Rapid Development of Extensible Profilers for the Java Virtual Machine with Aspect-Oriented Programming

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

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

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

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

ABSTRACT

Many profilers for Java applications are implemented with low-level bytecode instrumentation techniques, which is tedious, error-prone, and complicates maintenance and extension of the tools. In order to reduce development time and cost, we promote building Java profilers using high-level aspect-oriented programming (AOP). We show that the use of aspects yields concise profilers that are easy to develop, extend, and maintain, because low-level instrumentation details are hidden from the tool developer. Our profiler relies on inter-advice communication, an extension to common AOP languages that enables efficient data passing between advice woven into the same method. We illustrate our approach with two case studies. First, we show that an existing, instrumentation-based tool for listener latency profiling can be easily recast as an aspect. Second, we present an aspect for comprehensive calling context profiling. In order to reduce profiling overhead, our aspect parallelizes application execution and profile creation.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*

General Terms

Algorithms, Languages, Measurement, Performance

Keywords

Profiling, aspect-oriented programming, bytecode instrumentation, Calling Context Tree, concurrency, Java

1. INTRODUCTION

Bytecode instrumentation techniques are widely used for profiling [9, 8, 11, 6, 14, 7]. Java supports bytecode instrumentation using native code agents through the Java Virtual Machine Tool Interface (JVMTI) [18], as well as portable bytecode instrumentation through the `java.lang.instrument` API. Several bytecode engineering libraries have been developed, such as BCEL [19] and ASM [15]. However, because of the low-level nature of bytecode and of bytecode engineering libraries, it is difficult and error-prone to implement new profilers, which often requires high effort for development and testing. Moreover, profilers based on low-level instrumentation techniques are difficult to maintain and extend.

In order to ease, speed up, and reduce the cost of profiler development, maintenance, and extension, we resort to aspect-oriented programming (AOP) [13]. AOP allows concisely specifying instrumentations in a high-level manner. For example, an aspect can be used to specify profiling code to be executed before or after the invocation of some methods of interest, hiding the low-level details of bytecode instrumentation from the tool developer [16].

Profiling usually requires instrumentation with full method coverage, that is, comprehensive instrumentation reaching all methods executed in the Java Virtual Machine (JVM); otherwise, part of the program execution may be missing in the generated profiles. Prevailing aspect weavers, such as AspectJ [12] or *abc* [2], do not support comprehensive aspect weaving, notably because they prevent aspect weaving in the Java class library. Even though weaving of the Java class library can be forced in AspectJ, weaving core Java classes usually results in JVM crashes upon bootstrapping [5, 21].

Our approach to aspect-based profiler development relies on MAJOR [21, 22], an aspect weaver that complements AspectJ with support for comprehensive weaving. To this end, MAJOR leverages FERRARI[1] [5], a bytecode instrumentation framework that ensures full method coverage of any user-defined instrumentation.

We present an *inter-advice communication* model [20] for AOP, which has been integrated into MAJOR. Inter-

[1] `http://www.inf.usi.ch/projects/ferrari/`

advice communication allows for efficiently passing data between advice that are woven into the same method. With the aid of annotations, the aspect programmer can declare *invocation-local* variables, which correspond to local variables with the scope of a woven method. The inter-advice communication model is complementary to AspectJ constructs and enables important optimizations in our aspect-based profilers.

We demonstrate our approach with two case studies. First, we illustrate how the listener latency profiler LiLa [11], which is based on bytecode instrumentation using ASM [15], can be recast as an aspect using MAJOR and inter-advice communication.

Second, we present an aspect for calling context profiling. Calling context profiling helps analyse the dynamic inter-procedural control flow of applications. It is particularly important for understanding and optimizing object-oriented software, where polymorphism and dynamic binding hinder static analyses. The Calling Context Tree (CCT) [1] is a prevailing data structure for calling context profiling. It provides dynamic metrics, such as the number of method invocations, for each calling context. There is a large body of related work dealing with different techniques to generate CCTs [1, 17, 23], highlighting the importance of the CCT for calling context profiling. Our aspect yields complete CCTs representing overall program execution. In order to speed up CCT construction on multicores, we refine the aspect so as to parallelize CCT construction and program execution.

2. BACKGROUND

In this section we give a brief overview of AOP and summarize our prior work on which this paper builds.

