Deferred Methods: Accelerating Dynamic Program Analysis on Multicores

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

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

Abbas Heydarnoori
University of Lugano
Lugano, Switzerland
abbas.heydarnoori@usi.ch

Lydia Y. Chen
IBM Research
Zürich Laboratory
Rüschlikon, Switzerland
yic@zurich.ibm.com

ABSTRACT

Parallelization is attractive for speeding up dynamic program analysis on multicores. However, inter-thread communication overhead may outweigh any benefit from parallel execution. We propose deferred methods, a high-level Java framework to accelerate dynamic analysis on multicores. To minimize inter-thread communication overhead, invocations to analysis methods are automatically aggregated in thread-local buffers that are processed when full. In contrast to other approaches, our framework supports custom buffer processing strategies, eases pre-processing of buffers to reduce contention on shared data structures, and offers a synchronization mechanism to wait for the completion of previously invoked deferred methods. We also present a novel adaptive buffer processing strategy that parallelizes the analysis only when the observed workload leaves some CPU cores under-utilized. Using a profiler as case study, we show that deferred methods with the adaptive buffer processing strategy yield an average speedup of factor 4.09 on a quad-core machine. The speedup stems both from parallelization and from reduced contention.

Categories and Subject Descriptors

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

General Terms

Languages, Measurement, Performance

Keywords

Deferred methods, parallelization, dynamic program analysis, multicores.

1. INTRODUCTION

While the use of polymorphism, reflection, and dynamic code loading limits the applicability of static analysis in modern object-oriented programming languages, dynamic program analysis is becoming increasingly important because it supports a wide range of software engineering tasks such as debugging, performance optimization, and program comprehension [11]. However, the feasibility of complex dynamic analyses is often impaired by the large runtime overhead introduced in the base program, which in some cases can run more than 1000x times slower [13, 14]. In particular, this overhead often stems from frequent execution of fine-grained analysis tasks that do not produce any results on which the execution of the base program depends.

Modern shared-memory multicore systems offer potential for improving the performance of dynamic analyses by offloading the analysis tasks to under-utilized CPU cores and executing them in parallel with the base program. However, efficient parallelization of fine-grained tasks is not trivial, since the overhead from inter-thread communication may easily outweigh all benefits from parallelization. A possible solution to reduce communication overhead is to aggregate the invocations of analysis methods in thread-local buffers that are processed by helper threads when full [15, 7, 13, 2]. Compared to implementations based on standard thread pools, in this approach communication costs are paid only once per buffer rather than once per task.

In this paper, we propose deferred methods, a Java framework to efficiently parallelize fine-grained dynamic analysis tasks. As illustrated in Figure 1, upon invocation of a deferred method (e.g., methods *profCall(...)* and *profAlloc(...)*), instead of immediately executing the method body, the deferred method ID and all arguments are stored in a thread-local buffer. When a buffer becomes full, a customizable *processor* executes all tasks conveyed in the buffer; that is, the corresponding method bodies are executed using the buffered arguments. The processor is a pluggable component that implements custom buffer processing strategies. For example, the buffer can be synchronously processed by the same thread that has filled the buffer, or asynchronously processed by a helper thread.

Our framework provides a high-level, flexible API in standard Java. Thanks to automated code generation, low-level details of buffer management are hidden from the programmer. Nevertheless, the API provides users a set of primitives to customize the size and the processing of the buffers. Compared to other approaches, our framework introduces some new features to improve performance

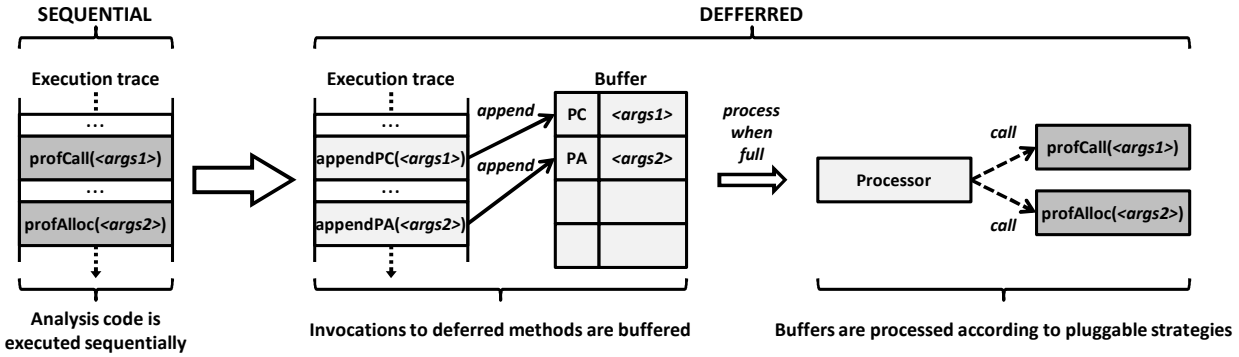


Figure 1: Comparison between sequential and deferred execution

of dynamic analysis tools based on buffering techniques. In particular, the novel contributions of this paper include:

- Description of a new buffer processing strategy that outperforms existing solutions by adapting to the CPU utilization of the base program, switching between synchronous and asynchronous processing depending on the amount of available resources.
- Provision of an interface to allow *coalescing* of buffers, that is, thread-local pre-processing of buffered data to reduce accesses to shared data structures, a common bottleneck for many dynamic analyses.
- Provision of an API for the creation of *processing checkpoints* to wait until all analysis data produced by a thread has been processed.
- A detailed study of the performance of deferred methods with different buffer capacities and processing strategies, of the benefits from coalescing, and of the impact of extended object lifetime on garbage collection.

Section 2 presents a motivating example which is used throughout this paper. Section 3 provides the API of our framework. 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.

