the aspect programmer (see Section 5). While our framework supports weaving with multiple aspects that may make use of buffered advice and of common invocation-local variables, we focus on weaving with a single aspect in order to simplify the presentation.

Method `process()` in the `Buffer` interface executes all advice bodies corresponding to the advice invocations conveyed in the buffer; this method is invoked by a processor. Until submission to a processor, each buffer[2] is thread-local to the thread that has filled the buffer; that is, each buffer conveys advice invocations from only a single thread.

In contrast to the automatically generated `Buffer` implementation, the `Processor` implementation has to be provided by the programmer. For convenience, our framework includes several general-purpose `Processor` implementations that are suited for many applications of buffered advice. Method `submit(Buffer)` is invoked (typically when a buffer is full) to initiate the processing of the passed buffer. Upon buffer submission, a processor may synchronously invoke `process()` on the passed buffer by the calling thread, or it may pass the buffer to another thread for processing. At the end of this section, we will show two example `Processor` implementations. In our framework, there is a singleton processor; the implementation class is configured with a system property.

The class `Control` provides some static methods for accessing the singleton processor, for managing the buffer size, and for forcing the immediate submission of certain buffers. The methods in class `Control` may be used by the aspect programmer to influence buffer processing at runtime. Method `getProcessor()` returns the singleton processor, which may offer a processor-specific API for changing the buffer processing strategy at runtime. The default buffer size, which must be a positive value and can be queried with method `getDefaultBufferSize()`, is initially specified through a system property, and may be changed at runtime with method `setDefaultBufferSize(int)`. For each thread, the buffer size may be individually changed with method `setBufferSize(Thread,int)`; a positive value will take precedence over the default buffer size for the specified thread, while a non-positive value means that the default buffer size will apply for the given thread. Method `getBufferSize(Thread)` returns the current buffer size for a thread (which may be the default buffer size). Changing the buffer size for a thread affects subsequent buffer allocations by the thread; it does not change the size of the thread's current buffer. Method `submitCurrentBuffer()` causes immediate submission of the current thread's buffer, independently of its filling state; afterwards, a new buffer is created.

Our framework maintains a table to keep track of all threads in the system that have allocated a buffer. Periodically, a dedicated polling thread of the framework checks the state of the threads in the table in order to detect terminated threads. The final buffer of each terminated thread t is submitted to the processor, and t is removed from the table. The Java memory model [13, 12] guarantees that the polling thread sees the effects of all assignments made by t, after it has detected the termination of t. Hence, the polling thread is guaranteed to see the up-to-date final buffer of t. In order to force the check for terminated threads programmatically, class `Control` offers the method `submitTerminatedThreadBuffers()`.

Figure 4 illustrates two example `Processor` implementations. The first implementation synchronously processes the submitted buffers by the calling thread, whereas the second implementation leverages Java's executor framework (package `java.util.concurrent` [12]) for asynchronous buffer execution using a thread pool with a fixed number of processing threads.

5. WEAVING BUFFERED ADVICE

In this section we illustrate how our framework transforms and weaves buffered advice, using the `LogAspectBuf` example of Figure 2(b).

Our framework is implemented as code transformations applied to code that has previously been woven with ajc. In this way, we avoid modifying ajc and ensure that our framework also works in conjunction with other ajc-based tools, such as the aspect weaver MAJOR [37] and the dynamic AOP framework HotWave [36]. Similar to the code transformations in support of invocation-local variables [36], our framework heavily relies on inlining techniques. As ajc is not aware of `@Buffered` annotations, ajc weaves buffered advice in the same way as standard advice, that is, it compiles advice into methods and inserts invocations to the advice methods in woven code.

Our framework analyzes the compiled aspect in order to detect `@Buffered` and `@InvocationLocal` annotations. It generates a `Buffer` implementation tailored to the buffered advice in the aspect, taking the context information used by the buffered advice into account (including invocation-local variables read by the buffered advice). In the woven code, it replaces each invocation of an advice method with an invocation of a corresponding method in the generated `Buffer` implementation that appends the required context information to the current thread's buffer. The `process()`

[2] In the following text, "processor" stands for "`Processor` instance", and "buffer" stands for "`Buffer` instance".

```
public final class BufferImpl implements Buffer {
  private final int bufSize =
    Control.getBufferSize(Thread.currentThread());
  private final int[] adviceID = new int[bufSize];
  private final Object[] owner = new Object[bufSize];
  private final JoinPoint[] jp = new JoinPoint[bufSize];
  private final JoinPoint.StaticPart[] jpsp =
                  new JoinPoint.StaticPart[bufSize];
  private final Thread[] thr = new Thread[bufSize];
  private int nextFree = 0;

  // Returns the current thread's buffer, creating it if it does not yet exist
  private static BufferImpl currentBuffer() {
    Thread t = Thread.currentThread();
    BufferImpl buf = (BufferImpl)t.currentBuf;
    if (buf == null) t.currentBuf = buf = new BufferImpl();
    return buf;
  }

  // If this buffer is full, it is submitted to the processor and a new buffer is created
  private void submitIfFull() {
    if (nextFree == bufSize) {
      Control.getProcessor().submit(this);
      Thread.currentThread().currentBuf = new BufferImpl();
    }
  }

  // Buffers an invocation of the first advice method
  public static void appendAdv_1(JoinPoint jp, Thread t) {
    BufferImpl buf = currentBuffer();
    buf.adviceID[buf.nextFree] = 1;
    buf.jp[buf.nextFree] = jp;
    buf.thr[buf.nextFree++] = t;
    buf.submitIfFull();
  }

  // Buffers an invocation of the second advice method
  public static void appendAdv_2(Object owner,
                    JoinPoint.StaticPart jpsp, Thread t) {
    BufferImpl buf = currentBuffer();
    buf.adviceID[buf.nextFree] = 2;
    buf.owner[buf.nextFree] = owner;
    buf.jpsp[buf.nextFree] = jpsp;
    buf.thr[buf.nextFree++] = t;
    buf.submitIfFull();
  }

  // Iteratively processes all buffered advice invocations
  public void process() {
    for (int i = 0; i < nextFree; i++) {
      switch (adviceID[i]) {
        case 1: // execute advice body 1
          Log.callWithArgs(jp[i], thr[i]); break;
        case 2: // execute advice body 2
          Log.fieldGet(owner[i], jpsp[i], thr[i]); break;
        default:
          assert false;
      }
    }
  }
}
```

Figure 5: Automatically generated Buffer implementation for LogAspectBuf in Figure 2(b)

method (see Figure 3) is automatically generated, too. It iterates through the entries in the buffer and executes the corresponding advice bodies.

