Rapid Development of Flexible and Efficient Dynamic Program Analysis Tools

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

ABSTRACT

In this work, we show that it is possible to raise the abstraction level of dynamic program analysis frameworks, providing high-level constructs to improve flexibility, efficiency, and usability of dynamic analysis tools. We introduce a new framework based on aspect-oriented programming that enables rapid development of efficient dynamic program analysis tools. Moreover, we present high-level constructs for the parallelization of specific code regions and for runtime adaptation of the scope of the analysis. The viability and benefits of our approach are confirmed with several dynamic analysis tools implemented with our framework.

1. PROBLEM AND MOTIVATION

Prevailing programming languages, such as Java, C#, Scala, as well as some implementations of Python and Ruby, can be compiled to an intermediate code representation (i.e., bytecode) and executed on a virtual machine. This approach allows code portability across different platforms, providing a standard abstraction of the underlying system. Moreover, state-of-the-art virtual machines offer automatic memory management, just-in-time compilation, and automatic code optimizations. All these features move the task of producing optimized code from the programmer to the virtual machine, reducing development time to achieve good performance.

The presence of a virtual machine layer in the software stack makes it difficult to understand the performance of complex applications executing on modern hardware platforms. For example, automatic memory management and runtime optimizations performed by the virtual machine may have a significant impact on overall performance. Since these virtual machine activities depend on the runtime behavior of an application, static program analysis may not always locate potential hotspots. In addition, some features of the programming language, such as polymorphism and reflection, may limit the scope of static analysis tools and potentially lead to biased results.

To overcome these limitations, dynamic program analysis tools provide insight into an application's runtime behavior by measuring relevant system activities while the program is executing. Such tools are usually based on bytecode instrumentation techniques to insert monitoring code into the bytecode of the observed application.

While there are dynamic analysis tools for common tasks, such as data race detection, memory leak detection, and calling-context profiling, there is a lack of tools that allow users to specify custom analyses. Moreover, since prevailing analysis tools lack the flexibility to profile only a specific subset of the observed application, resource utilization and performance are usually measured after the implementation phase, when the program is stable enough to be deployed and executed. However, detecting and solving runtime issues so late in the development process often incurs high costs, particularly if it turns out that some taken design choices prevent good performance and the design has to be revised.

We believe there is a need for a novel dynamic program analysis framework that allows programmers to rapidly specify custom analyses for measuring runtime performance of parts of a program, during the implementation phase. To this end, we identify the following shortcomings in prevailing dynamic analysis tools:

- *Limited flexibility*: bytecode instrumentation is usually performed using low-level techniques that require expert knowledge at the level of the virtual machine. While this approach allows developers to specify any possible instrumentation, tool development is tedious and error-prone, and the resulting tools are often hardly extensible and customizable [2, 24, 46].

- *Limited efficiency*: the inserted bytecode usually collects dynamic information about the observed program and performs some analysis. In many cases, this computation could be done in parallel with the execution of the observed application. However, only few dynamic program analysis tools take advantage of this opportunity to efficiently exploit under-utilized CPU cores and reduce analysis overhead [3, 4].

- *Performance interference*: due to the aforementioned issues, the user is often forced to perform a set of pervasive and computationally expensive analyses that may create some bias in the observed metrics.

The goal of our research is to overcome the aforementioned limitations by raising the abstraction level of current dynamic program analysis frameworks, providing high-level constructs to improve flexibility, efficiency, and usability of dynamic analyses.

2. BACKGROUND

Aspect-oriented programming (AOP) [23] enables the specification of cross-cutting concerns in applications, avoiding related code that is scattered throughout methods,

*Adviser: Prof. Walter Binder, University of Lugano, Switzerland

classes, or components. Aspects specify *pointcuts* to intercept certain points in the execution of programs (so-called *join points*), such as method calls, field accesses, etc. *Advice* are executed *before*, *after*, or *around* the intercepted join points, and have access to contextual information of the join points. During the *weaving process*, *aspect weavers* modify the code of the base application to execute advice at intercepted join points.