2. MOTIVATING EXAMPLE

To illustrate the benefits of deferred methods, we chose a dynamic program analysis tool, profiling blueprints [3, 4], that executes only a relatively lightweight analysis task for each intercepted event. That is, parallelization of the analysis tasks is not trivial and naïve parallelization using standard thread pools results in enormous slowdowns due to excessive communication overhead.

Profiling blueprints provide 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. Figure 2 illustrates a simplified analysis 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[1]. In this figure, the class *ProfileTS* ("TS" stands for "thread-safe"), which is not shown for the sake of a condensed presentation of the code, stores this mapping data for each method *mid*. This class relies on thread-safe maps (i.e., *ConcurrentHashMap*) and atomic counters (i.e., *AtomicLong*) for the implementation.

BlueprintAnalysis 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. Method *onEntry(...)* is invoked by the instrumentation upon each method entry. By calling the *profCall(...)* 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, of type *MethodID*, is pushed onto the shadow stack.

Method *onExit()*, which is invoked by the instrumentation after (normal and abnormal) method completion, pops the completing method identifier off the shadow stack. Method *onObjectAllocation(...)*, which is invoked after object allocations, invokes the *profAlloc(...)* 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. However, it can be observed that it is not necessary to synchronously call *profCall(...)* and *profAlloc(...)* in methods *onEntry(...)* and *onObjectAllocation(...)* by the same thread that runs the base program. Since the execution of the base program 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, 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 12.31, while a naïve parallel implementation introduces an average overhead of 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.

[1]In this paper, the term 'method' refers to 'method or constructor'.

```
1   public interface Profile {
2       void profCall(MethodID caller, MethodID callee);
3       void profAlloc(MethodID mid, Object allocObj);
4       void integrate(Profile prof); // used in coalescing
5                                     // (see Section 3.5)
6       ...
7   }

8   public class ProfileTS implements Profile {
9       ... // Perform the analysis using thread-safe
10          // data structures
11  }

12  public class BlueprintAnalysis {
13      private final ThreadLocal<Stack<MethodID>> stackTL =
14          new ThreadLocal<Stack<MethodID>>() {
15              protected Stack<MethodID> initialValue() {
16                  return new Stack<MethodID>();
17              }
18          };

19      static final Profile profTS = new ProfileTS();

20      void onEntry(MethodID thisMID) {
21          Stack<MethodID> localStack = stackTL.get();
22          profTS.profCall(localStack.peek(), thisMID);
23          localStack.push(thisMID);
24      }

25      void onExit() { stackTL.get().pop(); }

26      void onObjectAllocation(Object allocObj) {
27          profTS.profAlloc(stackTL.get().peek(), allocObj);
28      }
29      ...
30  }
```

Figure 2: Simplified analysis class for blueprint profiling

3. DEFERRED METHODS

This section presents our framework with the help of the motivating example presented in Section 2. Figure 3 illustrates the architecture of our framework as a layered class diagram consisting of three kinds of classes and interfaces: (1) classes and interfaces that are provided by the framework API; (2) classes and interfaces that are implemented by the application developer; and (3) classes that are automatically generated by the framework at runtime. In the following text, we describe these layers in detail.

3.1 Framework API

As illustrated in Figure 3, our framework provides a simple, flexible API in standard Java for specifying deferred methods and for custom processing of buffers. To this end, the *Deferred* interface in Figure 3 is a marker interface; the programmer defines the methods that he wants to execute in a deferred way in an interface that extends this *Deferred* interface.

The interface *DeferredEnv* provides some methods that can be used by programmers to influence the management of buffers and processing of deferred methods. The buffer capacity (defined in terms of the number of entries) can be queried with the method *getBufferCapacity()*, is initially specified through a system property, and can be changed at runtime by calling *setBufferCapacity(int)*. Changing the buffer capacity influences subsequent buffer allocations and it does not change the capacity of the current buffer. Although our framework automatically processes a buffer when it becomes full, the 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.

Finally, the method *createDeferredEnv(...)* of class *DeferredExecution* creates 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 multiple times to create several instances of the deferred environment with different buffers and processors for various needs.

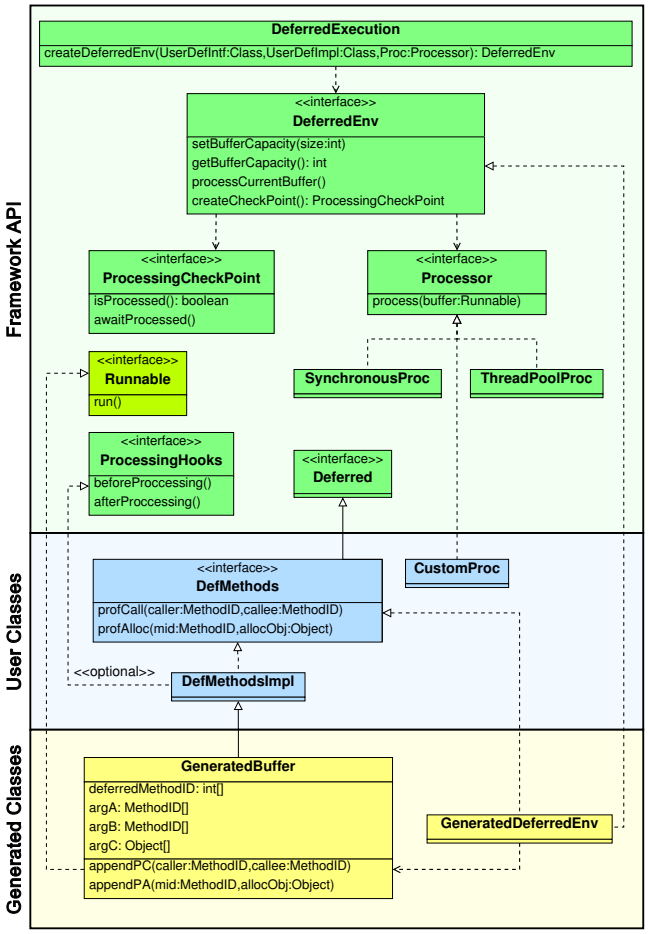


Figure 3: Architecture of deferred methods framework

3.2 User Classes

The code illustrated in Figure 4 implements the class diagram in Figure 3, showing how to refactor the code in Figure 2 to use deferred methods. The programmer extends the *Deferred* interface with the methods that should be deferred and then implements the extended interface. All deferred methods must return *void* and all data required by the analysis (e.g., objects allocated by the base program) must be passed as method arguments. As represented both in the refactored code (Figure 4) and in the class diagram (Figure 3), a custom interface named *DefMethods* 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(...)* invoke the corresponding methods in *ProfileTS*.