Figure 5 shows `BufferImpl`, the automatically generated `Buffer` implementation for `LogAspectBuf` in Figure 2(b). The constructor first obtains the buffer size through the `Control` API (see Section 4). `BufferImpl` stores the buffer contents in several arrays. The integer array `adviceID` stores unique IDs representing the invoked buffered advice (i.e., the buffered advice bodies in the compiled aspect are numbered). If the aspect contains only a single buffered advice, the array `adviceID` is useless and therefore not generated. The other arrays store the context information used by the buffered advice bodies. The element types of these arrays correspond to the types of context information accessed in the buffered advice. If multiple buffered advice bodies access context information of the same type, it is stored in the same array. For instance, in the `LogAspectBuf` example, both buffered advice bodies read an invocation-local variable of type `Thread`; in `BufferImpl`, there is only the single array `thr` for storing thread references. In general, for each type T, the generated `Buffer` implementation has n_T arrays with element type T, where n_T is the maximum number of context variables of type T accessed within a single buffered advice body.

The static helper method `currentBuffer()` returns the buffer of the current thread. When a thread calls this method for the first time, a buffer is created and stored in the thread-local variable `currentBuf`. For performance reasons, as access to `currentBuf` is very frequent (each invocation of buffered advice causes read access to `currentBuf`), our framework implements the thread-local variable as an instance field directly inserted into class `Thread` (instead of using Java's `ThreadLocal` API).[3] The helper method `submitIfFull()` submits the current buffer to the processor, if the buffer is full. It then creates a new buffer and stores it in the thread-local variable `currentBuf`.

For each buffered advice body with the assigned unique integer ID i, the public static method `appendAdv_`i`(...)` is generated, which takes the context information required by the advice body as arguments and stores it in the buffer. Upon transformation of woven code by our framework, each invocation of the buffered advice method i is replaced with an invocation of `appendAdv_`i`(...)`.

The generated `process()` method, to be invoked by the processor, iterates over the buffer entries and executes the corresponding advice bodies, taking the needed context information from the buffer. Because the buffered advice bodies are directly inlined within the `process()` method body, buffered advice must not use any non-public members of the aspect.

Figure 6 illustrates weaving of buffered advice with a small example. Weaving class `Foo` in Figure 6(a) with `LogAspectBuf` in Figure 2(b) results in the woven code in Figure 6(b). For the sake of easy readability, we present the woven class `Foo` in pseudo-code, using the AspectJ pseudo-variables `thisJoinPoint` and `thisJoinPointStaticPart`. In the woven code, at the beginning of methods `f()` and `g()`, the invocation-local variable `t` is correctly initialized with the current thread. An invocation of the generated static method `appendAdv_1(...)` respectively `appendAdv_2(...)` is inserted after each method invocation, respectively before each read access to an instance field. The relevant context information to be stored in the buffer is passed as arguments to `appendAdv_1(...)` and `appendAdv_2(...)`.

6. CASE STUDY: APPLYING BUFFERED ADVICE TO RACER

Racer [10] is an effective data race detection tool based on AspectJ. Racer extends the Eraser [30] algorithm to detect data races in multi-threaded Java programs. The work on Racer [10] has resulted in several important contributions to the AOP community, such as an AspectJ extension to support `lock()` and

[3]We have successfully tested this extension with many state-of-the-art JVMs (e.g., Sun's HotSpot VMs or IBM's J9 VMs) on various platforms. However, the field `currentBuf` has the declared type `Object`, so as not to introduce any new class dependency into the `Thread` class, which otherwise could break JVM bootstrapping. Therefore, the method `currentBuffer()` in the generated `BufferImpl` class has to downcast the buffer reference read from `currentBuf`.

(a) Original code:

```
public class Foo {
  private int c;

  void g() { c++; }
  void f() { g(); c--; }
}
```

(b) Woven and transformed code (simplified pseudo-code):

```
public class Foo {
  private int c;

  void g() {
    Thread t = Thread.currentThread();

    BufferImpl.appendAdv_2(this, thisJoinPointStaticPart, t);
    c++;
  }

  void f() {
    Thread t = Thread.currentThread();

    g();
    BufferImpl.appendAdv_1(thisJoinPoint, t);

    BufferImpl.appendAdv_2(this, thisJoinPointStaticPart, t);
    c--;
  }
}
```

Figure 6: Example code (a) before weaving, respectively (b) after weaving with LogAspectBuf in Figure 2(b)

`unlock()` pointcuts that match entering respectively exiting synchronized blocks or synchronized methods. Another contribution is the `maybeShared()` pointcut, an optimization based on static thread-local object analysis, which matches only access to shared fields (i.e., fields that are possibly accessed by multiple threads).[4]

Racer reports a data race, if two or more threads access the same field without holding any common lock, and if at least one of these threads is writing to the field. With each field f, Racer associates a set of locks $L(f)$ and a finite-state machine to keep track of read and write operations performed by different threads. The first time f is accessed, $L(f)$ is initialized with all the locks held by the accessing thread. Subsequently, each time f is accessed, $L(f)$ is replaced by the intersection of the locks in $L(f)$ and the locks held by the accessing thread. If $L(f)$ becomes empty and the field has been accessed by more than one thread, at least one of them having performed a write access, a race is reported.

Since field access is frequent and because the operations performed by Racer upon each field access are rather computationally expensive, involving access to thread-safe data structures and the intersection operation, Racer can cause very high overhead. Therefore, the aspect-based Racer implementation is a good candidate for optimization with buffered advice.

