Università della Svizzera italiana

Faculty of Informatics

*USI Technical Report Series in Informatics*

# Exploiting Under-Utilized Cores with Deferred Methods

Danilo Ansaloni[1], Walter Binder[1], Abbas Heydarnoori[1], Lydia Y. Chen[2]

[1] Faculty of Informatics, Università della Svizzera italiana, Switzerland
[2] IBM Zurich Research Laboratory, Zurich, Switzerland

Abstract

Effective parallelization of fine-grained tasks, such as in dynamic program analysis, is challenging because thread communication overheads may outweigh the benefits of parallelism. In this paper, we address this issue with deferred methods, a novel Java framework that aggregates invocations of analysis methods in thread-local buffers and processes them altogether when a buffer is full. The framework supports pluggable buffer processing strategies. We present an adaptive strategy that combines synchronous buffer processing (by the thread that has filled the buffer) and asynchronous buffer processing (using a thread pool). This strategy adapts at runtime to the CPU utilization of the workload and exploits under-utilized cores when possible. As case studies, we consider two dynamic program analyses, a profiler and a data race detector, which we apply to standard benchmarks. A thorough performance evaluation confirms that our framework together with the adaptive buffer processing strategy yields an average speedup of factor 2.2–2.7 for our case studies on two different quad-core machines.

Report Info

*Published*
August 2010

*Number*
USI-INF-TR-2010-7

*Institution*
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland

*Online Access*
www.inf.usi.ch/techreports

## 1 Introduction

Dynamic analysis which aims to understand the behavior of software systems at runtime is an important area in software engineering with various applications in debugging [1, 2], program comprehension [3], performance optimization [4, 5], and security [6]. However, collecting dynamic profiles not only imposes some overheads on the base program, but also the overhead attributable to user-specific analyses may significantly reduce the performance [7]. The wide-scale deployment of shared-memory multicore systems offers a huge potential to improve the performance of dynamic analyses by offloading the analysis tasks to under-utilized CPU cores and executing them in parallel. To this end, several approaches have been introduced in literature [7, 8, 9, 10, 11, 12].

Although all these approaches offload dynamic analyses to spare CPU cores to improve the performance, none of them provide a high-level API with which programmers can develop custom dynamic analysis tools on multicores. Unfortunately, developing parallel software for shared-memory multicores using today's programming languages can be challenging as well due to the lack of high-level parallelization constructs [13]. Additionally, existing programming constructs do not support all parallelization needs and may not yield the expected benefits. For instance, one common solution to program parallelization is to use thread pools. However, following this approach to parallelize fine-grained tasks, which is often the case in the domain of dynamic analysis, will hardly produce any benefits in terms of performance due to increased costs for object allocation, garbage collection, and communication among the parallel threads. The increased communication costs is especially a fundamental issue in shared-memory parallel computing where access to shared data structures is required [14, 10, 12].

In many dynamic analyses, the analysis methods are often fine-grained tasks that can be executed independently from the base program methods, i.e., they produce results or side effects on which the execution

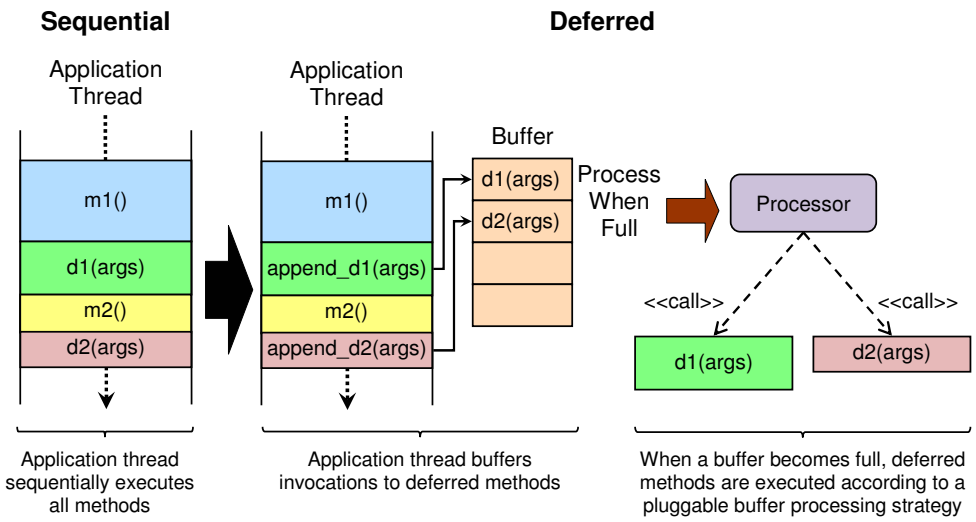



Figure 1: Sequential execution vs. deferred execution

of the base program do not depend. Hence, when parallelizing analysis tasks, one can aggregate the invocations to these methods in thread-local buffers and execute them altogether to make the parallelized tasks coarse-grained in order to reduce the communication overheads and to avoid allocating an object for each fine-grained task. To this end, this paper presents the framework of *Deferred Methods*. As illustrated in Figure 1, upon invocation of a deferred method (e.g., methods `d1(args)` and `d2(args)`), instead of immediately executing the method body, the method ID and all its arguments are stored in a thread-local buffer. When a buffer becomes full, a customizable *processor* processes the workload conveyed in the buffer altogether, that is, the respective method bodies are executed using the buffered arguments. The processor is a pluggable component that implements a particular buffer processing strategy. For example, the buffer can be processed synchronously by the same thread that has filled the buffer, or it can be asynchronously processed in parallel by another thread. In this paper, we present a novel adaptive strategy that combines these two processing strategies at runtime to adapt the CPU utilization to the workload conveyed in buffers and to exploit under-utilized cores when possible.

Our framework of deferred methods provides a simple, flexible API in standard Java for deferred execution of methods. Thanks to automated code generation, details regarding concrete buffer implementations are hidden from the programmer. Nevertheless, this API provides programmers the required primitives for managing and customized processing of buffers. More importantly, deferred methods offer several key advantages to improve the performance of dynamic analyses. First, deferred methods allow parallel execution of workloads conveyed in buffers, taking advantage of under-utilized CPU cores. Second, it is possible to process buffers using different pluggable, custom processing strategies in a flexible way. Third, with deferred methods communication costs among parallel threads are paid only once per buffer instead of once per method invocation. Finally, as the bodies of deferred methods that are part of the same buffer are executed one after the other by the same processing thread, locality will improve if those methods execute similar code (e.g., different invocations of the same method) or if they access the same data structures.

The contributions of the paper include (i) introducing the notion of deferred methods for parallelizing dynamic analyses and providing an API in pure Java that automatically generates the required code for buffering and handling the deferred methods, (ii) providing an adaptive buffer processing strategy that adapts at runtime to the CPU utilization of the base program and exploits under-utilized cores for improved performance, and (iii) a detailed performance evaluation with standard benchmarks that confirms the benefits of deferred methods.

In the rest of this paper, Section 2 presents a motivating example which is used throughout this paper. Section 3 provides the API of the framework of deferred methods. Section 4 discusses various buffer processing strategies. Section 5 describes a detailed performance evaluation. Finally, Section 6 compares the framework with related work, and Section 7 concludes.

```
1  public aspect BlueprintAspect {
2      private final ThreadLocal<Stack<JPSP>> stackTL =
3          new ThreadLocal<Stack<JPSP>>() {
4              protected Stack<JPSP> initialValue() {
5                  return new Stack<JPSP>();
6              }
7          };

8      pointcut allExecs() : execution(* *(..)) ||
9          (execution(*.new(..))) && !within(blueprint.*);

10     before() : allExecs() {
11         Stack<JPSP> localStack = stackTL.get();
12         CallerData.getCallerData(localStack.peek()).
13                         regCallee(thisJoinPointStaticPart);
14         localStack.push(thisJoinPointStaticPart);
15     }

16     after() : allExecs() { stackTL.get().pop(); }

17     after() returning(Object obj) : call(*.new(..)) &&
18                                     !within(blueprint.*) {
19         CallerData.getCallerData(stackTL.get().peek()).
20                                     regAlloc(obj);
21     }
22     ...
23 }

24 ... // CallerData is not shown to save space.
```

Figure 2: Simplified aspect for blueprint profiling

## 2 Motivating Example

Profiling blueprints [15] provides graph-like views of a program's execution to help programmers identify bottlenecks and give hints on how to remove them. In these views, each node is represented as a rectangle whose width and height independently illustrate two desired dynamic metrics (e.g., number of allocated objects and total size of the allocated objects) for a particular program element (e.g., method). Additionally, edges indicate relationships among the program elements (e.g., caller/callee relationships) and the properties of edges can show some characteristics of those relationships (e.g., the width of edges can show the number of times each callee is called). Therefore, to create such visualizations, we need to collect mappings `elem`→`<Dynamic metric1, Dynamic metric2, Related program elements and their properties>` for each program element `elem` at runtime. A relatively simple way to collect such profiles is to instrument programs using aspect-oriented programming languages like AspectJ[1]. For example, Figure 2 illustrates a simplified AspectJ program that can be used to collect mappings `caller`→`<Number of allocated objects, Total size of allocated objects, Callees and the number of times each callee is called>` for each method[2]. In this figure, the class `CallerData`, which is not shown for space reasons, stores this mapping data for each method `mid`. The instance of this class that corresponds to method `mid` can be obtained by the static method `getCallerData(mid)`.