3.3 Automatically Generated Classes

Based on the user class that implements the deferred methods (i.e., *DefMethodsImpl* in Figure 3), our framework generates a buffer class, named *GeneratedBuffer*, that implements the *Runnable* interface. Additionally, based on the deferred interface provided

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

5  public class DefMethodsImpl implements DefMethods {
6      static final Profile profTS = new ProfileTS();

7      public void profCall(MethodID caller, MethodID callee) {
8          profTS.profCall(caller, callee);
9      }

10     public void profAlloc(MethodID mid, Object allocObj) {
11         profTS.profAlloc(mid, allocObj);
12     }
13 }

14 public class DefBlueprintAnalysis {
15     private static final int NUM_THREADS = 4;

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

20     private final ThreadLocal<Stack<MethodID>> stackTL =
21         new ThreadLocal<Stack<MethodID>>() {
22             protected Stack<MethodID> initialValue() {
23                 return new Stack<MethodID>();
24             }
25         };

26     void onEntry(MethodID thisMID) {
27         Stack<MethodID> localStack = stackTL.get();
28         def.profCall(localStack.peek(), thisMID);
29         localStack.push(thisMID);
30     }

31     void onExit() { stackTL.get().pop(); }

32     void onObjectAllocation(Object allocObj) {
33         def.profAlloc(stackTL.get().peek(), allocObj);
34     }
35     ...
36 }
```

Figure 4: Blueprint profiling using deferred methods

by the user (i.e., *DefMethods*), our framework generates the class *GeneratedDeferredEnv*. At runtime, this class acts similar to a *dynamic proxy* and redirects each invocation of a deferred method (e.g., line 28 in Figure 4) to the corresponding method (e.g., *appendPC(...)*) in class *GeneratedBuffer*, which in turn appends that invocation to the current thread's buffer for later processing.

The buffer content is stored in several arrays. The integer array *deferredMethodID* 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 *MethodID*. Hence, the array *argA* in the buffer holds the first arguments of invocations to both of these methods.

3.4 Buffer Processing

When a buffer becomes full, it is automatically submitted to the specified processor for processing via calling the processor's *process(buffer)* method and a new instance of the buffer is created then. In addition, our framework modifies the *run()* method of class *java.lang.Thread* and of all its subclasses so as to submit the thread's buffer, even if it is not full, before the thread terminates. 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 may be provided by the programmer. This is represented by class *CustomProc* in the second layer of Figure 3. For convenience, our framework includes processor implementations that are suited for various applications. Section 4 provides a detailed discussion of these processing strategies.

```
1  public class ProfileTL implements Profile {
2      ... // Perform the analysis using thread-local
3          // data structures
4  }

5  public class DefMethodsImpl implements DefMethods,
6                                         ProcessingHooks {
7      private ProfileTL profTL;
8      static final Profile profTS = new ProfileTS();

9      public void beforeProcessing() {
10         profTL = new ProfileTL();
11     }

12     public void afterProcessing() {
13         profTS.integrate(profTL);
14     }

15     public void profCall(MethodID caller, MethodID callee) {
16         profTL.profCall(caller, callee);
17     }