Our case study is based on the aspect-based Racer implementation `RacerAJ` (version 1.0)[5]. First, we corrected four problems that prevented us from successfully applying `RacerAJ` to realistic workloads, such as the DaCapo benchmarks [6].

1. A race condition within the code that updated the finite-state machines sometimes caused incorrected state transitions. The state associated with a given field was not retrieved and updated atomically. We fixed the race condition by proper synchronization.

2. Lookup of locks relied on equality and on the hash code of objects. However, two distinct objects that are equal (according to an overridden `equals(Object)` method) have different locks. We fixed this issue by replacing the affected data structures with custom implementations that rely on object identity and on the identity hash code as provided by `System.identityHashCode(Object)`.

3. Objects with dynamically changing hash code[6] were used as keys in hash tables. Thus, lookup may fail if the key's hash code has changed after insertion in the hash table. The aforementioned correction, replacing equality-based data structures with identity-based data structures, also solved this issue.

4. Memory leaks due to non-reclaimed table entries caused `OutOfMemoryError` in some benchmarks. While objects allocated by application code could be reclaimed by the garbage collector thanks to the use of weak references, table entries corresponding to reclaimed objects were not cleared. We solved this problem by storing additional information (i.e., the table entry to clear) in each weak reference. A dedicated thread processes the weak references that have been cleared by the garbage collector (using a reference queue as provided in the package `java.lang.ref`).

Figure 7 shows part of the Racer implementation as described in [10]. The `Locking` aspect captures the acquisition and release of locks using the `lock()` and `unlock()` pointcut designators. The locks held by a thread are maintained in the thread-local bag `locksHeld`. Since in Java locks are reentrant, a thread may acquire the same lock multiple times without blocking. Hence, it is necessary to keep track of the number of times a lock has been acquired; that is, the held locks must be kept in a bag, and not in a set. The `Racer` aspect captures every field access and uses the context information provided by AspectJ's `thisJoinPointStaticPart` pseudo-variable in order to be able to identify the location of data races. Method `fieldSet(...)` (not shown in the figure) keeps track of the object where a field has been accessed (the `owner` reference; static fields are considered part of the type's `Class` instance), checks which threads have accessed the field, updates the state machine associated with the field, and computes the lock intersection in order to detect data races.

In contrast to the original `Racer` aspect described in [10], the aspect in Figure 7 makes access to the thread-local variable `locksHeld` and to the current thread explicit. This information is passed as extra arguments to `fieldSet(...)`. In the original implementation, this information is not passed as arguments, but accessed inside the body of `fieldSet(...)`, which is functionally equivalent. However, explicitly showing access to `locksHeld` and to the current thread eases comparison of the original `Racer` aspect with our optimized implementation using buffered advice, by emphasizing the input data needed for data race detection.

The pointcuts that intercept field accesses need not be executed synchronously by the advice-invoking program threads. If all relevant context information, including the thread's bag of held locks, remains unchanged after having been stored in the buffer, the data

[4]The `lock()` and `unlock()` pointcuts can be enabled in the standard AspectJ compiler ajc and in the AspectBenchCompiler *abc* [3] through a special option. The `maybeShared()` pointcut is only available in *abc*.

[5]`http://www.bodden.de/tools/raceraj/`

[6]For example, Java collections override the `hashCode()` method so as to compute the collection's hash code based on the hash codes of the contained objects. Hence, inserting or removing an element in a collection is likely to change the collection's hash code.

```
public aspect Locking {
  ThreadLocal<Bag> locksHeld = new ThreadLocal<Bag>() {
    protected Bag initialValue() { return new HashBag(); }
  };

  before(Object l) : lock() && args(l) && Racer.scope() {
    Bag locks = locksHeld.get();
    locks.add(l);
  }

  after(Object l) : unlock() && args(l) && Racer.scope() {
    Bag locks = locksHeld.get();
    locks.remove(l);
  }
}

public aspect Racer {
  pointcut staticFieldSet() : set(static * *) && maybeShared();
  pointcut fieldSet(Object owner) :
    set(!static * *) && target(owner) && maybeShared();
  pointcut staticFieldGet() : get(static * *) && maybeShared();
  pointcut fieldGet(Object owner) :
    get(!static * *) && target(owner) && maybeShared();

  pointcut scope() : ... ; // restricts the scope to avoid infinite recursions

  before() : staticFieldSet() && scope() {
    JoinPoint.StaticPart jpsp = thisJoinPointStaticPart;
    String id = jpsp.getSignature().toLongString().intern();
    Class owner = jpsp.getSignature().getDeclaringType();
    SourceLocation loc = jpsp.getSourceLocation();
    fieldSet(owner, id, loc,
             Locking.aspectOf().locksHeld.get(),
             Thread.currentThread());
  }

  before(Object owner) : fieldSet(owner) && scope() {
    JoinPoint.StaticPart jpsp = thisJoinPointStaticPart;
    String id = jpsp.getSignature().toLongString().intern();
    SourceLocation loc =  jpsp.getSourceLocation();
    fieldSet(owner, id, loc,
             Locking.aspectOf().locksHeld.get(),
             Thread.currenThread());
  }

  ... // similar advice for the pointcuts staticFieldGet and fieldGet
}
```

Figure 7: Original Locking and Racer aspects (for details, see [10])

race detection algorithm may be executed at a later moment. Since the bag is updated upon each lock acquisition and release, we need a kind of multi-version bag, such that effectively immutable versions of the bag can be stored in the buffer. While this approach will correctly detect the occurred data races, it may report them later than the original Racer implementation (upon buffer processing). Full buffers can be processed out-of-order and in parallel, because the intersection operation applied to the set of locks associated with an accessed field is both associative and commutative.