The BlueprintAspect keeps a *shadow stack* of calls for each thread in the thread-local variable `stackTL`. This shadow stack is maintained in order to reconstruct the caller/callee relationships. The `before()` advice is woven in method entries. By calling the `regCallee(...)` method, it registers the method invocation in the collection of callees for the method which is currently on top of the shadow stack. Furthermore, the callee's identifier is pushed onto the shadow stack. Instances of type `JoinPointStaticPart` (abbreviated in this paper as JPSP) serve as method identifiers. They are accessed through the AspectJ's `thisJoinPointStaticPart` pseudo-variable.

The first `after()` advice, which intercepts (normal and abnormal) method completion, pops the completing method identifier off the shadow stack. The second `after()` advice which is woven after object allocations, invokes the `regAlloc(...)` method to increment the number and the total size of the allocated objects for the current caller with the size of the newly allocated object.

Careful consideration of this program reveals that it is not necessary to synchronously call the

[1] http://www.eclipse.org/aspectj/

[2] In this section, the term method refers to both method and constructor unless explicitly mentioned.

`regCallee(...)` and `regAlloc(...)` methods in the `before()` and the second `after()` advice by the same thread that runs the base program. Since the execution of the base program methods does not depend on the computations performed by these methods, they can be asynchronously executed in parallel by another thread using under-utilized CPU cores. However, as will be discussed in our evaluations (see Section 5), since these methods perform fine-grained tasks, simply parallelizing them does not pay off due to increased costs for object allocation, garbage collection, and communication among the parallel threads. More specifically, we measured that sequential blueprint profiling introduces an average overhead of factor 10.58–13.07, while a naive parallel implementation introduces an overhead of up to factor 190 on standard benchmarks. Thus, to overcome the parallelization overheads and to improve performance, we make the parallelized tasks coarse-grained by aggregating the invocations to these methods in a thread-local buffer and processing them altogether when this buffer is full. The following section presents how this can be done with our framework of deferred methods.

## 3 Framework API

Our framework of deferred methods provides a simple, flexible API in standard Java for specifying deferred methods. It also provides the required primitives for managing and customized processing of buffers. We present this API with the help of the motivating example presented in Section 2. Figure 3 illustrates a layered class diagram showing how the framework of deferred methods is used. This class diagram includes three kinds of classes and interfaces: (i) the ones that are provided by the API, (ii) the ones that are implemented by the application developer, and (iii) the ones that are automatically generated by the framework at runtime. The actual code implementing this class diagram is illustrated in Figure 4, showing how this framework is used to refactor the code in Figure 2.

The `Deferred` interface in Figure 3 is a marker interface that serves only to mark the classes which calls to their methods should be deferred. Therefore, to use the framework of deferred methods, first the user needs to extend this interface with the methods that he wants to execute in a deferred way and then implement the extended interface. Our framework requires this class to be neither abstract nor final, and all deferred methods must return `void`. For instance, as can be seen in the refactored code in Figure 4, there is a new interface named `DefMethods` that extends the `Deferred` interface and defines two methods `profCall(...)` and `profAlloc(...)`. This interface is implemented in class `DefMethodsImpl` such that the methods `profCall(...)` and `profAlloc(...)` respectively invoke the methods `regCallee(...)` and `regAlloc(...)` which are to be deferred.

Based on the class that implements the deferred methods, i.e., `DefMethodsImpl`, our framework automatically generates a buffer class, i.e., `GeneratedBuffer` in Figure 3, that implements the `Runnable` interface. At runtime, our framework automatically replaces each invocation to a deferred method (e.g., Line 29 in Figure 4) with an invocation of a corresponding method (e.g., `append_profCall(...)`) in the generated buffer implementation that appends the method ID and all its arguments to the current thread's buffer. The buffer content is stored in several arrays. The integer array `methodID` holds unique IDs representing the invoked deferred methods (i.e., deferred methods are numbered). The other arrays keep the arguments of deferred methods. Consequently, the element types of these arrays correspond to the types of the arguments of deferred methods. If several deferred methods share the same argument type, the same array is used for storing the arguments to minimize the number of allocated arrays. For instance, in Figure 4, the type of the first argument of both methods `profCall(...)` and `profAlloc(...)` is `JPSP`. Hence, the field `argA` in the buffer holds the first arguments of invocations to both of these methods.

When a buffer becomes full, it is automatically submitted to the specified processor through calling the `process(buffer)` method and a new instance of the buffer is created then. Our framework also modifies the `run()` method in `java.lang.Thread` and in all its subclasses so as to submit the thread's buffer, even if it is not full, before its termination. The processor then calls the buffer's `run()` method which in turn executes all method invocations conveyed in the buffer. In contrast to the automatically generated buffer, the processor implementation has to be provided by the programmer by implementing the `Processor` interface. For convenience, our framework includes processor implementations that are suited for various applications. It also automatically adapts the processing strategy at runtime based on the CPU utilization of the workload to exploit under-utilized cores when possible. Section 4 provides a detailed discussion of these processing strategies.

The user may optionally implement the `ProcessingHooks` interface in his implementation class to receive callbacks from the framework before and after processing the buffered method invocations. The main