18     public void profAlloc(MethodID mid, Object allocObj) {
19         profTL.profAlloc(mid, allocObj);
20     }
21 }
```

Figure 5: DefMethodsImpl based on coalescing

3.5 Coalescing

A unique feature of our framework is that users can implement the *ProcessingHooks* interface to receive callbacks before and after processing the deferred method invocations. The main use case of this interface is to support the implementation of custom strategies for *coalescing* deferred method invocations. Instead of accessing shared data structures each time that a deferred method is invoked, one can define some instance fields in class *DefMethodsImpl* to record the invocations in thread-local data structures. 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 structures in the *afterProcessing()* method. In this way, we prevent the shared data structures from becoming a bottleneck in parallelized access.

For example, Figure 5 presents an updated implementation of class *DefMethodsImpl* from Figure 4 that supports the coalescing of deferred method invocations. To this end, it uses an optimized, non-thread-safe implementation (i.e., based on *HashMap* data structures and *long* counters) of the interface *Profile* (defined in Figure 2), named *ProfileTL* ("TL" stands for "thread-local"). The same analysis that was presented in Figure 4 is used again in this implementation. As can be seen in Figure 5, an instance of class *ProfileTL*, i.e., *profTL*, is instantiated in the *beforeProcessing()* method. Then, all invocations of deferred methods are stored in thread-local data structures used in *profTL* instead of accessing the shared data structures in *profTS*. The results in *profTL* are then integrated into the *profTS* in the *afterProcessing()* method. As our evaluation in Section 5.4 shows, coalescing of deferred methods yields significant performance improvements in our case study.

3.6 Processing Checkpoints

Synchronization between the thread invoking a deferred method and the thread executing it can be achieved by passing a synchro-

nizer (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. However, in order to prevent deadlocks, it is essential that the programmer ensures processing of the current buffer (i.e., by calling *processCurrentBuffer()*) before invoking any blocking method of a synchronizer that has been passed to a deferred method.

Processing checkpoints are a novel feature of our framework that makes it easy to wait until all analysis data produced by a thread has been processed. The method *createCheckPoint()* of class *DeferredEnv* 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. 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 just until the deferred methods conveyed in 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.

The implementation is optimized for incurring only negligible overhead when the feature is not used. A buffer may have at most one processing checkpoint associated. Hence, when a processing checkpoint is created, the current buffer has to be processed immediately, even though it may not be full. Therefore, frequent creation of processing checkpoints may incur high overhead because more buffers are created but not fully used. For each thread of the base program, the produced buffers are numbered (starting with buffer 1). The implementation keeps a counter C_T (initially zero) of consecutive buffers that have been processed for each thread T. After buffer n has been completely processed, C_T is incremented if its value is $n-1$; in this case, an eventual processing checkpoint associated with buffer n is set to the processed state. Otherwise, an entry containing the value n and an eventual processing checkpoint associated with buffer n is stored in the *heap*[2] data structure H_T which keeps track of buffers that have been processed while there is still at least one pending buffer with a smaller number (produced by thread T). Whenever C_T is increased to the new value x, the heap H_T is checked whether it contains the entry with the value $x+1$; in this case, the entry is removed from H_T, an eventual processing checkpoint stored in the entry is set to the processed state, C_T is incremented, and the heap check is repeated. Note that this implementation does not prevent processed buffers from being reclaimed by the garbage collector, since the entries stored on the heap do not refer to the buffers (but only to eventual processing checkpoints).

4. BUFFER PROCESSING STRATEGIES

We propose three strategies for processing full buffers: (1) *synchronous processing* (*SP*) by the thread that has filled the buffer, (2) *asynchronous processing* using a thread pool (*TP*), and (3) *adaptive processing* (*AP*) that reconciles synchronous and asynchronous processing. In the following text, 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. While the number of producers depends on the base program and on the scope of the analysis, the number of consumers is a controlled variable depending on the processing strategy.

[2]We use a heap because access to the smallest element is an $\mathcal{O}(1)$ operation.

4.1 Synchronous Processing (SP)

With SP, each full buffer is processed by its producer, i.e., 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.

Compared to a traditional, sequential dynamic analysis without deferred methods, SP introduces overhead due to buffer allocation, initialization, and garbage collection, as well as storing to and reading from the arrays in the buffer. However, if the deferred methods can be coalesced, the number of accesses to shared data structures in the analysis code can be reduced. Furthermore, as shown in Section 5.3, SP can effectively improve locality. Locality improvement thanks to deferred methods is quite common in dynamic analysis, as the data structures accessed by analysis code are often different from those accessed by the base program. Since the bodies of deferred methods are executed altogether when a buffer is processed, deferred methods 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 instead of performing analysis actions that may require expensive access to shared data structures.

4.2 Asynchronous Processing (TP)