Figure 8 presents our optimized `RacerBuf` aspect. In order to better show the use of invocation-local variables, we integrated the functionalities of the previous `Locking` and `Racer` aspects into a single `RacerBuf` aspect. `RacerBuf` defines two invocation-local variables, `ilThread` and `ilLocksHeld`, which are needed for preserving the current thread and its bag of held locks in the buffer. This information must be preserved in the buffer, since the advice bodies will be executed later upon buffer submission, possibly by a different thread (depending on the installed processor). The first (standard) advice in Figure 8 correctly initializes the invocation-local variables at the beginning of each method, constructor, or static initializer. All context information used by the buffered advice that capture field access is stored in the buffer, including the static join point instance and the invocation-local variables `ilThread` and `ilLocksHeld`. In addition, for the buffered advice capturing instance field access, the `owner` object is stored as well. In order to ensure that the bag versions stored in the buffer are effectivly immutable, the current thread's bag is cloned before each update. That is, the (standard) advice capturing lock acquistion respectively lock release perform copy-on-write. Thanks to buffered advice, the computationally expensive data race detection algorithm is performed upon buffer processing, possibly in parallel with the execution of application threads.

```
public aspect RacerBuf {
  @InvocationLocal public static Bag ilLocksHeld;
  @InvocationLocal public static Thread ilThread;

  ThreadLocal<Bag> locksHeld = new ThreadLocal<Bag>() {
    protected Bag initialValue() { return new HashBag(); }
  };

  before() : ( execution(* *.*(..)) ||
               preinitialization(*.new(..)) ||
               staticinitialization(*)
             ) && scope() {
    ilLocksHeld = locksHeld.get();
    ilThread = Thread.currentThread();
  }

  before(Object l) : lock() && args(l) && scope() {
    ilLocksHeld = ilLocksHeld.clone(); // create new version of the bag
    locksHeld.set(ilLocksHeld);
    ilLocksHeld.add(l);
  }

  after(Object l) : unlock() && args(l) && scope() {
    ilLocksHeld = ilLocksHeld.clone(); // create new version of the bag
    locksHeld.set(ilLocksHeld);
    ilLocksHeld.remove(l);
  }

  pointcut staticFieldSet() : set(static * *) && maybeShared();
  pointcut fieldSet(Object owner) :
    set(!static * *) && target(owner) && maybeShared();
  pointcut staticFieldGet() : get(static * *) && maybeShared();
  pointcut fieldGet(Object owner) :
    get(!static * *) && target(owner) && maybeShared();

  pointcut scope() : ... ; // restricts the scope to avoid infinite recursions

  @Buffered
  before() : staticFieldSet() && scope() {
    JoinPoint.StaticPart jpsp = thisJoinPointStaticPart;
    String id = jpsp.getSignature().toLongString().intern();
    Class owner =  jpsp.getSignature().getDeclaringType();
    SourceLocation loc = jpsp.getSourceLocation();
    fieldSet(owner, id, loc, ilLocksHeld, ilThread);
  }

  @Buffered
  before(Object owner) : fieldSet(owner) && scope() {
    JoinPoint.StaticPart jpsp = thisJoinPointStaticPart;
    String id = jpsp.getSignature().toLongString().intern();
    SourceLocation loc =  jpsp.getSourceLocation();
    fieldSet(owner, id, loc, ilLocksHeld, ilThread);
  }

  ... // similar buffered advice for the pointcuts staticFieldGet and fieldGet
}
```

Figure 8: Optimized RacerBuf aspect using buffered advice

7. EVALUATION

In this section we evaluate the benefits of buffered advice using the Racer case study presented in Section 6. We explore the performance impact of buffered advice and of different buffer processing strategies. In addition, we investigate the impact of different buffer sizes on performance.

Table 1: Execution times and overhead factors for the corrected and the optimized versions of Racer (standard advice), as well as for RacerBuf with SynchronousProc and with ThreadPoolProc (buffered advice)

	Orig.	Racer				RacerBuf			
		Corrected		Optimized		SynchronousProc		ThreadPoolProc	
DaCapo	**[s]**	**[s]**	**ovh**	**[s]**	**ovh**	**[s]**	**ovh**	**[s]**	**ovh**
antlr	0.09	112.32	1248.00	3.09	34.33	2.72	30.22	1.17	13.00
bloat	0.48	1549.33	3227.77	4.84	10.08	4.07	8.48	1.60	3.33
chart	0.42	243.86	580.62	4.56	10.86	3.86	9.19	1.46	3.48
eclipse	1.44	>1h		77.02	53.49	56.66	39.35	15.16	10.53
fop	0.11	65.90	599.09	1.56	14.18	1.44	13.09	0.63	5.73
hsqldb	0.65	227.16	349.48	19.50	30.00	19.26	29.63	9.70	14.92
jython	0.22	>1h		27.31	124.14	24.09	109.50	6.39	29.05
luindex	0.15	481.95	3213.00	13.15	87.67	12.88	85.87	6.91	46.07
lusearch	0.33	3504.39	10619.36	130.83	396.45	38.42	116.42	14.23	43.12
pmd	0.03	11.48	382.67	0.42	14.00	0.39	13.00	0.19	6.33
xalan	0.35	2411.51	6890.03	35.58	101.66	20.05	57.29	7.55	21.57
geomean	0.25			10.11	40.44	7.77	31.08	3.07	12.28

All measurements are collected on a quad-core machine (Intel Xeon CPU E7340, 2.4 GHz, 16 GB RAM) running CentOS Enterprise Linux 5.3 with the Sun JDK 1.6.0_16 Hotspot Server VM (64 bit). We use MAJOR [37] with AspectJ[7] version 1.6.5 in order to weave the `Racer` and `RacerBuf` aspects.

For our experiments, we use the DaCapo benchmark suite (dacapo-2006-10-MR2) [6]. All measurements are collected with 'small' workload size. We report the median of 15 runs within the same JVM process, in order to attenuate the perturbations due to class-loading, load-time instrumentation, and just-in-time compilation. We also report average execution time and overhead factor for the benchmark suite, which we compute as the geometric mean ("geomean").

We do not use the `maybeShared()` pointcut designator, because it is currently only supported by *abc*. Note that compile-time weaving with *abc* would not be suitable for the DaCapo benchmarks, since certain benchmark classes could not be woven. For example, "jython" dynamically generates classes at runtime, and "eclipse" uses classes that are provided within plugin archives. As the `maybeShared()` optimization would be beneficial both for `Racer` and for `RacerBuf`, excluding that optimization does not favor any particular setting. In the future, it would be interesting to integrate support for buffered advice also in *abc* in order to study the combined impact of buffered advice and the `maybeShared()` optimization, which are complementary.