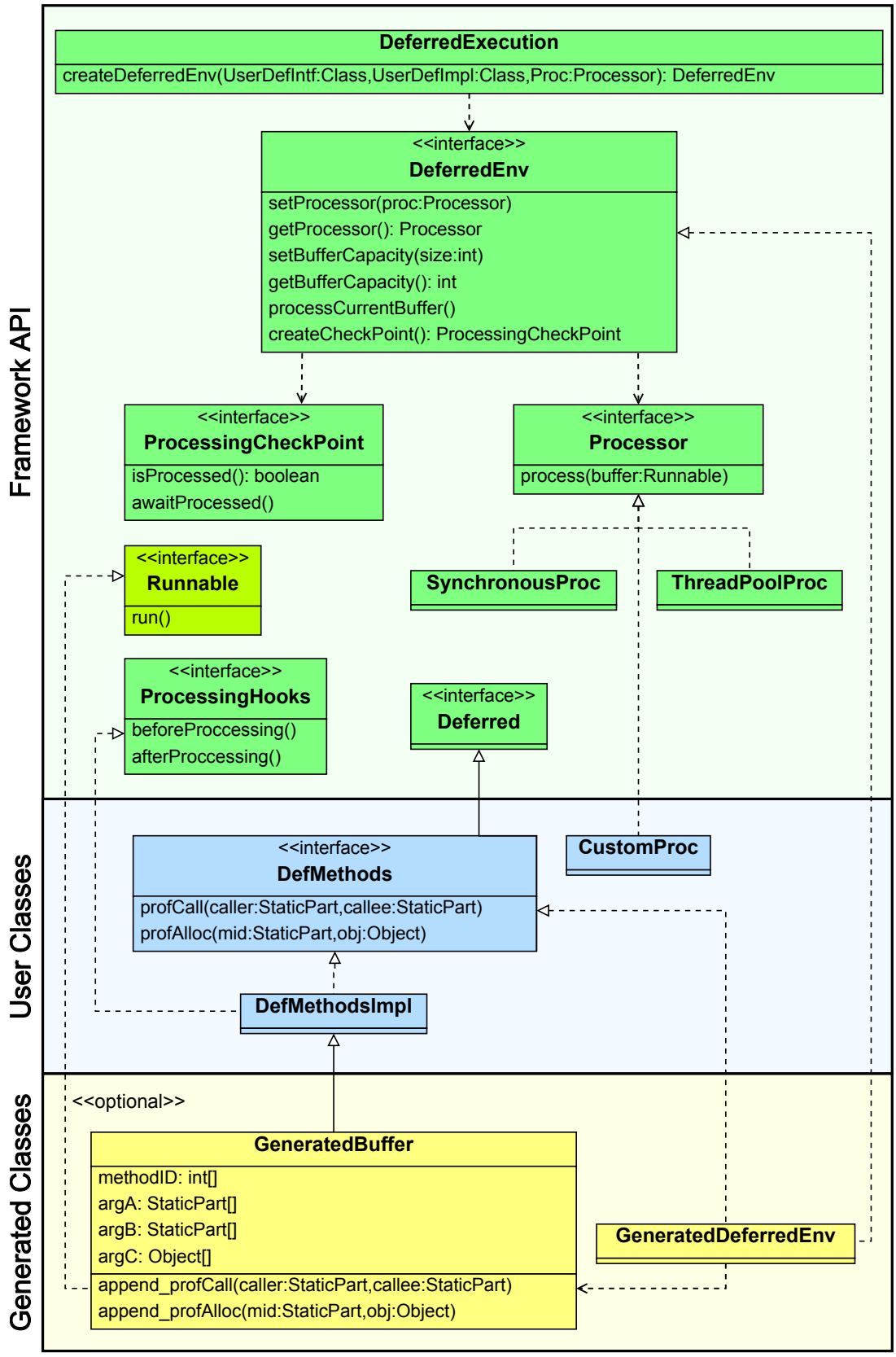



Figure 3: Framework API

```
1  public interface DefMethods extends Deferred {
2      void profCall(JPSP caller, JPSP callee);
3      void profAlloc(JPSP mid, Object obj);
4  }

5  public class DefMethodsImpl implements DefMethods {
6      public void profCall(JPSP caller, JPSP callee) {
7          CallerData.getCallerData(caller).regCallee(callee);
8      }

9      public void profAlloc(JPSP mid, Object obj) {
10         CallerData.getCallerData(mid).regAlloc(obj);
11     }
12 }

13 public aspect DefBlueprintAspect {
14     private static final int NUM_THREADS = 4;

15     private static final DefMethods def =
16         (DefMethods)DeferredExecution.createDeferredEnv(
17                         DefMethods.class, DefMethodsImpl.class,
18                         new ThreadPoolProc(NUM_THREADS));

19     private final ThreadLocal<Stack<JPSP>> stackTL =
20         new ThreadLocal<Stack<JPSP>>() {
21             protected Stack<JPSP> initialValue() {
22                 return new Stack<JPSP>();
23             }
24         };

25     pointcut allExecs() : execution(* *(..)) ||
26         (execution(*.new(..))) && !within(blueprint.*);

27     before() : allExecs() {
28         Stack<JPSP> localStack = stackTL.get();
29         def.profCall(localStack.peek(), thisJoinPointStaticPart);
30         localStack.push(thisJoinPointStaticPart);
31     }

32     after() : allExecs() { stackTL.get().pop(); }

33     after() returning(Object obj) : call(*.new(..)) &&
34                                     !within(blueprint.*) {
35         def.profAlloc(stackTL.get().peek(), obj);
36     }
37     ...
38 }

39 ... // CallerData is not shown to save space.
```

Figure 4: Simplified aspect for blueprint profiling using deferred methods

use case of this interface is to support the implementation of custom *coalescing* strategies. For instance, instead of accessing a shared data structure each time that a method is invoked, one can define some instance fields in class `DefMethodsImpl` to record the invocation. These fields can be initialized in the `beforeProcessing()` method, updated in the deferred methods, and finally the results can be integrated into the shared data structure in the `afterProcessing()` method. In this way, we prevent the shared data structure becoming a bottleneck in parallelized access.

The interface `DeferredEnv` provides some methods that can be used by programmers to influence the management of buffers and processing of deferred methods at runtime. The methods `setProcessor(...)` and `getProcessor()` are used to change or access the processor associated with a deferred environment and so, they can be used to influence the buffer processing strategy at runtime. The buffer capacity, which can be queried with method `getBufferCapacity()`, is initially specified through a system property, and may be changed at runtime by calling the method `setBufferCapacity(int)`. Changing the buffer capacity influences subsequent buffer allocations; it does not change the capacity of the current buffer. Although our framework automatically processes a buffer when it becomes full, method `processCurrentBuffer()` can be used to force immediate processing of the current thread's buffer, independently of its filling state; afterwards, a new buffer is created.

Synchronization between the thread invoking a deferred method, and the thread executing it can be achieved by passing a synchronization object (e.g., a `Semaphore` instance) as an argument to the deferred method. It is also possible to pass future value objects as arguments to deferred methods, such that the caller can later wait for a result. This mitigates the restriction that only methods with return type `void` can be deferred.

```
public class SynchronousProc implements Processor {
    public void process(Runnable buffer) {
        buffer.run();
    }
}
```

(a) SP processor implementation

```
public class ThreadPoolProc implements Processor {
    private final Executor exec;

    public ThreadPoolProc(int numberThreads) {
            exec = Executors.newFixedThreadPool(numberThreads);
    }

    public void process(Runnable buffer) {
            exec.execute(buffer);
    }
}
```

(b) TP processor implementation

Figure 5: Two sample processor implementations SP and TP

Additionally, the method `createCheckPoint()` can be used to create processing checkpoints (instances of type `ProcessingCheckPoint`). A processing checkpoint marks the current position in the buffer of the current thread $T$, and allows $T$ to wait for the processing of all deferred methods invoked by it before creating the checkpoint. For instance, this can be used to make sure that a certain previously invoked deferred method has been executed before a thread of the base program can proceed. While method `isProcessed()` of type `ProcessingCheckPoint` is non-blocking, method `awaitProcessed()` is blocking. Since buffers submitted by the same thread may be processed out-of-order (depending on the `Processor` implementation), it is not sufficient to wait until the deferred methods conveyed in just the current buffer have been processed; the deferred methods in previously submitted buffers (for the same deferred environment and the same thread) must have been processed as well.

Based on the deferred interface provided by the user (i.e., `DefMethods`), our framework automatically generates and loads the class `GeneratedDeferredEnv`. This class acts similar to a *dynamic proxy* and redirects all invocations to deferred methods to their corresponding methods in the buffer implementation (i.e., `GeneratedBuffer`) which in turn appends that invocation to the buffer for later processing.

The method `createDeferredEnv(...)` of class `DeferredExecution` is called by the programmer to create an instance of the deferred environment. After that, all invocations to deferred methods are buffered and processed later using this environment. It is also possible for the user to call this method several times to create several instances of the deferred environment with different buffers and processors for various needs.

## 4 Buffer Processing Strategies

We propose three strategies for processing the full buffers: (i) *synchronous processing* (*SP*) by the thread that has filled the buffer, (ii) *asynchronous (thread-pool) processing* (*TP*) by dedicated threads in a pool, and (iii) *adaptive processing* (*AP*) that reconciles synchronous and asynchronous processing. In the following, the term *producer* refers to a thread of the base program that fills buffers, whereas the term *consumer* refers to a dedicated thread that only processes buffers. The number of producers is the number of threads in the base program, whereas the number of consumers is a controlled variable depending on the processing strategy.

### 4.1 Synchronous Processing (SP)

With SP (Figure 5(a)), each full buffer is processed by its producer; there are no consumers. Hence, SP does not parallelize base program execution and dynamic analysis. SP guarantees that for each producer, the full buffers are processed in order.

At first glance, SP only incurs extra overheads in comparison with traditional, sequential dynamic analysis without deferred methods: buffer allocation, initialization, and garbage collection, as well as storing to and reading from the arrays in the buffer. However, the results of our evaluations in Section 5.4 show that SP can improve locality since buffers are processed altogether. Furthermore, if the deferred methods implement

some coalescing algorithms, the number of time-consuming actions executed by the dynamic analysis may be reduced.