2.1 Aspect-Oriented Programming

AOP [13] enables a clean modularization of crosscutting concerns in applications, such as error checking and handling, synchronization, monitoring, or logging. AOP helps avoid related code that is scattered throughout methods, classes, or components.

AspectJ [12] is a popular AOP language for Java, allowing new functionality to be systematically added to existing programs. In AspectJ, an *aspect* is an extended class with additional constructs. A *join point* is any identifiable execution point in a system (e.g., method call, method execution, object creation, or field assignment). Join points are the places where a crosscutting action can be inserted. The user can specify *weaving* rules to be applied to join points through so-called *pointcuts* and *advice*. A pointcut identifies or captures join points in the program flow, and the action to be applied is called advice.

AspectJ supports three kinds of advice: `before()`, `after()`, and `around()`, which are executed prior, following, or surrounding a join point's execution. Aspects are compiled into standard Java classes. In the aspect class, advice are compiled into methods. During the weaving process, the weaver inserts code in the woven class to invoke these advice methods. Advice can receive some context information, e.g., to identify which join point has been captured.

During the execution of a woven class, by default, a singleton instance of the aspect is instantiated. Several aspects can be woven simultaneously and can therefore coexist during the execution.

2.2 Prior Work

Our approach to profiler development relies on MAJOR [21, 22], an aspect weaver that enhances AspectJ with support for comprehensive weaving. That is, MAJOR is able to weave aspects into all classes loaded in the JVM, including dynamically loaded or generated classes, as well as the standard Java class library. MAJOR can weave aspects into all classes in the Java class library in a portable way, compatible with any standard, state-of-the-art JVM.

MAJOR deals with issues such as bootstrapping the JVM with a woven Java class library and preventing infinite recursions when an advice invokes methods in the woven Java class library [21]. To this end, MAJOR leverages FERRARI [5], a generic bytecode instrumentation framework that guarantees complete method coverage of user-defined instrumentations (after an initial JVM bootstrapping phase, which is completed before the application's `main(...)` method is invoked).

BMW [4] is a profiler generator based on bytecode instrumentation techniques. In contrast to MAJOR, the programming model of BMW is very much restricted. It provides only a small set of low-level pointcuts, which severely limits the kind of profilers that can be specified. Hence, BMW cannot be considered an AOP framework. In addition, BMW does not support instrumentation of the Java class library.

In [6, 14], we addressed platform-independent calling context profiling using a hard-coded, low-level instrumentation that was difficult to extend. Thanks to AOP and to our new inter-advice communication model, we can now concisely express profilers as aspects in just a few lines of code. The high-level specification of aspect-based profilers eases the implementation of optimizations, such as the parallelized CCT construction presented in this paper.

2.3 Inter-Advice Communication

In AspectJ, the `around()` advice, in conjunction with a `proceed()` statement, allows storing data in local variables before a join point, and accessing that data after the join point. The invocation of `proceed()` within an advice causes the execution of the captured join point. Hence, one common use of the `around()` advice can be regarded as communicating data produced in a `before()` advice to an `after()` advice, within the scope of a woven method.

However, there are two severe limitations when using the `around()` advice for communicating data across a join point:

1. The AspectJ weaver implements the `around()` advice by inserting wrapper methods in woven classes [10]. However, wrapping certain methods in the Java class library breaks stack introspection in many recent JVMs, including Sun's HotSpot JVMs and IBM's J9 JVM [14, 22].
2. The `around()` advice does not constitute a general mechanism for efficiently passing data in local variables between arbitrary advice that are woven into the same method body. For instance, it is not possible to pass data in local variables from one "`before() execution()`" advice to an "`after() call()`" advice.

Inter-advice communication [20] allows efficient data passing in local variables between advice bodies. This solves the two aforementioned problems, since it does not require the insertion of any wrapper methods and it enables data passing between arbitrary advice woven into the same method body. To make use of inter-advice communication, we mark public static fields in an aspect with the Java annotation `@InvocationLocal`. Within advice bodies woven on the same method, invocation-local variables are mapped to local variables.

3. CASE STUDY 1: LISTENER LATENCY PROFILING

In this section we show how an existing profiler that uses low-level bytecode instrumentation can be recast as a compact aspect. With this case study, we also illustrate how a common use of the `around()` advice can be replaced by a combination of `before()` and `after()` advice in conjunction with inter-advice communication.