With `RacerBuf`, a benchmark may complete before all pending buffers have been processed. In order to ensure that the ending time is not taken prematurely, we extend the benchmark harness to wait in each run before the ending time is taken until all pending buffers and the final buffers of all terminated threads have been processed.

Table 1 shows the execution times and overhead factors ("ovh") for `Racer` (standard advice) and for `RacerBuf` (buffered advice). The column "Orig." corresponds to the original DaCapo benchmarks without any aspect. The columns "Racer Corrected" correspond to the original `Racer` implementation with the four corrections described in Section 6. The overhead caused by the corrected `Racer` may even exceed factor 10 000; we consider such excessive overhead prohibitive for analyzing realistic workloads. We stopped measurements when the execution time exceeded one hour (">1h" in Table 1). The high overhead is not due to our corrections; the original `Racer` implementation incurs similar overhead when it does not fail.

In order to reduce overhead, we refactored the `Racer` code and optimized the internal data structures without changing the algorithm. For instance, we reduced the number of allocated objects (notably in the state machine associated with each field and in the bag of locks), and we optimized set operations and iterators. Furthermore, in order to reduce contention, we applied lock striping [12] to the shared table that maintains the state of accessed fields. The columns "Racer Optimized" in Table 1 correspond to the resulting optimized `Racer` implementation (which only uses standard advice). The optimizations result in considerable overhead reductions; in some cases, the overhead is reduced by two orders of magnitude. On average, the optimized `Racer` incurs an overhead of factor 40.44. To a large extend, the overhead reductions are due to significantly reduced memory consumption, which helps avoid swapping by the operating system and reduces garbage collection time.

The optimized `Racer` implementation serves as baseline for comparison with `RacerBuf`, which internally uses the same optimized data structures. We evaluate `RacerBuf` with two buffer processing strategies, `SynchronousProc` and `ThreadPoolProc` as shown in Figure 4. In all settings with buffered advice, we use a buffer size of 1 024 for all program threads. We choose this value, as it results in a proper balance between good performance and moderate memory consumption. Figure 9 shows overhead factors for `RacerBuf` with `ThreadPoolProc` (geometric mean for the DaCapo suite) depending on the buffer size. We configured `ThreadPoolProc` to use a fixed pool with four threads, which is an appropriate choice on our quad-core machine.

For both buffer processing strategies, `RacerBuf` outperforms the optimized `Racer`. On average, `RacerBuf` causes an overhead factor of 31.08 with `SynchronousProc`, respectively an overhead factor of 12.28 with `ThreadPoolProc`.

Figure 10 illustrates the speedup obtained over the optimized `Racer` thanks to buffering and parallelization. In particular, the speedup of `RacerBuf` with `SynchronousProc` is due to buffering, whereas the speedup of `RacerBuf` with `ThreadPoolProc` is due to buffering and parallelization. On average, the speedup of `RacerBuf` over the optimized `Racer` is a factor of 1.30 with `SynchronousProc`, respectively a factor of 3.29 with `ThreadPoolProc`. The fact that `RacerBuf` with `Synchronous-`

[7] `http://www.eclipse.org/aspectj/`

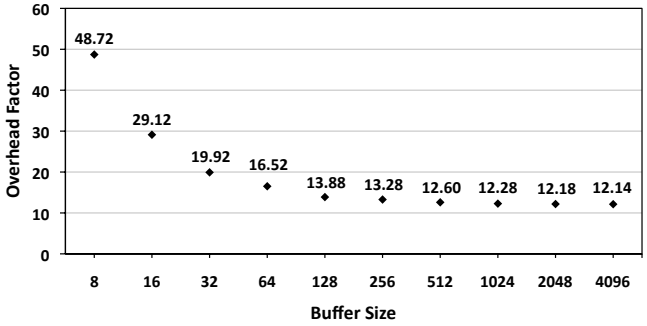

Figure 9: Overhead factor for RacerBuf with ThreadPoolProc depending on the buffer size

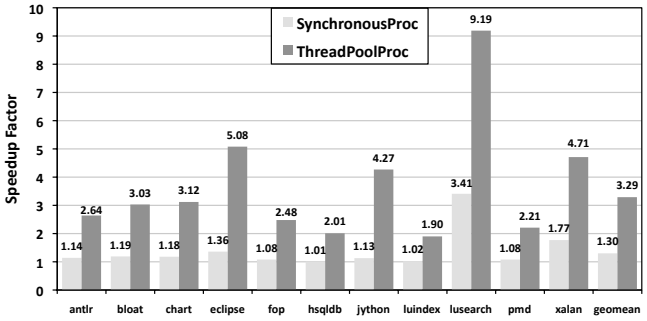

Figure 10: Speedup of RacerBuf with SynchronousProc and with ThreadPoolProc over the optimized Racer

`Proc` outperforms the optimized `Racer` suggests that buffering improves locality.

In summary, our evaluation shows that buffered advice can be an important optimization for reducing the overhead of aspect-based dynamic analysis tools on multicores. It helps improve locality and allows exploiting idle CPU cores. Buffered advice is complementary to traditional code optimizations, such as reducing the number of object instances within data structures, lock striping, etc.

8. RELATED WORK

In AspectJ, threads in the base program synchronously execute advice; the base program does not make any progress before the execution of the advice body has completed. Similarly, the standard approach to procedural reflection [31] results in the base-level being suspended when the meta-level is active. In [21], asynchronous events are proposed for communication between the base-level and the meta-level. In [19], asynchronous, typed events provide implicit concurrency in program designs when events are signaled and consumed without the need for explicit locking of shared states. Our approach using buffered advice enables the base program to progress by reducing advice execution to simple buffering operations in thread-local storage. Thanks to pluggable buffer processing strategies, buffered advice invocations may be executed asynchronously in many different ways.