### 4.2 Thread-Pool Processing (TP)

With TP (Figure 5(b)), all buffers are processed by dedicated consumers. In this paper, we assume that the number of consumers $N_{TP}$ (i.e., the thread pool size) is fixed, and that the thread pool uses a bounded blocking FIFO queue of capacity $Q_{TP}$. That is, if the queue is full, producers block until there is space in the queue. The use of a bounded blocking queue helps limiting memory consumption when full buffers are produced at a faster rate than they are consumed. On the other hand, when $Q_{TP}$ is too small, producers might be blocked too frequently and consumers thus starve. Moreover, with TP, full buffers from the same producer may be processed out-of-order due to the presence of multiple consumers. In contrast to SP, TP takes the advantage of under-utilized cores by parallelizing execution of the base program (by producers) and dynamic analysis (by consumers). Similarly to SP, TP may also benefit from improved locality and from coalescing.

Nonetheless, since the communication of full buffers between threads introduces extra overhead, there are workloads where SP outperforms TP, as will be shown in Section 5. In particular, if producers keep all cores busy, there are no under-utilized cores that could be exploited by the consumers.

### 4.3 Adaptive Processing (AP)

AP aims at combining the benefits of TP and SP. On the one hand, when the producers under-utilize some cores, consumers can take advantage of the available computing resources for processing full buffers in parallel with the execution of the base program. On the other hand, when the producers are able to keep all cores busy, it is more efficient to process a full buffer by its producer than to pass it to a consumer, which would compete with the producer for CPU time.

Similar to TP, AP maintains a pool of consumers, which are however executing at minimum thread priority.[3] As (most) producers are typically executing at normal thread priority, they are generally scheduled more frequently than consumers if they are competing for CPU time (albeit the exact scheduling behavior depends on the operating system, as state-of-the-art JVMs rely on native threads). Consequently, consumers can execute when some cores are under-utilized, but rarely preempt producers. Here, we assume the number of consumers $N_{AP}$ equals to the number of cores in the system. That is, if all producers are blocked, the consumers can exploit all cores if there are enough full buffers to be processed. If the dynamic analysis involves frequently blocking actions (which is not the case for the analyses considered in this paper), a higher number of consumers may be appropriate.

While AP uses a bounded FIFO queue of capacity $Q_{AP}$, it never blocks a producer. If the queue is full, the producer itself processes the full buffer, like SP. That is, if there are not enough under-utilized cores for consumers to keep up with the production of full buffers, the queue becomes full, and then buffer production is throttled as producers process their full buffers.

While AP significantly outperforms SP and TP in our case studies (see Section 5), it also has some drawbacks. Out-of-order processing of full buffers from the same producer occurs much more frequently than with TP. Moreover, AP is prone to starvation, if a producer uses synchronization to wait for the completion of a deferred method (e.g., using a `ProcessingCheckPoint`); this is however not the case for the dynamic analyses considered in this paper. If other producers keep all cores busy, consumers may not be able to process full buffers in the queue. However, this problem is mitigated on operating systems where the scheduler dynamically changes thread priorities (i.e., if a thread has not been scheduled for a long time, its priority is increased). Such schedulers ensure that the low-priority consumers eventually receive some CPU time, if they are not blocked.

## 5 Evaluation

In this section we evaluate our framework using the blueprint profiler shown in Figure 4 on two machines with different micro-architectures. First, we investigate the performance impact of different buffer and queue capacities in order to select the best values of these parameters. (Section 5.2). Second, we explore the performance impact and CPU utilization of different buffer processing strategies (Section 5.3). Third, we analyze the

[3] We use method `setPriority(...)` in class `java.lang.Thread` to manipulate the priorities of the consumers before they are started. Note that the Java language specification [16] and the JVM specification [17] do not exactly specify the semantics of Java thread priorities, in order to ease implementation of the JVM on different platforms.

| $Q_{AP}$ | $B_{AP}$ | | | | | | |
|---|---|---|---|---|---|---|---|
| | 32 | 128 | 512 | 1024 | 8192 | 16384 | 32768 |
| 16 | 0.70 | 1.02 | 1.38 | 1.70 | 2.56 | 2.57 | 2.60 |
| 32 | 0.72 | 1.01 | 1.47 | 1.73 | 2.55 | 2.58 | 2.61 |
| 64 | 0.72 | 1.05 | 1.49 | 1.78 | 2.56 | **2.62** | 2.60 |
| 128 | 0.71 | 1.09 | 1.54 | 1.84 | 2.55 | 2.61 | 2.59 |
| 512 | 0.73 | 1.18 | 1.71 | 1.94 | 2.52 | 2.57 | 2.56 |
| 1024 | 0.74 | 1.21 | 1.72 | 1.97 | 2.49 | 2.50 | 2.44 |
| 2048 | 0.75 | 1.26 | 1.75 | 1.96 | 2.42 | 2.39 | 2.36 |

(a) Machine 1

| $Q_{AP}$ | $B_{AP}$ | | | | | | |
|---|---|---|---|---|---|---|---|
| | 32 | 128 | 512 | 1024 | 8192 | 16384 | 32768 |
| 16 | 0.35 | 0.67 | 1.04 | 1.34 | 2.48 | 2.63 | 2.67 |
| 32 | 0.35 | 0.63 | 1.06 | 1.42 | 2.46 | 2.61 | **2.70** |
| 64 | 0.34 | 0.61 | 1.10 | 1.48 | 2.44 | 2.62 | 2.67 |
| 128 | 0.33 | 0.64 | 1.15 | 1.51 | 2.49 | 2.61 | 2.66 |
| 512 | 0.35 | 0.64 | 1.16 | 1.53 | 2.46 | 2.57 | 2.57 |
| 1024 | 0.35 | 0.64 | 1.18 | 1.54 | 2.40 | 2.47 | 2.46 |
| 2048 | 0.35 | 0.64 | 1.17 | 1.54 | 2.28 | 2.36 | 2.31 |

(b) Machine 2

Table 1: Speedup factor (geometric mean for DaCapo) of blueprint profiling with deferred methods and AP over sequential analysis for different buffer capacities $B_{AP}$ and queue capacities $Q_{AP}$

impact of deferred methods on locality using hardware performance counters (Section 5.4). Fourth, we investigate the impact of extended object lifetime due to deferred methods on the consumption of heap memory and on garbage collection time (Section 5.5). Finally, we summarize the results obtained with a second dynamic analysis—data race detection—in order to confirm that deferred methods are beneficial to a wide range of different analyses (Section 5.6).

### 5.1 Evaluation Setup and Baseline

We use the benchmarks in the DaCapo suite (dacapo-2006-10-MR2)[4] as base programs in our evaluations. To ensure that the ending time of a benchmark run is not prematurely taken, we extend the benchmark harness to wait until all pending buffers have been processed. The measurements reported in this paper correspond to the median of 11 benchmark runs within the same JVM process in order to attenuate the perturbations due to class-loading, load-time instrumentation, and just-in-time compilation.

Our dynamic analyses are implemented as aspects in the AspectJ language. The analysis aspects are woven into the base programs with MAJOR [18] (i.e., MAJOR performs the required bytecode instrumentation), an aspect weaver that enables complete method coverage; that is, every method which has a bytecode representation is woven, including methods in the Java class library. Thanks to MAJOR, the analyses yield comprehensive information representing overall program execution (but excluding the execution of native code).

All measurements are collected on two different quad-core machines with different micro-architectures, an Intel Core i7 Q720 (1.6 GHz, 8 GB RAM) referred to as *Machine 1* in this section, and an Intel Core2 Quad Q9650 (3.0 GHz, 8 GB RAM) referred to as *Machine 2*. We disable frequency scaling on both machines and also disable hyper-threading on Machine 1. Both machines run under Ubuntu GNU/Linux 10.04 and we use Oracle's JDK 1.6.0_20 Hotspot Server VM (64 bit) with 7 GB maximum heap size and with the default garbage collector. As discussed in Section 4, we configured the TP and AP buffer processing strategies to use a fixed thread pool of four threads which is an appropriate choice on our quad-core machines.

Traditional, sequential analysis serves us as a *baseline* for assessing the speedup thanks to deferred methods. In comparison with the execution of the unmodified benchmarks without any instrumentation, the baseline introduces an average overhead of factor 10.58 (resp. factor 13.07) on Machine 1 (resp. on Machine 2) for blueprint profiling, and an average overhead of factor 28.67 (resp. factor 35.51) on Machine 1 (resp. on Machine 2) for data race detection; the average is computed as the geometric mean of the measurements for all DaCapo benchmarks.