Listener latency profiling (LLP) [11] helps developers locate slow operations in interactive applications, where the perceived performance is directly related to the response time of event listeners. LiLa[2] is an implementation of LLP based on ASM [15], a low-level bytecode engineering library.

The `LiLaAspect` in Figure 1 is an implementation of LLP providing the functionality of LiLa. In contrast to LiLa, the aspect is concisely implemented in a few lines of code. Figure 1 shows two versions of the `LiLaAspect`; one version relying on the `around()` advice, and a second version leveraging inter-advice communication.

To calculate the response time of events, the aspect in Figure 1(a) uses the `around()` advice to surround the execution of methods on instances (target objects) of any subtype of the `EventListener` interface, which is specified by the "`execution(* EventListener+.*(..))`" expression of the `allEvents()` pointcut. The start time is computed before the execution of the intercepted method, which is triggered by the call to `proceed(...)`. After (normal or abnormal) method completion, the actual execution time is computed. Whenever the execution time exceeds a given threshold (indicating high latency), the event is profiled. The method `profileEvent(...)`, which is not shown in the figure, logs an identifier of the intercepted method (conveyed by the static join point, which is accessed through AspectJ's pseudo-variable `thisJoinPointStaticPart`), the execution time, and the target object. This information helps developers locate the causes of potential performance problems due to slow event handling.

Unfortunately, as described in Section 2.3, the use of the `around()` advice compromises comprehensive aspect weaving. Therefore, in general, the aspect in Figure 1(a) cannot be used to capture events in classes of the Java class library. However, it is of paramount importance that LLP covers the Java class library, because it includes many implementations of the `EventListener` interface. Our inter-advice communication mechanism solves this problem.

Figure 1(b) shows the `LiLaAspect` using inter-advice communication to emulate the functionality of the `around()` advice with `before()` and `after()` advice. The invocation-local variable `start` is used to pass the time information from the `before()` advice to the `after()` advice. The `before()` advice is simpler because it does not need to access the target object. Thanks to inter-advice communication, the aspect can be comprehensively woven, thus providing the basic functionality of the original LiLa tool in a compact and extendible way.

[2] `http://www.inf.usi.ch/phd/jovic/MilanJovic/Lila/Welcome.html`

(a) LiLaAspect with around() advice:

```
public aspect LiLaAspect {
   // only calls lasting at least 100ms are profiled
   public static final long THRESHOLD_NS = 100L * 1000L * 1000L;

   pointcut allEvents() : execution(* EventListener+.*(..));

   Object around(EventListener l) : target(l) && allEvents() {
      long start = System.nanoTime();
      try {
         return proceed(l); // proceed with the execution
      } finally {
         long exectime = System.nanoTime() - start;
         if (exectime >= THRESHOLD_NS)
            profileEvent(thisJoinPointStaticPart, exectime, l);
      }
   }
   ...
}
```

(b) Equivalent aspect using inter-advice communication:

```
public aspect LiLaAspect {
   public static final long THRESHOLD_NS = 100L * 1000L * 1000L;

   pointcut allEvents() : execution(* EventListener+.*(..));

   @InvocationLocal
   public static long start;

   before() : allEvents()  {
      start = System.nanoTime();
   }

   after(EventListener l) : target(l) && allEvents() {
      long exectime = System.nanoTime() - start;
      if (exectime >= THRESHOLD_NS)
         profileEvent(thisJoinPointStaticPart, exectime, l);
   }
   ...
}
```

Figure 1: Simplified LiLaAspect implementing LLP

4. CASE STUDY 2: CALLING CONTEXT PROFILING

In this section we illustrate our approach to profiler development with aspects that generate complete CCTs covering all method execution in an application, including methods in the Java class library. We leverage MAJOR's ability of weaving with full method coverage and inter-advice communication for efficiently passing state between advice.

First, we discuss our data structure representing a CCT. Second, we present a simple aspect for CCT creation. Third, we show that the aspect can be easily extended to collect additional dynamic metrics. Fourth, we optimize CCT construction by parallelizing it with program execution.