So far, optimizing the execution of woven code has focused on compilation techniques [2, 28, 29], on static analysis [9], or on the integration of aspect support within the JVM [8, 7]. Aspect optimization with buffered advice is a different direction for speeding up advice execution on multicores. On the one hand, buffered advice is not a transparent optimization automatically performed by the weaver. The aspect programmer has to identify and annotate advice that can be buffered. On the other hand, low-level implementation details are automatically handled by the weaver, allowing the programmer to rapidly experiment with different buffering schemes and distinct buffer processing strategies.

AspectJ does not provide explicit support for concurrent programs; the programmer must manually deal with synchronization and explicitly handle thread communication in advice bodies. In [11], Concurrent Event-based AOP (CEAOP) is proposed to overcome this limitation. CEAOP introduces the notion of concurrent aspects and handles their coordination. Concurrent aspects allow to synchronize advice of different aspects applied to the same join point, or conversely, to execute advice bodies in parallel. Contrarily to our approach, this is achieved by parallelizing the aspect itself, whereas we allow different strategies to execute buffered advice bodies in parallel. While CEAOP introduces its own aspect language and weaver, our approach builds on standard AspectJ.

Exploiting parallelism on multi-processor systems is an important research topic in the area of High Performance Computing (HPC). In [15], AOP is used to solve the problem of tangling code in high-performance parallel scientific software. The authors show that aspects can be used to parallelize numerical applications, provided that the underlying design is sound and the loop structure is compatible with AspectJ's join points (otherwise requiring major refactoring of the code to be parallelized). While our approach aims at exploiting hardware parallelism in order to optimize advice execution, we do not consider parallelization of the woven application.

In [23], the AWED aspect language for distributed computing is presented. It supports asynchronous and synchronous advice execution. Similar to DJCutter [24], AWED introduces remote pointcuts that enable monitoring of execution events on different hosts. Other approaches to distributed AOP include dynamic deployment of aspects [33], DADO [40], JAC [26], DyRes [35], and ReflexD [34]. Even though our approach does not target distributed AOP, it naturally enables asynchronous advice execution, and remote processing can be implemented as a special `Processor` implementation (assuming that the buffered data is serializable).

Shadow profiling [22] is a technique for sampling long traces of instrumented code in parallel with normal program execution. Instead of directly instrumenting an application, a shadow process is periodically created and instrumented for profiling, while the original process continues execution. The Pin [20] binary instrumentation tool is used to instrument the shadow process. While this approach avoids the execution of profiling code by the primary application process, the execution of the shadow process is not optimized. Similar to shadow profiling, SuperPin [39] is also based on Pin and exploits parallel hardware to speed up instrumented code. It runs the uninstrumented application and periodically forks off instrumented, non-overlapping slices of large code regions that execute concurrently, thereby hiding much of the cost of executing the analysis code. Both shadow profiling and SuperPin target parallelism at the level of processes and use a low-level instrumentation scheme. In contrast, our approach enables a high-level description of transformations using aspects, exploits fine-grained parallelism at the thread level, and supports different processing strategies, while hiding low-level buffering issues from the developer.

In [14], a dynamic analysis framework is presented to exploit multicore hardware. The framework assumes an 1:1 association of program and analysis threads and makes use of carefully set CPU affinities of threads. While [14] relies on low-level instrumentations and modifications to the Jikes RVM[8], buffered advice are part of a high-level framework based on thread-local buffering, support

[8] `http://jikesrvm.org/`

any n:m association of program and analysis threads, and are compatible with any standard JVM.

In previous work [4], we explored aspects for calling context profiling and investigated the parallelization of program execution and profile creation, leveraging idle CPU cores on a multicore machine. We showed that parallelized calling context profiling is up to 120% faster than a primitive, sequential approach (geometric mean for the DaCapo [6] benchmark suite). This encouraging result has motivated our work on buffered advice. Whereas [4] presents a hard-coded, manually crafted parallelization scheme, buffered advice offers automated code generation for buffering advice invocations and supports pluggable buffer processing strategies, making it easy to experiment with different parallelization schemes.

9. DISCUSSION

In this section we discuss the strengths and limitations of our approach and outline some ongoing research.

Buffered advice eases the optimization of aspects when synchronous advice execution by application threads causes high overhead. If synchronous advice execution is not a necessity, buffered advice enables asynchronous advice execution in a very flexible manner. Asynchronously executing individual advice invocations often causes too much communication overhead for paying off, as the information regarding the invoked advice and the relevant context information need to be passed to another thread using some shared data structure. Buffered advice allows aggregating advice invocations to increase the granularity of the workload to be executed asynchronously. As the communication overhead is incurred only once per buffer, a sufficiently large buffer size mitigates that overhead. With our framework, the buffer size can be tuned at runtime.

Although buffered advice introduces some extra sources of overhead in comparison with standard advice (buffer allocation and garbage collection, storing/loading data from/in the buffer, invocation of a custom processor, etc.), in our case study we have shown that the benefits of improved locality due to buffering can outweigh these overheads, even without any parallelization. Storing context information in thread-local storage upon invocation of buffered advice has less impact on the locality of program code than more complex advice bodies that may access some shared data structures of the aspect. Conversely, processing all advice invocations conveyed in a buffer together can improve locality of aspect code.

Correctly using buffered advice requires careful reasoning with respect to the state to be preserved in the buffer. It is often not sufficient to simply add `@Buffered` annotations to advice in an aspect that has not been designed with buffering in mind. As buffers may be processed by any thread according to the configured processing strategy, the bodies of buffered advice should not use `Thread.currentThread()` for obtaining a reference to the advice-invoking program thread. Invocation-local variables, which complement buffered advice, allow storing arbitrary state (maintained by standard advice) in the buffer. In the examples in Figure 2(b) and in Figure 8, invocation-local variables are used to store a reference to the advice-invoking program thread in the buffer. Furthermore, in many cases, it is important that the data stored in the buffer remains unchanged until the buffer is processed. For example, in our case study, `RacerBuf` ensures that effectively immutable versions of the current thread's bag of held locks are stored in the buffer (see Figure 8).