[4] http://www.dacapobench.org/

| | SP | TP | | AP | |
|---|---|---|---|---|---|
| | $B_{SP}$ | $B_{TP}$ | $Q_{TP}$ | $B_{AP}$ | $Q_{AP}$ |
| Machine 1 | 32768 | 8192 | 2048 | 16384 | 64 |
| Machine 2 | 32768 | 8192 | 2048 | 32768 | 32 |

Table 2: Parameters of SP, TP, and AP for blueprint profiling

### 5.2 Impact of Buffer and Queue Capacities

In the following we investigate the performance impact of buffer capacity (needed for SP, TP, and AP) and queue capacity (needed for TP and AP) on both machines. The values for these parameters that yield the highest average speedup (geometric mean for the DaCapo benchmarks) over our baseline are used in the experiments presented in the next subsections.

Table 1 presents the average speedup factor for deferred blueprint profiling with AP depending on different buffer capacities $B_{AP}$ and queue capacities $Q_{AP}$. Note that a more complete exploration of the parameter space was not possible due to the large number of time-consuming experiments. For each value in Table 1, $11 \times 11 = 121$ benchmark runs are needed (geometric mean of 11 benchmarks, for each benchmark taking the median of 11 runs within the same JVM process). Consequently, the data for some buffer and queue capacities the reader might expect (e.g., $B_{AP} = 2048$) are not presented here.

Nonetheless, Table 1 presents enough data to observe four trends for blueprint profiling with AP, which hold for both machines: (i) For a given buffer capacity $B_{AP}$, the speedup factor is rather stable across different queue capacities. (ii) For a given queue capacity $Q_{AP}$, the speedup factor increases with an increasing buffer capacity. (iii) When the buffer capacity is sufficiently large, (i.e., $B_{AP} \geq 16384$), small queue capacities yield higher speedups than larger queue capacities. (iv) Small buffer capacities may result in slowdowns with respect to the baseline (i.e., values below 1.0 in Table 1). The last observation also supports our claim that parallelizing small workloads often does not pay off.

As highlighted in Table 1, the highest speedup factor for deferred blueprint profiling with AP on the first machine is 2.62 (achieved with $Q_{AP} = 64$ and $B_{AP} = 16384$), whereas on the second machine, the highest speedup factor is 2.7 (achieved with $Q_{AP} = 32$ and $B_{AP} = 32768$). For the other buffer processing strategies (SP and TP), we gathered the best parameter settings on both machines in the same way. While the details of this study cannot be presented in this paper due to space limitations, Table 2 gives a summary of the parameter settings that yield the highest speedup factors.

### 5.3 Impact of Buffer Processing Strategies

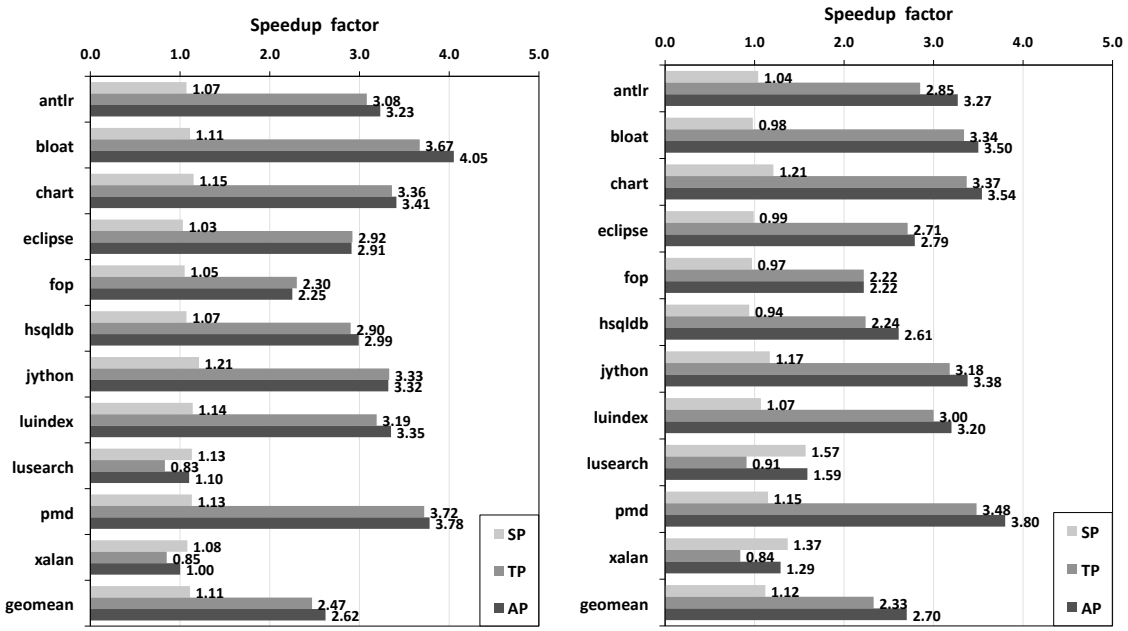
In the following we explore the performance impact of different buffer processing strategies. Figure 6 summarizes the speedup obtained with deferred methods (over the baseline) for the buffer processing strategies SP, TP, and AP (the concrete parameters are given in Table 2).

We analyze the results for the different buffer processing strategies in more detail by exploring CPU utilization (for the baseline and for deferred execution with SP, TP, and AP) and the number of pending buffers in the queue (for deferred execution with TP and AP). Because of space limitations, CPU utilization diagrams are presented only for two benchmarks on Machine 1, the single-threaded `bloat` (Figure 7) and the multi-threaded `lusearch` (Figure 8). The diagrams correspond to the third run of the respective benchmark within a single JVM process; hence, class-loading, load-time instrumentation, and just-in-time compilation do not significantly perturbate the measurements. The data is obtained by sampling CPU utilization every 200ms.

CPU utilization of the baseline is illustrated in Figure 7(a) for `bloat` and in Figure 8(a) for `lusearch`. For the single-threaded `bloat`, CPU utilization is almost constantly 100%; that is, the benchmark thread keeps one of the four available cores busy; temporarily reduced CPU utilization is due to blocking actions such as I/O. For the multi-threaded `lusearch`, CPU utilization is about 400% most of the time; that is, the benchmark executes a well-parallelized workload that does not leave much under-utilized CPU resources.

SP. As shown in Figure 6, the use of deferred methods with SP (which does not involve any parallelization) outperforms the baseline for many benchmarks, in spite of the extra overheads due to buffer management. This result suggests that buffering improves locality; this claim is validated later in Section 5.4.

CPU utilization for deferred analysis with SP is illustrated in Figure 7(b) for `bloat` and in Figure 8(b) for `lusearch`. While the CPU utilization is similar to the baseline, there are a few periods where CPU utilization is reduced. We found this reduction to be caused by extra garbage collections due to the increased memory




(a) Machine 1 (b) Machine 2

Figure 6: Speedup of blueprint profiling with deferred methods with SP, TP, and AP (over sequential analysis)

consumption because of buffer allocations. For the multi-threaded `lusearch`, multiple threads allocate buffers in parallel such that the impact of the extra garbage collections are more significant than for `bloat`.

TP. In contrast to SP (where the observed speedup is only due to buffering), the speedup with TP is because of buffering and parallelization. Figure 6 confirms that on average (geometric mean for DaCapo), deferred analysis with TP yields a speedup of factor 2.33–2.47 over the baseline.