4.1 CCT Representation

Before presenting an aspect that creates CCTs, we briefly discuss our data structure representing CCTs at runtime. In a multithreaded environment, there are two options: Each thread may create a separate, thread-confined CCT, and the per-thread CCTs may be integrated after termination of the corresponding threads. Alternatively, the CCT under construction may be shared between all threads, and access to

```
public aspect CCTProf {
   public static final CCTNode root = new CCTNode();

   public static final ThreadLocal<CCTNode> currentNode =
      new ThreadLocal<CCTNode>() {
         protected CCTNode initialValue() { return root; }
   };

   @InvocationLocal
   public static CCTNode caller, callee;

   pointcut execs() : execution(* *.*(..)) && !within(CCTProf);

   before() : execs() {
      caller = currentNode.get();
      callee = caller.profileCall(thisJoinPointStaticPart);
      currentNode.set(callee);
   }

   after() : execs() { currentNode.set(caller); }
   ...
}
```

Figure 2: Simplified profiling aspect that generates a CCT using inter-advice communication

the CCT has to be thread-safe. While the first approach enables efficient access to the thread-local CCTs, it can result in high memory consumption if several concurrent threads execute the same code. Therefore, we chose the second option, a single, shared, thread-safe CCT.

Thread-safety of a shared CCT can be achieved in different ways. Synchronizing every access to the CCT is too expensive. Hence, we rely on a non-blocking data structure using atomic Compare-And-Swap (CAS) instructions [3]. Each CCT node has a reference to a method identifier, a reference to callee nodes, as well as counters to store the dynamic metrics collected within the corresponding calling context, such as the number of method invocations.

Each node in the CCT is represented by an instance of type `CCTNode`, which stores the dynamic metrics collected for the corresponding calling context and offers a simple interface to update the profiling data in the form of methods `profile`M`(...)`, where M corresponds to a dynamic metric (`profileTime(long time)`, `profileAllocation(Object o)`, etc.). The method `profileCall(JoinPoint.StaticPart mid)` plays a special role; it returns the child CCT node representing a callee respectively creates such a node if it does not already exist. The argument `mid` (method identifier) is the static join point identifying the callee method.

4.2 Simple Profiling Aspect for CCT Creation

Figure 2 illustrates a simple aspect for generating a CCT (for simplicity, we do not consider constructors in this example). The aspect keeps the root of the shared CCT in a static field and defines the thread-local variable `currentNode` representing the current position in the CCT for each thread. The two advice in Figure 2 build the CCT. The `before()` advice is woven in method entries. It loads the caller's `CCTNode` instance from the thread-local variable `currentNode`, looks up the callee's `CCTNode` (`profileCall(...)`), and stores it into the thread-local variable. The static join point representing the woven method (i.e., the method identifier of the callee) is accessed through AspectJ's `thisJoinPointStaticPart` pseudo-variable.

The fields `caller` and `callee` are declared as `@InvocationLocal`. That is, thanks to our inter-advice communication mechanism, both the caller node and the callee node can be efficiently accessed from local variables in other advice that are woven into the same method body.

```
public aspect AllocCCTProf {
   ... // same code as in CCTProf (see Figure 2)

   pointcut allocs() : call(*.new(..)) && !within(AllocCCTProf);

   after() returning(Object o) : allocs() {
      callee.profileAllocation(o);
   }
}
```

Figure 3: Extended aspect to collect also object allocation metrics

The `after()` advice, woven before (normal and abnormal) method completion, restores the caller's `CCTNode`, which is referenced by the invocation-local variable `caller`, into the thread-local variable `currentNode`.

Note that without inter-advice communication, direct access to the caller's `CCTNode` instance would be impossible in the `after()` advice. However, the aspect programmer has two other options: (1) In the `after()` advice, the callee's `CCTNode` is loaded from the thread-local variable in order to to access the parent node (assuming that each `CCTNode` instance keeps a reference to the parent node in the tree). (2) Instead of the `before()` and `after()` advice, an `around()` advice is used. Both options have drawbacks. The first option causes higher overhead because of extra memory read accesses. The second option results in code transformations that introduce wrapper methods, which can cause problems when weaving the aspect in the Java class library (see Section 2.3).

In order to highlight the flexibility of our AOP-based approach to profiler development, Figure 3 illustrates an extension of the `CCTProf` aspect, called `AllocCCTProf`, in order to collect also object allocation metrics for each calling context.

The additional `after()` advice is woven after constructor invocations, and the newly created object is passed to the advice as argument. We assume that the method `profileAllocation(...)` updates object allocation metrics, such as the number of allocated instances and an estimate of the total allocated bytes in a calling context.