With TP, 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 size $\mathcal{Q}_{TP}$. 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. However, when $\mathcal{Q}_{TP}$ is too small, producers may be blocked too frequently. 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 advantage of under-utilized cores by parallelizing execution of the base program (by producers) and dynamic analysis (by consumers). Similarly to SP, TP also benefits 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. 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 is a novel processing strategy that is unique in combining the benefits of TP and SP. On the one hand, when producers underutilize some cores, consumers can take advantage of available computational resources to process full buffers in parallel with the execution of the base program. On the other hand, when producers already keep all cores busy, full buffers are synchronously processed by their producers, avoiding inter-thread communication overhead when it is not possible to further parallelize the workload.

Similar to TP, AP uses a bounded FIFO queue of size $\mathcal{Q}_{AP}$, which however never blocks a producer. If the queue is full, the producer

itself processes the full buffer, like SP. Alternatively, buffers are passed to a pool of consumers, which execute 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 that the number of consumers N_{AP} is equal 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 significantly outperforms SP and TP in our evaluations, 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*). If other producers keep all cores busy, consumers may not be able to process full buffers in the queue. This problem is mitigated on operating systems where the scheduler dynamically changes thread priorities, so that low-priority threads eventually receive some CPU time (unless they are blocked).

5. EVALUATION

In this section we evaluate our framework using the blueprint profiler on a modern quad-core machine, typically used for tasks such as dynamic program analysis. Section 5.1 presents the setup of our evaluation. First, we investigate the speedup factor thanks to deferred methods with different buffer processing strategies compared to when no buffering is used (Section 5.2). Second, we explore the overhead of buffering (Section 5.3). Third, we analyze the benefits of coalescing the execution of deferred methods (Section 5.4). Fourth, we investigate the performance impact and CPU utilization of different buffer processing strategies (Section 5.5). Fifth, we explore the impact of extended object lifetime due to deferred methods on the number and the duration of garbage collection runs (Section 5.6).

5.1 Evaluation Setup

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. Our dynamic analysis is implemented as an aspect in the AspectJ language. The measurements reported in this article 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.

All measurements are collected on a quad-core machine with an Intel Core i7 Q720 1.6 GHz processor and 8 GB RAM in which we disable frequency scaling, turbo boost, and hyper-threading. We run this machine 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. We configure the TP and AP buffer processing strategies to use a fixed thread pool of four threads on our quad-core machine.

[3]We use method *setPriority(...)* in class *java.lang.Thread* to manipulate the priorities of the consumers before they are started.

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

Table 1: Speedup factor (geometric mean for DaCapo) of blueprint profiling with deferred methods and different buffer processing strategies over sequential analysis for different buffer capacity $\mathcal{B}$ and queue size $\mathcal{Q}$

Buffer capacity ($\mathcal{B}$)				
100	1000	10000	100000	1000000
0.94	0.95	0.97	**0.99**	0.97

(a) SP without coalescing (SP noC)

Buffer capacity ($\mathcal{B}$)				
100	1000	10000	100000	1000000
0.86	1.29	1.54	1.65	**1.69**

(b) SP with coalescing

Queue size ($\mathcal{Q}$)	Buffer capacity ($\mathcal{B}$)				
	100	1000	10000	100000	1000000
1	0.65	2.19	2.95	3.22	3.72
10	0.75	2.31	2.97	3.40	**3.79**
100	1.01	2.43	2.97	3.45	3.32
1000	1.37	2.78	3.23	3.14	-
10000	1.41	2.80	2.83	-	-
100000	1.27	1.93	-	-	-

(c) TP with coalescing

Queue size ($\mathcal{Q}$)	Buffer capacity ($\mathcal{B}$)				
	100	1000	10000	100000	1000000
1	0.75	1.99	3.01	3.68	**4.09**
10	1.29	2.45	3.26	3.74	4.02
100	1.41	2.70	3.37	3.70	3.62
1000	1.75	2.96	3.32	3.47	-
10000	1.75	2.95	2.93	-	-
100000	1.20	2.11	-	-	-

(d) AP with coalescing

5.2 Impact of Buffer and Queue Size

In the following text, we first investigate the performance impact of the chosen buffer capacity (for SP, TP, and AP) and queue size (only for TP and AP), both measured in terms of the number of entries. The 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. Then, we explore the performance impact of different buffer processing strategies.

Table 1 presents the average speedup factor for deferred blueprint profiling with various buffer processing strategies depending on different buffer capacity $\mathcal{B}$ and queue size $\mathcal{Q}$. The *baseline* for assessing the speedups is the traditional, sequential execution that does not make use of deferred methods. Measurements for blueprint profiling with SP and without coalescing are presented in Table 1(a). In the rest of this article, we refer to this setting as SP noC, whereas SP, TP, and AP refer to settings with coalescing.

In Table 1, empty cells refer to settings producing out-of-memory errors on some of the benchmarks, while the numbers in bold indicate the parameters that yield the highest speedups. These numbers are also presented diagrammatically in Figure 6 to better compare different buffer processing strategies. In Figure 6, the *naïve* implementation simply uses a thread pool without any buffering. This approach is extremely slow with an overhead of factor 190.

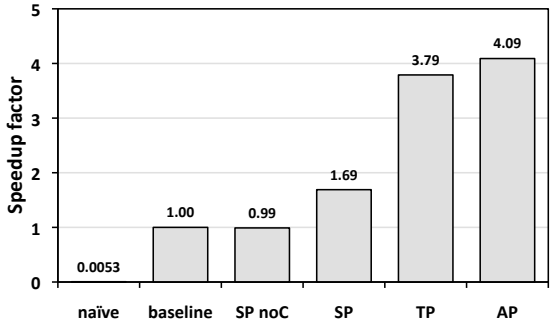


Figure 6: Speedup factor (geometric mean for DaCapo)

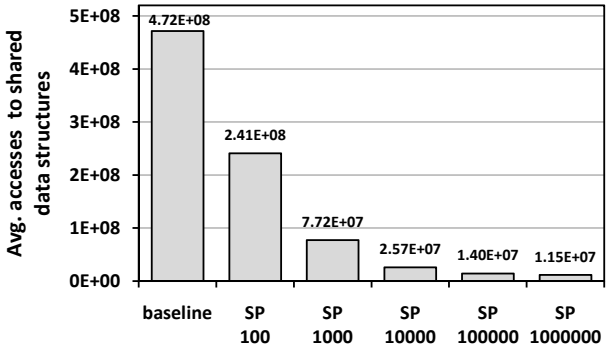


Figure 7: Average accesses to shared data structures

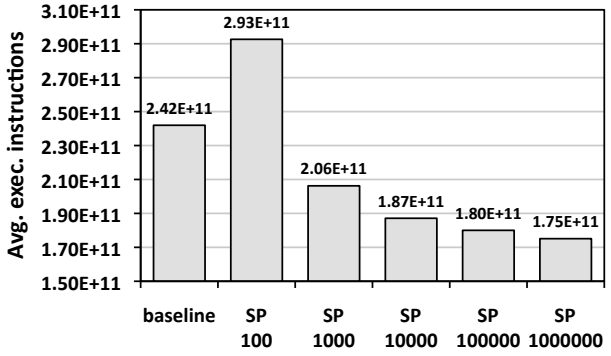


Figure 8: Average number of executed CPU instructions

For all buffer processing strategies except SP noC, we can see performance improvements in Table 1 and Figure 6. The speedup with SP is due to coalescing and buffering whereas the speedups with TP and AP are because of coalescing, buffering, and parallelization.