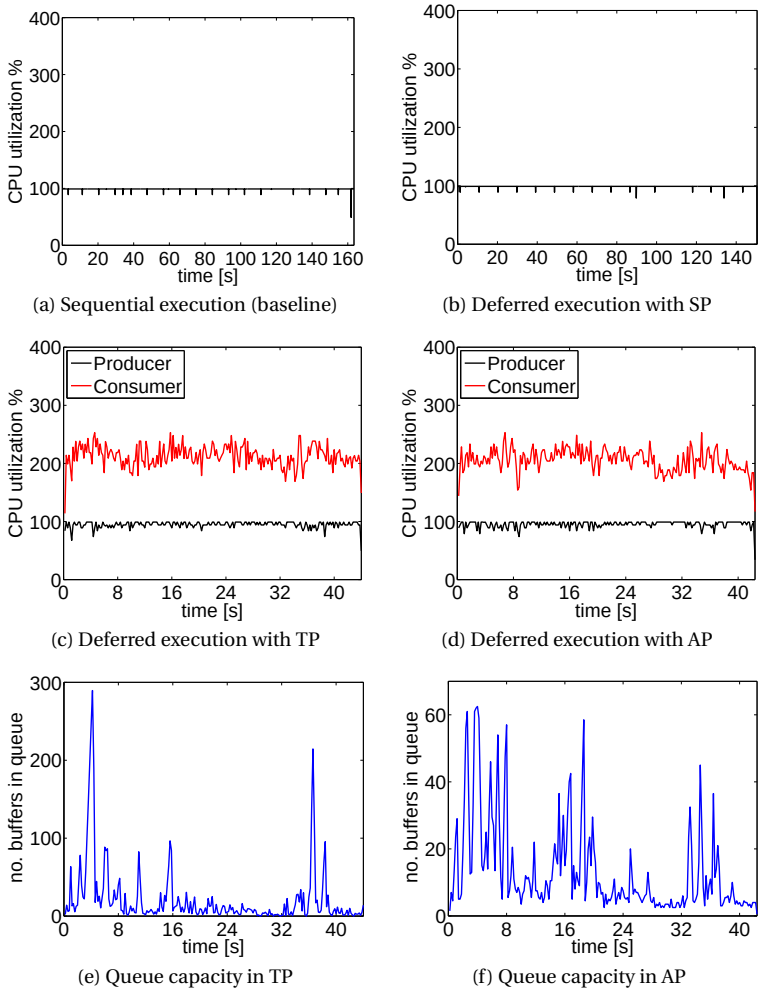
For `bloat` in Figure 7(c), the four consumer threads roughly use 200% of the CPU resources (i.e., two cores), in addition to the CPU utilization of the producer thread (in the base program) of roughly 100%. Figure 6(a) confirms that deferred analysis with TP yields high speedup of factor 3.67 for `bloat` on Machine 1. Figure 7(e) illustrates the number of pending buffers in the queue. There are always less than 300 pending buffers for `bloat`.

In contrast to `bloat`, `lusearch` does not benefit from deferred analysis with TP, because the producers constantly keep all cores busy; that is, there are no under-utilized cores the consumers could exploit. In fact, deferred analysis with TP performs worse than the baseline on both machines (speedup of factor 0.83 in Figure 6(a)). Figure 8(c) shows that the producers only utilize roughly 100% of the CPU resources, whereas the four consumers use up to 300%. Figure 8(e) indicates that the queue quickly becomes full ($Q_{TP} = 2048$); that is, producers are blocked waiting for free space in the queue. Consequently, in the case of `lusearch`, deferred analysis with TP results in many thread switches that introduce extra overhead. In addition, the number of garbage collections may increase because of the increased memory consumption due to pending buffers. As shown in Figure 6, `xalan`, another heavily multi-threaded benchmark, also performs badly under deferred analysis with TP. For all other benchmarks, which are either single-threaded or synchronized in a way that severely limits parallelism, deferred analysis with TP yields a considerable speedup of factor 2.22–3.72 on both machines.

AP. As shown in Figure 6, deferred analysis with AP yields the highest average speedup of factor 2.62–2.7 on both machines. It also achieves the maximum speedup of factor 4.05 for `bloat` on Machine 1 and of factor 3.8 for `pmd` on Machine 2.

For `bloat` in Figure 7, there is not much difference in CPU utilization between TP and AP. Still, the speedup with AP is a factor of 4.05 (in comparison to factor 3.67 with TP) on Machine 1 (Figure 6(a)). The number of pending buffers with AP is smaller than with TP, because $Q_{AP} = 64$, whereas $Q_{TP} = 2048$.

For `lusearch` in Figure 8, CPU utilization with AP is completely different from TP. Because the queue is




(a) Sequential execution (baseline)
(b) Deferred execution with SP
(c) Deferred execution with TP
(d) Deferred execution with AP
(e) Queue capacity in TP
(f) Queue capacity in AP

Figure 7: CPU utilization and number of buffers in queue for `bloat` on Machine 1

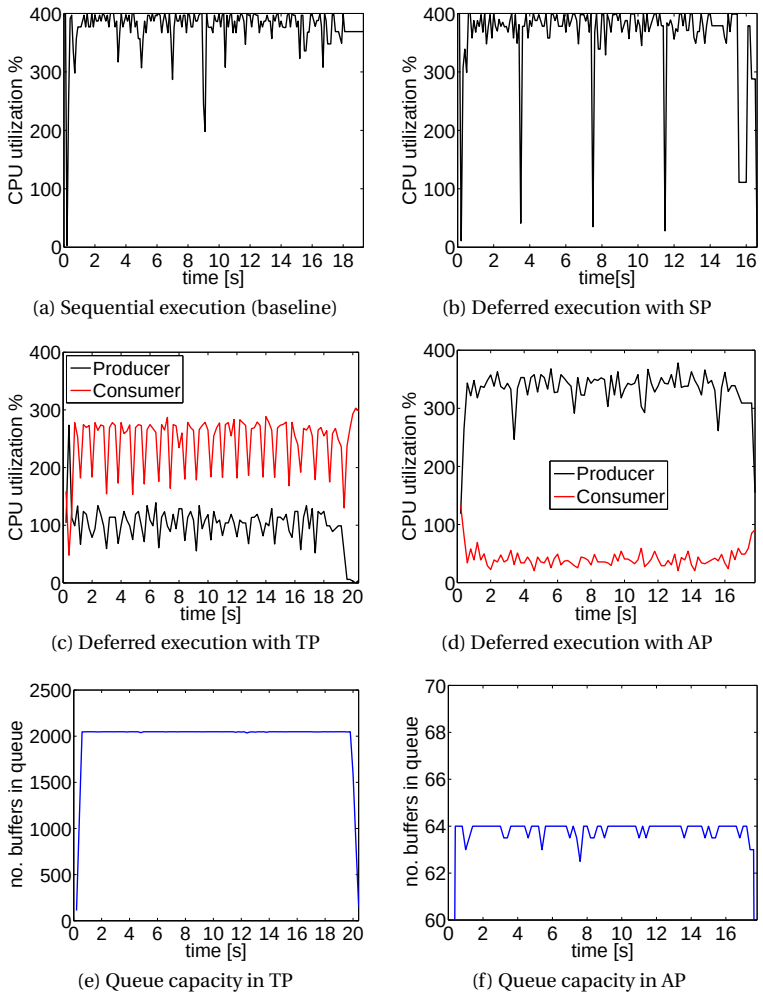
often full (Figure 8(f)), the producers process the full buffers most of the times; in contrast to TP, the producers are not blocked with AP. The consumers have lowest thread priorities and, therefore, execute rarely since the producers with higher priorities keep all cores busy. The producers use roughly 350% of the CPU resources. Figure 6(a) confirms that deferred analysis with AP yields a speedup of factor 1.10 for `lusearch` on Machine 1. In this particular case, SP still outperforms AP, but the difference in performance is small.

Since under AP all consumers run with lowest thread priorities (whereas under TP all consumers run with normal thread priorities), we also explore whether TP with lowest thread priorities for the consumers achieves comparable speedup as AP. However, through evaluating TP with lowest consumer priorities, we did not measure any significantly different speedup for any of the benchmarks on both machines. In fact, on average (geometric mean for DaCapo), the speedup for TP with normal consumer priorities is slightly higher on both machines.

We conclude that AP outperforms SP and TP for most benchmarks, as it achieves to reconcile the benefits of SP and TP. In those cases where SP or TP outperform AP, the difference in speedup between the winning strategy and AP is very small.

### 5.4 Impact on Locality

On the one hand, deferred methods introduce some sources of overhead, such as buffer allocation and garbage collection, storing to and loading from the buffer, and invocation of a buffer processor. On the other hand, deferred methods allow parallelization of small workloads and may improve locality. Locality improvement thanks to deferred methods is quite common in dynamic analysis, as the data structures accessed by analysis code are typically different from those accessed by the base program. Since the bodies of deferred methods are




(a) Sequential execution (baseline)
(b) Deferred execution with SP
(c) Deferred execution with TP
(d) Deferred execution with AP
(e) Queue capacity in TP
(f) Queue capacity in AP

Figure 8: CPU utilization and number of buffers in queue for `lusearch` on Machine 1

executed altogether when a buffer is processed, deferred methods may improve locality of the analysis code and sometimes also of the base program because the execution flow in the base program is less "disrupted" by storing data in a thread-local buffer than by performing analysis actions that may require expensive access to shared data structures.

To quantify these locality benefits, we use hardware performance counters to measure performance of the L1-data cache for deferred analysis with SP. As SP does not benefit from parallelization, the fact that the overheads of buffer management are not noticeable in Figure 6 suggests that they are outweighed by locality improvements.