In Figure 3 the invocation-local variable `callee` is read in order to directly access the `CCTNode` instance corresponding to the current calling context. Hence, thanks to inter-advice communication, the thread-local variable `currentNode` need not be accessed in the given advice. Note that in the `AllocCCTProf` aspect, the invocation-local variable `callee` is used to efficiently pass data from a "`before(): execution()`" advice to an "`after(): call()`" advice. Prevailing AOP languages, such as AspectJ, do not offer any mechanism to this end.

4.3 Parallelizing CCT Creation

In order to reduce the overhead of calling context profiling on multicores, we parallelize application code and CCT construction [3]. Each thread maintains a calling context representation that does not require access to the shared CCT upon update, but preserves enough information that allows another thread to asynchronously perform the corresponding CCT updates. To this end, each thread produces "packets" of method calls and returns. When a packet is full, it is passed to the "CCT manager" through a queue. The

```
public class TC { // thread context
   // shadow stack
   public static final int STACK_SIZE = 10000;
   public final Object[] stack = new Object[STACK_SIZE];
   public int sp = 0; // next free entry on shadow stack

   // current packet
   public static final int PACKET_SIZE = 40000;
   public Object[] packet = new Object[PACKET_SIZE];
   public int nextFree = 1; // next free entry in packet
}

public aspect ParCCTProf {
   public static final ThreadLocal<TC> currentTC =
      new ThreadLocal<TC>() {
         protected TC initialValue() { return new TC(); }
   };

   @InvocationLocal
   public static TC tc;

   pointcut execs() : execution(* *.*(..))
                   && !within(ParCCTProf);

   before() : execs() {
      tc = currentTC.get();
      if (tc.nextFree >= TC.PACKET_SIZE) { // current packet full
         CCTManager.submitPacket(tc.packet);
         tc.packet = new Object[TC.PACKET_SIZE];
         for (int i = 0; i < tc.sp; ++i) // create packet header
            tc.packet[i] = tc.stack[i];
         tc.nextFree = tc.sp + 1;
      }
      tc.stack[tc.sp++] = thisJoinPointStaticPart;
      tc.packet[tc.nextFree++] = thisJoinPointStaticPart;
   }

   after() : execs() { tc.sp--; tc.nextFree++; }
   ...
}
```

Figure 4: Simplified aspect for parallelized CCT creation

CCT manager has a thread pool to process incoming packets and to update the CCT. In order to enable parallel and possibly out-of-order processing of packets, the packets must be independent of each other. Each packet corresponds to a partial CCT, and the integration of these partial CCTs must be a commutative and associative operation.

Figure 4 shows the aspect `ParCCTProf`, implementing parallelized CCT creation. The thread-local variable `currentTC` of type `TC` (thread context) keeps calling context information for each thread. The state of a `TC` instance includes a shadow stack (`stack`), a "stack pointer" (`sp`), which indicates the next free element on the shadow stack, a packet of method calls and returns (`packet`), and the index of the next free entry in the packet (`nextFree`). The invocation-local variable `tc` helps reduce (relatively expensive) access to the thread-local variable `currentTC`, which is read only once in each woven method.

Each packet consists of a header and a sequence of method calls and returns. The header corresponds to the calling context of the first method call in the packet. It can be regarded as routing information that describes a path in the shared CCT, starting from the root. The header ends with a `null` value. In the following sequence, method calls are represented by the corresponding method identifiers (i.e., `JoinPoint.StaticPart` instances), whereas returns are simply indicated by `null` values. Since returns at the end of a packet have no effect on the partial CCT corresponding to the packet, we allow `nextFree` to exceed the packet size.

Figure 4 specifies how the packet is updated upon method entry and completion. On method entry, the method identifier of the callee method has to be appended to the packet. Hence, if the current packet is full, it is submitted to the CCT manager (`CCTManager.submitPacket(...)`). Afterwards, a new packet is created, where the header is a copy of the current shadow stack. On method completion, `nextFree` is simply incremented, creating `null` entries in the packet.

When a thread terminates, its last packet is usually incomplete and has not been submitted to the CCT manager. We address this issue by registering each thread upon creation in the CCT manager. The CCT manager is responsible for detecting thread termination (e.g., by periodically polling the thread state), to collect the last packet from the `TC` instance of a terminated thread, and to put it into the queue.