As shown in Table 1 and Figure 6, AP outperforms SP and TP since it aims at combining the benefits of SP and TP. Due to both coalescing and parallelization, AP achieves a speedup of factor 4.09 (with $\mathcal{Q} = 1$ and $\mathcal{B} = 1,000,000$). If we focus on the parameters of AP in Table 1, we observe three trends for blueprint profiling with AP: (i) for a given buffer capacity $\mathcal{B}$, the speedup factor is rather stable across different queue sizes; (ii) when the buffer capacity is sufficiently large, small queue sizes yield higher speedups than larger queue sizes; and (iii) 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.

In summary, SP compared to SP noC illustrates the benefits of coalescing; TP compared to SP presents the benefits due to parallelization; and finally, AP compared to TP indicates the benefits of a flexible strategy that does not offload the buffer to a consumer thread when the queue is full.

5.3 Buffering Overhead

As shown in Table 1(a) and Figure 6, the speedup factor of SP noC over the baseline is almost one (i.e., 0.99). Since SP noC does not benefit from parallelization, the fact that the overheads of buffer management are not significant suggests that they are outweighed by locality improvements. To quantify these locality benefits, we used hardware performance counters to measure performance of the L1-data, L1-instruction, and L2 caches for deferred analysis with SP noC. The results show that deferred execution with SP noC reduces miss rates, particularly for L1-instruction caching. On average (geometric mean for DaCapo), the miss rate is reduced by 0.61% for L1-data, 1.32% for L1-instruction, and 0.63% for L2. These relatively small improvements can significantly improve performance, because millions of instructions are executed by the CPU.

5.4 Impact of Coalescing

In this section we explore the impacts of coalescing the executions of deferred methods. Table 1(b) and Figure 6 illustrate that SP with coalescing is considerably faster than SP noC, i.e., a speedup of factor 1.69 vs. 0.99. The best speedup factor with SP noC is reached for $\mathcal{B} = 100,000$. However, with SP, it is possible to use buffers with bigger capacities (e.g., $\mathcal{B} = 1,000,000$) because the costs of increased memory utilization are compensated by the speedups due to coalescing[5].

Coalescing the execution of deferred methods has two main benefits: it reduces the number of expensive accesses to shared data structures and consequently reduces the overall number of executed CPU instructions. Figure 7 reports the number of accesses to shared data structures for the baseline (i.e., no deferred methods) and for SP with different buffer capacities. For example, in our case study, accesses to shared data structures are reduced by 94.56% when coalescing buffers of capacity $\mathcal{B} = 10,000$. Moreover, Figure 8 shows the average number of executed CPU instructions. As we increase the buffer capacity, the average number of executed instructions decreases. This reduction is due to the reduced complexity involved in accessing thread-local data structures that do not require synchronization (e.g., *HashMap*) compared to thread-safe implementations (e.g., *ConcurrentHashMap*). For the smallest measured buffer capacity of 100 entries, the extra instructions due to buffering are not compensated by coalescing.

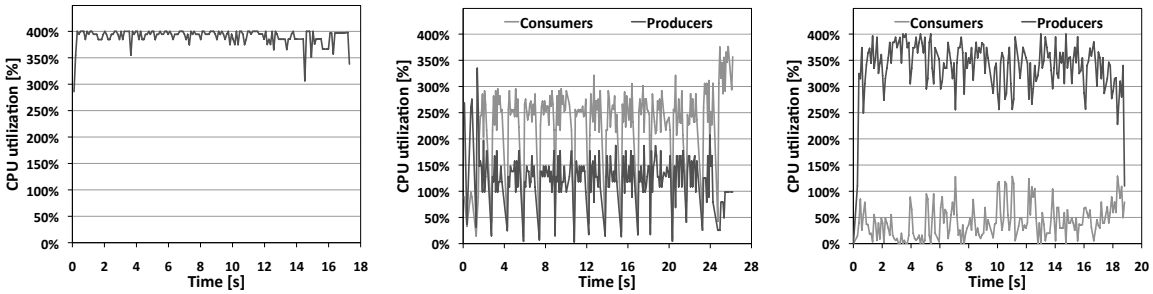
5.5 Impact of Parallelization

This section explores the performance impact of parallelization due to using the TP and AP buffer processing strategies. In contrast to SP, which only benefits from buffering and coalescing, these approaches also leverage under-utilized CPU cores to parallelize the execution of deferred methods. As can be seen in Table 1(c) and Table 1(d), these approaches yield much better speedup factors when compared to SP. More specifically, careful considera-

[5]Interestingly, the best results for SP, TP, and AP are achieved with large buffer capacities, where whole buffers cannot fit in the largest cache of the system.

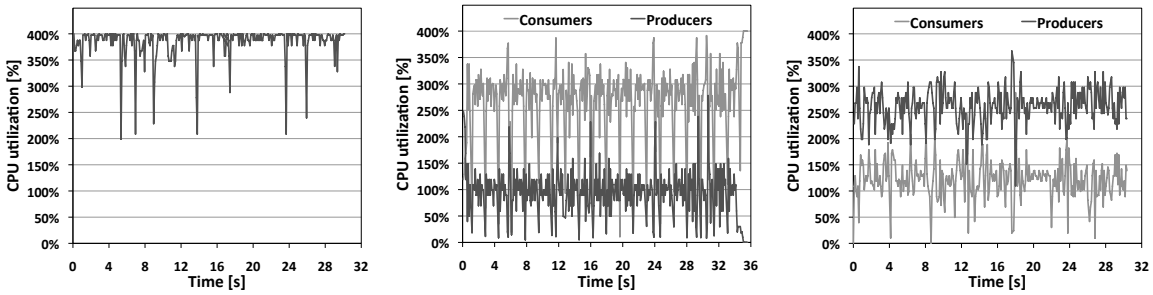
Table 2: CPU utilization for the DaCapo benchmarks

Benchmark	SP		TP			AP		
	Elapsed time [s]	Idle CPU [%]	Parallelism speedup factor	Idle CPU [%]	CPU used by consumers [%]	Parallelism speedup factor	Idle CPU [%]	CPU used by consumers [%]
antlr	26.53	292	3.21	57	229	3.29	54	227
bloat	164.43	299	3.38	38	260	3.72	30	258
chart	63.49	299	3.11	88	210	3.20	74	208
eclipse	154.69	297	2.56	129	163	2.67	115	163
fop	8.77	288	2.34	124	152	2.41	120	150
hsqldb	35.29	289	2.24	133	144	2.31	129	143
jython	78.03	271	2.95	56	208	3.08	48	206
luindex	64.88	299	2.71	105	191	2.87	97	190
lusearch	17.75	14	0.68	47	237	0.94	13	51
pmd	75.12	298	3.43	41	253	3.46	35	251
xalan	30.30	7	0.85	7	251	1.00	5	124



(a) Execution with SP (b) Execution with TP (c) Execution with AP

Figure 9: CPU utilization of application threads (producers) and analysis threads (consumers) for lusearch



(a) Execution with SP (b) Execution with TP (c) Execution with AP

Figure 10: CPU utilization of application threads (producers) and analysis threads (consumers) for xalan

tion of Figure 6 indicates that parallelization improves the speedup over SP with a factor of $3.79/1.69 = 2.24$ for TP, and a factor of $4.09/1.69 = 2.42$ for AP. These speedup factors due to parallelization are particularly good since some of the computational resources are already used by the non-parallelizable parts of the execution such as the base program or the computation necessary to fill and submit the buffers.

Table 2 presents a detailed speedup analysis due to parallelization for different benchmarks in the DaCapo benchmark suite. A careful study of this table indicates that the parallelization speedups that are reached with TP and AP are closely related to the amount of available CPU resources with SP. For *lusearch* and *xalan*, which are characterized by a high concurrency level, parallelization using TP or AP is not beneficial. For these two benchmarks, the consumers in TP or AP are taking CPU resources from the application threads which results in reduced performance.