Figure 9 summarizes the measured L1-data cache hit rate for the DaCapo benchmarks on Machine 1, comparing the baseline with deferred analysis using SP. The results show that deferred analysis with SP increases L1-data cache hit rate for all benchmarks. On average (geometric mean for DaCapo), the hit rate is 95.1% for the baseline, and 96.6% for deferred analysis with SP. While measurements of hardware performance counters confirm that other memory-related performance metrics are also improved, a detailed evaluation of such results is not in the scope of this paper.

### 5.5 Impact of Extended Object Lifetime

Two well-known issues concerning the use of buffers are increased heap memory consumption and increased object lifetime. As threads in the base program may perform blocking actions, filling a buffer can take arbitrarily long time. Before a full buffer has been processed, the garbage collector cannot reclaim any of the buffered objects, thus leading to a higher number of objects alive at the same time. In particular, for generational garbage collection, increased object lifetime can increase garbage collection time, because old objects are moved

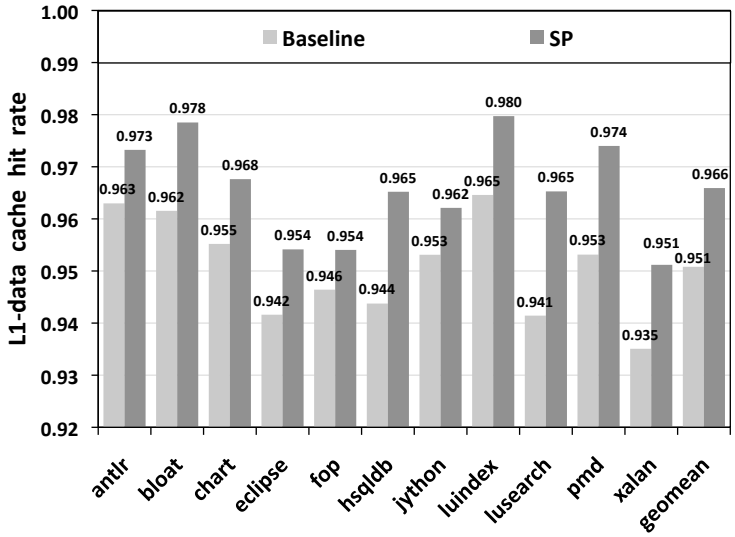



Figure 9: Impact of SP on L1-data cache hit rate on Machine 1

| | $SP$ | $TP$ | | $AP$ | |
|---|---|---|---|---|---|
| | $B_{SP}$ | $B_{TP}$ | $Q_{TP}$ | $B_{AP}$ | $Q_{AP}$ |
| Machine 1 | 16384 | 4096 | 2048 | 8192 | 64 |
| Machine 2 | 16384 | 4096 | 2048 | 16384 | 32 |

Table 3: Parameters of SP, TP, and AP for data race detection

to the next generation. In the case of blueprint profiling, each object allocated in the base program is stored in a buffer. Consequently, deferred analysis may possibly increase the lifetime of each allocated object.

Figure 10 illustrates the impact of an increasing buffer capacity on garbage collection time and on heap memory usage (average for DaCapo) with AP on Machine 1. The setting of $Q_{AP}$ corresponds to Table 2 (i.e., $Q_{AP} = 64$). For buffer capacities between 8192 and 32768, garbage collection time increases only slightly, whereas for larger buffer capacities, garbage collection time increases significantly. Heap memory usage remains rather stable. These results suggest that increased object lifetime due to deferred methods can have a strong negative impact on garbage collection time, whereas the extra heap memory consumption for the buffers is relatively insignificant.

We conclude that too large buffer capacities must be avoided because they can impair performance of the garbage collector. Similarly, with TP and AP, the queue capacity $Q_{TP}$ respectively $Q_{AP}$ must not be set too large.

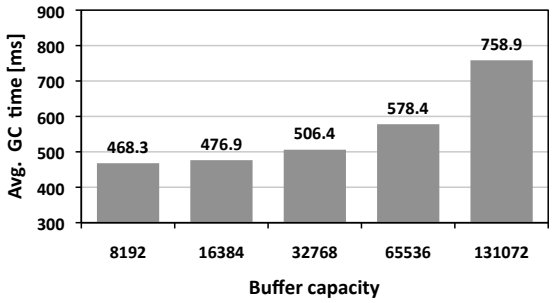
### 5.6 Data Race Detection

To confirm the applicability and effectiveness of deferred methods and of the presented buffer processing strategies, we evaluate a second dynamic analysis, data race detection. While space limitations prevent us from presenting the detailed evaluation results, in the following we summarize the achieved speedup over the baseline, a sequential implementation of data race detection.

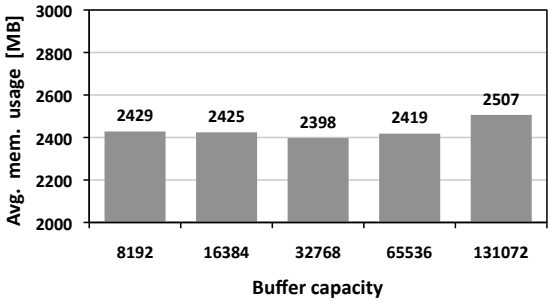
Data race detection is based on Racer [19], an adaptation of the Eraser algorithm [20] for Java. We refactored an existing parallelized implementation of Racer [11] to use deferred methods. While blueprint profiling acts upon method invocation, completion, and object allocation, data race detection operates upon field access, lock acquisition, and lock release.

Following the approach used for blueprint profiling, we first empirically determine the buffer and queue capacities that yield the highest average speedup (geometric mean for DaCapo) for SP, TP, and AP on Machine 1 and Machine 2; Table 3 summarizes these values. Table 4 presents the average speedup for the different buffer processing strategies.

The general trend of the speedup factors—that is, increasing from SP, TP, to AP—is observed similarly to the case of blueprint profiling. SP only gains marginal speedup on Machine 2, but results in a slight slowdown on Machine 1. In contrast, TP and AP achieve high speedup, and AP significantly outperforms TP on both machines. The speedup gains for all three buffer processing strategies are consistent across both machines.




(a) Average garbage collection time




(b) Average heap memory consumption

Figure 10: Impact of buffer capacity on garbage collection time and heap memory usage with AP on Machine 1

| | SP | TP | AP |
|---|---|---|---|
| Machine 1 | 0.99 | 1.98 | 2.22 |
| Machine 2 | 1.03 | 1.86 | 2.22 |

Table 4: Geometric mean of the speedup factor for data race detection.

The presented results from data race detection as well as blueprint profiler show that deferred methods together with the adaptive buffer processing strategy AP allow to effectively parallelize base program execution and dynamic analysis, yielding consistent speedup in different case studies and on different machines.

## 6 Related Work

Dynamic Analysis on Multicores. Offloading the analysis overheads to under-utilized cores to improve the performance of dynamic analyses is becoming increasingly popular in recent years with the widespread deployment of multicores [7, 10, 12, 9, 8, 11].

*PiPA* (*Pipelined Profiling and Analysis*) [7] applies pipelining techniques to parallelize profiling on multicores. It performs low overhead profiling in the same thread of the application to generate profiles, and then it uses several threads to rebuild the collected profile and and to analyze it. To reduce the costs of profiling and communication among the parallel threads, PiPA keeps the collected profiles and analysis in a buffer. In another work [10], a dynamic analysis framework is introduced which uses *CAB* (*Cache-friendly Asymmetric Buffering*), a lock-free ring buffer, to enable efficient communication between the base program and analysis threads. In contrast to our framework of deferred methods, although both of these approaches also use buffering to reduce thread communication overheads, they do not provide an API for programmers to develop parallelized dynamic analyses. On the other hand, our framework provides such an API and automatically generates the required code for parallelizing the base program and handling the buffers.

[12] also takes advantage of multicores to parallelize the profiling tasks with the base program. However, its approach to reduce the communication overheads is based on the observation that it is not necessary to communicate all the data items to the profiler and it itself can compute some needed information based on the already communicated data. As such, it benefits from compiler support to determine the set of data items to forward and the set to be computed by the profiler. On the contrary, our framework does not restrict the data items to be communicated to the profiler; instead, we store the data items in thread-local buffers and

communicate them altogether when a buffer becomes full.