Our framework offers a flexible API to customize buffer processing. On the one hand, generic buffer processors (e.g., processors using thread pools) can be reused in different aspects that make use of buffered advice. On the other hand, a buffer processor can be specialized for a particular aspect. The `submit(Buffer)` method to be implemented by a processor allows executing any code before and after processing a buffer. For example, locks may be acquired/released before/after processing a buffer. At the same time, code for lock acquisition/release in the buffered advice bodies may be removed. That is, instead of executing each buffered advice body in a separate critical section, the whole buffer can be processed within a single critical section, which will largely reduce the number of lock acquisition/release operations. To this end, it is necessary to design the processor and the aspect together, implementing an optimized synchronization policy.

Our approach introduces only two annotations, `@Buffered` and `@InvocationLocal`, which are used in conjunction with standard AspectJ language constructs. Aspects with buffered advice may be programmed using traditional AspectJ pointcut and advice syntax, or using the annotation-based syntax introduced in AspectJ 5. Since buffered advice is expressed with Java annotations, we avoid modifying the aspect compiler and concentrate our efforts on weaving of buffered advice, which involves code transformations and code generation. Recently, many AOP frameworks, such as JBossAOP [17], AspectWerkz [1], Spring AOP [32], or Spoon [25], also support annotation-based aspect specifications.

The code transformations and code generation in support of buffered advice are implemented as a separate weaver module that post-processes code that has previously been woven with the standard AspectJ weaver ajc. This approach allows us to add support for buffered advice to different versions of ajc. Furthermore, the weaver module works in conjunction with MAJOR [37] and with the dynamic AOP framework HotWave [36], since both of them are based on ajc. Both MAJOR and HotWave ensure comprehensive aspect weaving, including the Java class library, which is important for many aspect-based software-engineering tools that perform some kind of dynamic program analysis. Hence, MAJOR or HotWave in conjunction with buffered advice are convenient, high-level frameworks for rapidly developing practical and efficient dynamic analysis tools for the JVM.

Regarding limitations, our implementation suffers from the same problem as any other AOP framework relying on Java bytecode instrumentation. The JVM imposes strict limits on certain parts of a class file (e.g., the method size is limited); these limits may be exceeded by the code inserted upon aspect weaving. Our approach aggravates this issue by inlining (standard) advice that access invocation-local variables. Nonetheless, in our use cases and tests, we have not yet encountered significant problems due to code growth.

The use of advice inlining results in a restriction of the supported AspectJ constructs. Non-singleton aspect instances using `per*` clauses are not supported in conjunction with invocation-local variables and buffered advice. Moreover, buffered advice as well as standard advice accessing invocation-local variables can only use public members of the aspect.

Buffered advice may increase memory consumption. On the one hand, each thread maintains a buffer, which may comprise several possibly large arrays. On the other hand, the lifetime of objects stored in the buffer may be extended, since these objects are kept alive until a buffer has been processed. While extended life time is not an issue for static join point instances (which are allocated only in the static initializers of woven classes) and for invocation-local variables of primitive types, it has to be taken into consideration when objects of other types are stored in the buffer. For a large buffer size and many concurrent program threads, the increase in memory consumption due to buffered advice can become significant. In addition, for asynchronous buffer processing using a

thread pool executor, as a means of limiting memory consumption, it is advisable to limit the maximum number of threads in the pool as well as the maximum size of the queue used for communicating buffers [12]. This approach will prevent program threads from overloading the system with full buffers.

Concerning ongoing research, we are extending our framework to support *buffered methods*, that is, `@Buffered` annotations for methods with return type `void`. Instead of directly executing a method, the receiver and the arguments are stored in the buffer. Note that buffered advice can be recast as standard advice that invokes buffered methods. The context information to be preserved in the buffer is explicitly passed from the advice to the invoked buffered method. In this way, it is not necessary to use invocation-local variables in order to store arbitrary data in the buffer. In contrast to buffered advice, which is specific to AOP, buffered methods may be used in a pure object-oriented setting.

We are also working on the domain-specific AOP language and framework @J [5], which aims at simplifying the development of instrumentation-based dynamic analysis tools for the JVM. @J is an annotation-based AOP language with dedicated support for advice composition, data passing between advice woven in the same method, weave-time evaluation of advice, as well as join points at the level of individual bytecodes and basic blocks of code. @J allows recasting a large variety of instrumentation-based tools as compact aspects that can be easily extended. Buffered advice is an important feature to be integrated in @J, since it eases the optimization of aspect-based tools so as to leverage multicores.

10. CONCLUSION

In this paper, we have introduced buffered advice, a new AOP feature that decouples advice invocation by program threads from the advice execution policy. Buffered advice enables the aggregation of advice invocations, preserving the required context information in a thread-local buffer. When a buffer is full, all advice invocations collected in the buffer are processed together according to a pluggable, custom processing strategy. Buffer processing may occur in parallel with program execution, in order to exploit idle CPU cores. Buffering the relevant context information upon advice invocations in thread-local storage is an effective way of increasing the granularity of the workload to be asynchronously executed by other threads. Instead of separately passing the context information for each advice invocation, which may incur high thread communication overhead, only complete buffers are passed between threads. Our approach helps achieve considerable speedup on multicores when parallelizing the execution of aspect and program code. The speedup not only stems from parallelization, but also from improved locality in the woven code and in the aspect.

Thanks to a high-level programming model based on AspectJ, augmented with two new annotations, the details of buffering are hidden from the aspect programmer. The programmer specifies the advice that shall use buffering, as well as the needed context information, which may include any aspect state maintained by standard advice. Our framework then automatically generates the buffering code for the aspect. A flexible API eases the experimentation with different buffer processing strategies.

As case study, we optimized an existing aspect-based data race detector with the aid of buffered advice. On a quad-core machine, the aspect using buffered advice achieves a speedup of factor 3.29 on average.

Acknowledgements: The work presented in this paper has been supported by the Swiss National Science Foundation.

11. REFERENCES

[1] Aspectwerkz. AspectWerkz - Plain Java AOP. Web pages at `http://aspectwerkz.codehaus.org/`.

[2] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Optimising AspectJ. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 117–128, New York, NY, USA, 2005. ACM.

[3] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 87–98, New York, NY, USA, 2005. ACM Press.