Below, we provide a detailed analysis of CPU utilization for deferred execution with SP, TP, and AP. For space limitations, we focus our analysis on *lusearch* (Figure 9) and *xalan* (Figure 10), the critical cases in which TP or AP cannot outperform SP. The diagrams correspond to the sixth run of the respective benchmark within a single JVM process; hence, class-loading, load-time instrumentation, and just-in-time compilation do not significantly perturb the measurements. The data is obtained by sampling CPU utilization every 100ms.

Synchronous Processing (SP) CPU utilization with SP is illustrated in Figure 9(a) for *lusearch* and in Figure 10(a) for *xalan*. In both cases, CPU utilization is almost constantly 400%; that is, producer threads keep all available cores busy. Temporarily reduced CPU utilization is mainly due to garbage collection or blocking actions such as I/O. This implies that both *lusearch* and *xalan* execute a well-parallelized workload that does not leave much

under-utilized CPU resources; that is, producer threads keep all available cores busy.

Thread-Pool Processing (TP) Table 2 and Figure 9(b) show that in *lusearch* the producers only utilize 116% of CPU time on average, whereas the four consumers use an average of 237%. This is because producers are very often blocked waiting for free space in the queue. Consequently, deferred analysis with TP causes frequent 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 Table 2, *xalan*, another heavily multi-threaded benchmark, also performs badly under deferred analysis with TP. Figure 10(b) shows that in *xalan* the producers only utilize roughly 142% of the CPU resources, whereas the four consumers use up to 251%. 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 2.24–3.43 over SP.

Adaptive Processing (AP) Table 2 shows that in *lusearch* deferred analysis with AP yields a speedup of factor 0.94 over SP in *lusearch*. In this particular case, SP still outperforms AP, but the difference is relatively small. However, as depicted in Figure 10(c), in *xalan* producers use roughly 271% of the CPU resources and AP reaches the same level of performance as SP. Figure 9(c) shows that in *lusearch* CPU utilization with AP is completely different from TP. While in TP producers use 116% of CPU time, in AP they use 336%. Since the queue is often full, the producers process the buffers themselves most of the time. We also explore whether TP can benefit from lowest thread priority for the consumers, as it happens for AP. However, on average, this solution introduces some overhead compared to the standard implementation of TP.

We conclude that AP outperforms SP and TP for most benchmarks, reconciling the benefits of SP and TP. Even for *lusearch*, where SP outperforms AP, the performance of the two processing strategies is very similar.

5.6 Impact of Garbage Collection

One of the well-known issues concerning the use of buffers is extended 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 to the next generation. In our case study, each object allocated in the base program is stored in a buffer. Consequently, deferred analysis may possibly increase the lifetime of each allocated object.

To find out whether extended object lifetime is really an issue with our framework, we evaluate the impact of different buffer capacities and queue sizes on the duration of garbage collection runs with AP. Table 3 illustrates the results of this analysis (average for the DaCapo benchmarks). Since SP and TP follow similar trends in our measurements, their results are not provided here. All empty cells in Table 3 refer to tests that produce out-of-memory errors in at least one benchmark. In general, for each queue size, the garbage collection time is less than 15% until the total number of entries (i.e., the product of $\mathcal{Q}$ and $\mathcal{B}$) is higher than 1,000,000. These results suggest that increased object lifetime due to deferred methods can have a strong negative impact on garbage collection time, which grows proportionally to the product of $\mathcal{Q}$ and $\mathcal{B}$. We conclude that it is preferable to choose a small size for the queue and the highest possible capacity for the buffer, to leverage coalescing and to improve locality. It is up to the user to tune these parameters, depending on the number of arguments passed to deferred methods, which defines the memory footprint of a single buffer entry. If the analysis requires a large amount of data as input, the benefits from deferred methods may be limited.

Table 3: Average percentage of garbage collection time with AP depending on buffer capacity and queue size

Queue Size ($\mathcal{Q}$)	Buffer Capacity ($\mathcal{B}$)				
	100	1000	10000	100000	1000000
1	0.43	1.02	1.51	2.14	7.33
10	0.71	1.27	1.64	2.17	7.61
100	0.78	1.41	1.75	2.59	33.33
1000	1.08	1.95	2.45	16.05	-
10000	2.40	11.19	22.20	-	-
100000	52.88	75.94	-	-	-

6. RELATED WORK

Approaches based on *parallelized slice profiling*, such as Shadow Profiling [9] and SuperPin [14], parallelize dynamic analysis by periodically forking a shadow process that executes a slice of instrumented code while the application process runs uninstrumented code. Both frameworks are based on a dynamic binary instrumentation system, and both are limited to single-threaded applications. This limitation stems from the implementation of fork on most thread libraries, which can only fork from the current thread. Unlike these approaches, deferred methods do not pose any constraint to the internal concurrency of the base program.

PiPA (*Pipelined Profiling and Analysis*) [15] is a technique for parallelizing dynamic analysis by associating an analysis pipeline with each application thread. Analysis data is collected within the application thread and stored in a thread-local buffer. Once full, the buffer is passed to the pipeline, where multiple helper threads perform the analysis as stages of the pipeline. Compared to deferred methods, PiPA is based on a dynamic binary instrumentation system, does not provide a high-level API, and does not support adaptive processing strategies, coalescing, and processing checkpoints.

CAB (*Cache-friendly Asymmetric Buffering*) [7], is a dynamic analysis framework based on lock-free ring buffers to communicate analysis data from the base program to analysis threads. This approach is implemented at the JVM level and efficiently exploits shared caches of multicore systems. However, CAB does not allow adaptive buffer processing strategies and does not provide high-level constructs for coalescing and processing checkpoints. The implementation of lock-free ring buffers could be used in our framework to further speed up dynamic analysis on the supported JVM.