Both *Shadow Profiling* [9] and *SuperPin* [8] are profiling techniques that fork off a shadow process that runs in parallel with the base program process. In these techniques, the shadow process executes the instrumented code while the thread of the base program runs the uninstrumented code. However, both Shadow Profiling and SuperPin are limited to single-threaded applications, target parallelism at the level of processes, and use a low-level instrumentation scheme without providing an API for developing parallelized applications.

The work in [11] introduces the notion of *buffered advice*, a mechanism for the AspectJ aspect-oriented programming language in which advice invocations that are marked with an `@Buffered` annotation are aggregated in a thread-local buffer, and processed altogether when this buffer is full. Compared to buffered advice, our framework provides the following improvements: (i) while buffered advice is limited to the AspectJ, deferred methods offer a general solution for Java; (ii) while buffered advice requires new language features, deferred methods do not depend on any new language constructs; (iii) while buffered advice supports only a single buffer with a singleton processor, deferred methods support several buffers with different processing strategies; (iv) deferred methods support processing check points; and (v) coalescing of deferred methods is possible.

Thread Management and Scheduling. Thread pools are widely adopted for task parallelization and performance improvement in modern applications [21]. For instance, [22] presents a number of examples in the domain of realtime middleware infrastructure where thread pools are used to improve the performance of programs on multicores.

[23] provides a framework based on dynamic performance tuning to determine where and how to create *speculative threads* at runtime in the context of the *Thread-Level Speculation* (*TLS*) execution model. TLS allows potentially dependent threads to run speculatively in parallel. The creation of speculative threads is based on the performance impact estimated by monitoring hardware counters of the speculative threads. Another work presented in [24] also uses dynamic feedbacks at runtime to determine the optimal number of threads for each loop in a parallel application on a SMT multiprocessor.

In the field of thread scheduling, there are many approaches showing the importance of correct thread-to-CPU mapping. For instance, [25] illustrates how and to what extent contention for shared resources, such as cache, memory controller, and memory bus can be mitigated via thread scheduling. [26] presents a compiler-supported approach to map an already parallelized application to cores. It uses machine learning techniques to model the machine behavior and to predict the optimal number of threads and optimal scheduling policy for any given program.

While all the aforementioned approaches work at low levels (e.g., operating system) to automatically determine the number of threads or to schedule them, our framework provides a high-level API for programmers and supports pluggable buffer processing strategies to explore under-utilized core.

## 7 Conclusion

The wide-scale spread of multicores offers a significant opportunity to improve the performance of dynamic analyses by parallelizing the analysis tasks with the base program execution. However, since these analyses are often fine-grained, the potential benefits of parallelism would otherwise be offset by the overheads of thread communication, plus the allocation and garbage collection of wrapper objects. This paper addressed this issue by presenting the framework of deferred methods, a simple, flexible API in standard Java that helps improve locality and reduce communication among the parallel threads by aggregating the invocations to analysis methods in thread-local buffers and processing them altogether. In this framework, while programmers can provide their own buffer processing strategies, the details of buffering are hidden and the required code is automatically generated.

We also presented an adaptive buffer processing strategy that adapts to the CPU utilization of the workload conveyed in buffers in order to exploit under-utilized cores when possible. A thorough performance evaluation with two different cases studies (a profiler and a data race detector) on two different quad-core machines with standard benchmarks confirmed that this framework is effective in improving the performance of dynamic analyses. In particular, our framework together with the adaptive buffer processing strategy resulted in an average speedup of factor 2.2–2.7 for our case studies.

For future work, we plan to extend this framework and automatically tune the capacity of buffers at runtime for improved performance. We also intend to explore the use of this framework in other application domains such as graphical user interfaces (GUIs).

References


[1] S. Artzi, S. Kim, and M. D. Ernst, "ReCrash: Making Software Failures Reproducible by Preserving Object States," in *ECOOP'08: Proceedings of the 22th European Conference on Object-Oriented Programming*, ser. Lecture Notes in Computer Science, J. Vitek, Ed., vol. 5142. Springer-Verlag, 2008, pp. 542–565.

[2] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *PLDI'07: Proceedings of the ACM SIGPLAN 2007 conference on Programming Language Design and Implementation*. ACM Press, 2007, pp. 89–100.

[3] A. Heydarnoori, K. Czarnecki, and T. T. Bartolomei, "Supporting framework use via automatically extracted concept-implementation templates," in *ECOOP'09: Proceedings of the 23rd European Conference on Object-Oriented Programming*, ser. Lecture Notes in Computer Science, S. Drossopoulou, Ed., vol. 5653. Springer-Verlag, 2009, pp. 344–368.

[4] A. Shankar, M. Arnold, and R. Bodik, "Jolt: lightweight dynamic analysis and removal of object churn," *ACM SIGPLAN Notices*, vol. 43, no. 10, pp. 127–142, 2008.

[5] N. R. Tallent and J. M. Mellor-Crummey, "Effective performance measurement and analysis of multithreaded applications," in *PPoPP'09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. ACM Press, 2009, pp. 229–240.

[6] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: a program query language," in *OOPSLA'05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 2005, pp. 365–383.

[7] Q. Zhao, I. Cutcutache, and W.-F. Wong, "PiPA: Pipelined profiling and analysis on multi-core systems," in *CGO'08: Proceedings of the 6th IEEE/ACM International Symposium on Code Generation and Optimization*. ACM Press, 2008, pp. 185–194.

[8] S. Wallace and K. Hazelwood, "SuperPin: Parallelizing dynamic instrumentation for real-time performance," in *CGO'07: Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2007, pp. 209–217.

[9] 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 5th IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2007, pp. 198–208.

[10] 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 Programing, Systems, Languages, and Applications*. ACM Press, 2009, pp. 155–174.

[11] D. Ansaloni, W. Binder, A. Villazón, and P. Moret, "Parallel dynamic analysis on multicores with aspect-oriented programming," in *AOSD'10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development*. ACM Press, 2010, pp. 1–12.

[12] G. He and A. Zhai, "Improving the performance of program monitors with compiler support in multi-core environment," in *IPDPS'10: Proceeding of the 24th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2010, pp. 1–12.

[13] M. D. Allen, S. Sridharan, and G. S. Sohi, "Serialization sets: a dynamic dependence-based parallel execution model," in *PPoPP'09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM Press, 2009, pp. 85–96.

[14] M. Wang, N. Benoit, F. Bodin, and Z. Wang, "Model driven iterative multi-dimensional parallelization of multi-task programs for the Cell BE: A genetic algorithm-based approach," in *PDP'10: Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE Computer Society, 2010, pp. 218–222.

[15] A. Bergel, R. Robbes, and W. Binder, "Visualizing dynamic metrics with profiling blueprints," in *TOOLS EUROPE'10: Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*, ser. Lecture Notes in Computer Science, J. Vitek, Ed., vol. 6141. Springer-Verlag, 2010, pp. 291–309.

[16] J. Gosling, B. Joy, G. L. Steele, and G. Bracha, *The Java Language Specification*, 3rd ed., ser. The Java Series. Addison-Wesley, 2005.

[17] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd ed. Addison-Wesley, 1999.

[18] A. Villazón, W. Binder, P. Moret, and D. Ansaloni, "Comprehensive aspect weaving for Java," *Science of Computer Programming*, 2010.

[19] E. Bodden and K. Havelund, "Racer: Effective race detection using AspectJ," in *ISSTA'08: Proceedings of the International Symposium on Software Testing and Analysis*. ACM Press, 2008, pp. 155–165.

[20] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems*, vol. 15, no. 4, pp. 391–411, 1997.

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

[22] Y. Zhang, C. Gill, and C. Lu, "Real-time performance and middleware on multicore linux platforms," Washington University in St Louis, Tech. Rep. WUCSE-2008-10, 2008.

[23] Y. Luo, V. Packirisamy, W.-C. Hsu, A. Zhai, N. Mungre, and A. Tarkas, "Dynamic performance tuning for speculative threads," in *ISCA'09: Proceedings of the 36th Annual International Symposium on Computer Architecture*. ACM Press, 2009, pp. 462–473.

[24] J. Lee, J.-H. Park, H. Kim, C. Jung, D. Lim, and S. Han, "Adaptive execution techniques of parallel programs for multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 70, no. 5, pp. 467–480, 2010.

[25] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," *ACM SIGPLAN Notices*, vol. 45, no. 3, pp. 129–142, 2010.

[26] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: a machine learning based approach," in *PPoPP'09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. ACM Press, 2009, pp. 75–84.