Each thread in the CCT manager's thread pool repeatedly takes a packet from the queue and integrates the corresponding partial CCT into the shared CCT. The thread first creates a thread-local, partial CCT representing the method calls in the packet. Afterwards, the partial CCT is integrated into the shared CCT using a recursive method. This approach helps reduce the number of accesses to nodes in the shared CCT, because packets frequently include repetitive call patterns (e.g., methods invoked in a loop). Instead of accessing the shared CCT upon each method call, we need to access it only once for a series of accumulated method calls. Note that the packet format can be extended to support also other events than method call and return. For example, in order to profile object allocation, the packet could store a class reference for each allocated object (in addition to the method identifiers upon method invocation).

5. CONCLUSION

Bytecode instrumentation techniques are very useful in the development of profilers, but mastering them requires deep knowledge of the virtual machine and of low-level bytecode engineering libraries. The resulting profilers are often complex, require considerable development effort, and are hard to extend. As an alternative to bytecode instrumentation, we promote AOP for the rapid development of extensible profilers. With the aid of AOP, many important profilers can be concisely specified in a few lines of code.

However, current AOP frameworks, such as AspectJ, lack some features that are essential for building useful and efficient aspect-based profilers. For instance, aspect-weaving in the standard Java class library is seriously restricted or even impossible. Furthermore, it is not possible to efficiently pass data in local variables from one join point to another within the same woven method. To overcome these limitations, we leverage MAJOR, an aspect weaver that ensures comprehensive weaving in all classes loaded in a JVM. Furthermore, we use inter-advice communication, a novel mechanism for passing data in local variables between woven advice bodies. Inter-advice communication is complementary to the features offered by current AOP languages, and we have integrated it into MAJOR.

To illustrate our approach to profiler development, we presented two case studies, an aspect for listener latency profiling, as well as an extensible and efficient aspect for calling context profiling.

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

6. REFERENCES

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 85–96. ACM Press, 1997.

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

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

[4] W. Binder and J. Hulaas. Flexible and efficient measurement of dynamic bytecode metrics. In *Fifth International Conference on Generative Programming and Component Engineering (GPCE-2006)*, pages 171–180, Portland, Oregon, USA, Oct. 2006. ACM.

[5] W. Binder, J. Hulaas, and P. Moret. Advanced Java Bytecode Instrumentation. In *PPPJ'07: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, pages 135–144, New York, NY, USA, 2007. ACM Press.

[6] W. Binder, J. Hulaas, P. Moret, and A. Villazón. Platform-independent profiling in a virtual execution environment. *Software: Practice and Experience*, 39(1):47–79, 2009. `http://dx.doi.org/10.1002/spe.890`.

[7] W. Binder, M. Schoeberl, P. Moret, and A. Villazón. Cross-profiling for embedded Java processors. In *Fifth International Conference on the Quantitative Evaluation of SysTems (QEST-2008)*, pages 287–296, Saint-Malo, France, Sept. 2008. IEEE Computer Society.

[8] M. Dmitriev. Profiling Java applications using code hotswapping and dynamic call graph revelation. In *WOSP '04: Proceedings of the Fourth International Workshop on Software and Performance*, pages 139–150. ACM Press, 2004.

[9] B. Dufour, L. Hendren, and C. Verbrugge. *J: A tool for dynamic analysis of Java programs. In *OOPSLA '03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 306–307, New York, NY, USA, 2003. ACM Press.

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

[11] M. Jovic and M. Hauswirth. Measuring the performance of interactive applications with listener latency profiling. In *PPPJ '08: Proceedings of the 6th international symposium on Principles and practice of programming in Java*, pages 137–146, New York, NY, USA, 2008. ACM.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP-2001)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, 2001.

[13] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[14] P. Moret, W. Binder, and A. Villazón. CCCP: Complete calling context profiling in virtual execution environments. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 151–160, Savannah, GA, USA, 2009. ACM.

[15] ObjectWeb. ASM. Web pages at `http://asm.objectweb.org/`.

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

[17] J. M. Spivey. Fast, accurate call graph profiling. *Softw. Pract. Exper.*, 34(3):249–264, 2004.

[18] Sun Microsystems, Inc. JVM Tool Interface (JVMTI) version 1.1. Web pages at `http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html`, 2006.

[19] The Apache Jakarta Project. The Byte Code Engineering Library (BCEL). Web pages at `http://jakarta.apache.org/bcel/`.

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

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

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

[23] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 263–271, New York, NY, USA, 2006. ACM.