Buffered advice [2] allow asynchronous execution of AspectJ advice that are marked with a special annotation. Similar to deferred methods, input parameters are stored in thread-local buffers that are processed when full. However, buffered advice is specific to AspectJ, requires a custom compiler, and does not support custom buffer processing strategies, coalescing, and processing checkpoints.

In [13], the authors introduce a framework based on a dynamic binary instrumentation system to speed up data collection by means of thread-local buffers. Similar to deferred methods, this work describes a high-level API that automatically generates the code to fill and submit the buffer. However, this work lacks important features for developers of dynamic analysis tools, such as coalescing, processing checkpoints, and the adaptive processing strategy.

HeapMon [12] uses an extra helper thread that runs on a separate processor to monitor the status of each word on the heap and to detect memory bugs such as reads from uninitialized or unallocated memory locations. HeapMon achieves low-overhead because of hardware buffering and instrumentation support such as augmenting each cached word with one extra state bit, communication queues between the application thread and the helper thread, and a small private cache for the helper thread. In contrast to the framework of deferred methods, HeapMon is thus a special-purpose tool aiming to monitor the memory at runtime.

ParCCT [5, 6] is a technique for parallelizing calling context profiling and application execution on multicores. ParCCT profiles calling contexts as *Calling Context Trees* (*CCTs*) [1]. In this technique, each thread maintains a *shadow stack* and generates packets of method calls and returns that correspond to partial CCTs. Packet consuming threads then merge these partial CCTs with the overall CCT of the program in parallel with its execution. While this technique is restricted only to calling context profiling, our API can be used to develop different dynamic analyses for various purposes.

From Java 7, the language includes the Fork/Join framework [8], which provides a thread pool implementation based on work-stealing [10] and eases scalable implementations of recursive divide-and-conquer algorithms. Compared to our work, the Fork/Join framework does not address the parallelization of fine-grained, independent tasks, where the overhead of allocating a wrapper object and submitting it to the thread pool outweighs the benefits from parallelization. Our framework allows the implementation of custom processors based on *ForkJoinPool*.

7. CONCLUSION

This paper addresses the issue of parallelizing fine-grained dynamic analysis tasks by means of deferred methods, a Java framework that helps reduce communication among the parallel threads by aggregating the invocations to analysis methods in thread-local buffers and processing them altogether. While programmers can provide their own buffer processing strategies, the details of buffering are hidden and the required code is automatically generated.

Distinguishing features of our framework are the support for coalescing and processing checkpoints. Moreover, we presented a novel buffer processing strategy that adapts to the overall CPU utilization. A thorough performance evaluation on a quad-core machine with standard benchmarks confirms that our framework, together with the adaptive buffer processing strategy and coalescing, yields an average speedup of factor 4.09 in our case study.

Acknowledgements: This work was supported by an IBM Shared University Research (SUR) award.

8. REFERENCES

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 85–96, 1997.

[2] D. Ansaloni, W. Binder, A. Villazón, and P. Moret. Parallel dynamic analysis on multicores with aspect-oriented programming. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 1–12, 2010.

[3] A. Bergel, F. Bañados, R. Robbes, and W. Binder. Execution profiling blueprints. *Software: Practice and Experience*, 2011.

[4] A. Bergel, R. Robbes, and W. Binder. Visualizing dynamic metrics with profiling blueprints. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*, volume 6141 of *Lecture Notes in Computer Science*, pages 291–309, 2010.

[5] W. Binder, D. Ansaloni, A. Villazón, and P. Moret. Parallelizing calling context profiling in virtual machines on multicores. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 111–120, 2009.

[6] W. Binder, D. Ansaloni, A. Villazón, and P. Moret. Flexible and efficient profiling with aspect-oriented programming. *Concurrency and Computation: Practice and Experience*, 23(15):1749–1773, 2011.

[7] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *Proceeding of the 24th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 155–174, 2009.

[8] D. Lea. A Java Fork/Join framework. In *JAVA '00: Proceedings of the ACM Conference on Java Grande*, pages 36–43, 2000.

[9] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *Proceedings of the 5th Symposium on Code Generation and Optimization*, pages 198–208, 2007.

[10] K. H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1998.

[11] D. Röthlisberger, M. Härry, W. Binder, P. Moret, D. Ansaloni, A. Villazón, and O. Nierstrasz. Exploiting dynamic information in IDEs improves speed and correctness of software maintenance tasks. *IEEE Transactions on Software Engineering*, PrePrint, 2011.

[12] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. HeapMon: a helper-thread approach to programmable, automatic, and low-overhead memory bug detection. *IBM Journal of Research and Development*, 50(2/3):261–275, 2006.

[13] D. Upton, K. Hazelwood, R. Cohn, and G. Lueck. Improving instrumentation speed via buffering. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 52–61, 2009.

[14] S. Wallace and K. Hazelwood. SuperPin: Parallelizing dynamic instrumentation for real-time performance. In *Proceedings of the 5th International Symposium on Code Generation and Optimization*, pages 209–217, 2007.

[15] Q. Zhao, I. Cutcutache, and W.-F. Wong. PiPA: Pipelined profiling and analysis on multi-core systems. In *Proceedings of the 6th International Symposium on Code Generation and Optimization*, pages 185–194, 2008.