[4] W. Binder, D. Ansaloni, A. Villazón, and P. Moret. Parallelizing Calling Context Profiling in Virtual Machines on Multicores. In *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 111–120, New York, NY, USA, 2009. ACM.

[5] W. Binder, A. Villazón, D. Ansaloni, and P. Moret. @J - Towards Rapid Development of Dynamic Analysis Tools for the Java Virtual Machine. In *VMIL '09: Proceedings of the 3th Workshop on Virtual Machines and Intermediate Languages*, New York, NY, USA, 2009. ACM.

[6] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, Oct. 2006. ACM Press.

[7] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *AOSD*, pages 83–92, 2004.

[8] C. Bockisch, S. Kanthak, M. Haupt, M. Arnold, and M. Mezini. Efficient Control Flow Quantification. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 125–138, New York, NY, USA, 2006. ACM.

[9] E. Bodden, F. Chen, and G. Roşu. Dependent advice: A general approach to optimizing history-based aspects. In *International Conference on Aspect-Oriented Software Development (AOSD)*, pages 3–14. ACM Press, Mar. 2009.

[10] E. Bodden and K. Havelund. Racer: Effective Race Detection Using AspectJ. In *International Symposium on Software Testing and Analysis (ISSTA), Seattle, WA, July 20-24 2008*, pages 155–165, New York, NY, USA, 07 2008. ACM.

[11] R. Douence, D. Le Botlan, J. Noyé, and M. Südholt. Concurrent Aspects. In *GPCE '06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, pages 79–88, New York, NY, USA, 2006. ACM.

[12] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[13] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java*

Language Specification, Third Edition. The Java Series. Addison-Wesley, 2005.

[14] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 155–174, New York, NY, USA, 2009. ACM.

[15] B. Harbulot and J. R. Gurd. Using AspectJ to separate concerns in parallel scientific Java code. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 122–131, New York, NY, USA, 2004. ACM.

[16] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 26–35, New York, NY, USA, 2004. ACM.

[17] JBoss. Open source middleware software. Web pages at `http://labs.jboss.com/jbossaop/`.

[18] C. Kung and C. Ju-Bing. Aspect-Based Instrumentation for Locating Memory Leaks in Java Programs. In *Computer Software and Applications Conference, 2007. COMPSAC 2007*, pages 23–28, Beijing, China, 2007. IEEE Computer Society.

[19] Y. Long, S. Mooney, and H. Rajan. Panini: A language with asynchronous, typed events. Technical Report 09-28, Iowa State University, Department of Computer Science, October 2009.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Ptil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI 2005: Proceedings of the ACM SIGPLAN 2005 conference on Programming Language Design and Implementation*, pages 191–200. ACM Press, 2005.

[21] J. Malenfant and S. Denier. ARM : un modèle réflexif asynchrone pour les objets répartis et réactifs. *L'OBJET*, 9(1-2):91–103, 2003.

[22] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow Profiling: Hiding Instrumentation Costs with Parallelism. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 198–208, Washington, DC, USA, 2007. IEEE Computer Society.

[23] L. D. B. Navarro, M. Südholt, W. Vanderperren, B. De Fraine, and D. Suvée. Explicitly Distributed AOP using AWED. In *AOSD '06: Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 51–62, New York, NY, USA, 2006. ACM.

[24] M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut: a language construct for distributed AOP. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 7–15, New York, NY, USA, 2004. ACM.

[25] R. Pawlak. Spoon: Compile-time annotation processing for middleware. *IEEE Distributed Systems Online*, 7(11):1, 2006.

[26] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli. JAC: An Aspect-based Distributed Dynamic Framework. *Software: Practice and Experience*, 34(12):1119–1148, 2004.

[27] D. J. Pearce, M. Webster, R. Berry, and P. H. J. Kelly. Profiling with AspectJ. *Software: Practice and Experience*, 37(7):747–777, June 2007.

[28] A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for Java. In *AOSD '03: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 100–109, New York, NY, USA, 2003. ACM Press.

[29] Y. Sato, S. Chiba, and M. Tatsubori. A selective, just-in-time aspect weaver. In *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 189–208, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[30] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[31] B. C. Smith. *Procedural Reflection in Programming Languages*. PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, 1982.

[32] Spring Framework. Open source application framework. Web pages at `http://www.springsource.org/`.

[33] E. Tanter, J. Fabry, R. Douence, J. Noyé, and M. Südholt. Expressive scoping of distributed aspects. In *AOSD '09: Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development*, pages 27–38, New York, NY, USA, 2009. ACM.

[34] E. Tanter and R. Toledo. A Versatile Kernel for Distributed AOP. In *DAIS 2006 : Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 316–331. Springer-Verlag, June 2006.

[35] E. Truyen, N. Janssens, F. Sanen, and W. Joosen. Support for distributed adaptations in aspect-oriented middleware. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 120–131, New York, NY, USA, 2008. ACM.

[36] A. Villazón, W. Binder, D. Ansaloni, and P. Moret. Advanced Runtime Adaptation for Java. In *GPCE '09: Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*, pages 85–94. ACM, Oct. 2009.

[37] A. Villazón, W. Binder, and P. Moret. Aspect Weaving in Standard Java Class Libraries. In *PPPJ '08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*, pages 159–167, New York, NY, USA, Sept. 2008. ACM.

[38] A. Villazón, W. Binder, and P. Moret. Flexible Calling Context Reification for Aspect-Oriented Programming. In *AOSD '09: Proceedings of the 8th International Conference on Aspect-oriented Software Development*, pages 63–74, Charlottesville, Virginia, USA, Mar. 2009. ACM.

[39] S. Wallace and K. Hazelwood. SuperPin: Parallelizing dynamic instrumentation for real-time performance. In *5th Annual International Symposium on Code Generation and Optimization*, pages 209–217, San Jose, CA, March 2007.

[40] E. Wohlstadter, S. Jackson, and P. Devanbu. DADO: Enhancing Middleware to Support Crosscutting Features in Distributed, Heterogeneous Systems. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 174–186, Washington, DC, USA, 2003. IEEE Computer Society.