Traditionally, AOP has been used for disposing of "design smells" such as needless repetition, and for improving maintainability of applications. AOP has also been successfully applied to the development of software engineering tools, such as profilers, debuggers, or testing tools [7,28,40], which in many cases can be specified as aspects in a concise manner. Hence, in a sense, AOP can be regarded as a versatile approach for specifying certain program transformations at a high level, hiding low-level implementation details, such as bytecode manipulation, from the programmer.

Dynamic AOP allows aspects to be changed and code to be rewoven in a running system, thus enabling runtime adaptation of applications. Dynamic AOP has been used for adding persistence or caching at runtime [26], or for debugging and fixing bugs in a running server [15]. However, prevailing aspect languages have not been specifically developed for implementing dynamic analyses, thus lacking some features to express essential instrumentations for certain dynamic analyses.

3. RELATED WORK

Since our framework relies on Java bytecode instrumentation, Section 3.1 describes prevailing approaches for Java bytecode manipulation. Section 3.2 reviews common profiling tools for Java. Section 3.3 offers a description of state-of-the-art frameworks for dynamic AOP. Finally, Section 3.4 discusses how recent dynamic analysis tools take advantage of multicore machines.

3.1 Java Bytecode Instrumentation

Java bytecode instrumentation is usually performed with low-level tools that require in-depth knowledge of the bytecode format.

BCEL [1] and ASM [2] allow programmers to analyze, create, and manipulate Java class files by means of a low-level API. Java classes, constant pools, methods, and bytecode instructions are represented as objects containing all information related to the corresponding construct.

Soot [35] is a bytecode optimization framework that includes a library for bytecode analysis and transformation. Multiple bytecode representations are used to simplify the manipulation and the optimization of Java bytecode.

Shrike [3] is a bytecode instrumentation library that is part of the T. J. Watson Libraries for Analysis (WALA) and provides interesting features to increase efficiency. For example, parsing is limited to the parts of the class to be modified, bytecode instructions are represented by immutable objects, and many constant instructions can be represented with a single object shared between methods.

Javassist [14] is a library for load-time structural reflection and for defining new classes at runtime. The API allows two different levels of abstraction: source-level and bytecode-level. Compared to the aforementioned approaches, the source-level abstraction of Javassist is particularly interesting as it does not require any knowledge of the Java bytecode format and it allows insertion of code fragments specified with source text.

While these low-level approaches are very flexible because the possible transformations are not restricted, tool development is tedious and error-prone. As another drawback, the resulting analysis tools are often complex and difficult to maintain and to extend [2].

3.2 Java Profiling Tools

Prevailing Java profilers, such as JProf [4], and JFluid [16] are good examples of current state-of-the-art profilers. They are optimized for common tasks, such as memory-leak detection, CPU utilization monitoring, and heap walking. However, prevailing profilers fall short when it comes to provide a way to specify custom instrumentations to perform more specific analysis. The same limitation applies to many dynamic analysis tools, such as *J [17], jPredictor [12], and THOR [34].

BTrace [5] provides an annotation-based language to let the user specify which program attributes to profile. However, this language is limited and does not support the specification of more advanced instrumentations. For example, a BTrace profiler cannot allocate new objects, specify synchronized blocks, or have loops.

AOP-based tools, such as DJProf [28], allow the developer to define custom instrumentation by writing custom aspects. However, mainstream AOP languages lack high-level constructs to develop efficient dynamic analysis tools that can take advantage of modern multicore architectures.

3.3 Dynamic Aspect-Oriented Programming

While dynamic AOP is often supported in dynamic languages [8, 21] such as Lisp or Smalltalk, where code can be easily manipulated at runtime, it is more difficult and challenging to offer dynamic AOP for languages like Java.

PROSE [27, 29, 30] is an adaptive middleware platform offering dynamic AOP. PROSE has evolved in three versions, which make use of the three different approaches to dynamic AOP. The latest version of PROSE supports two weaving strategies: advice weaving, based on method replacement, and stub weaving, for join points involving external advice. This version has been implemented in two JVMs, in the Jikes RVM and in Oracle's HotSpot VM.

Steamloom [11] provides support for AOP at the JVM level, based on the Jikes RVM. Steamloom extends the Java bytecode language with two additional instructions specific to AOP: *beginadvice* and *endadvice*. At weaving time, the woven code is automatically surrounded by these two instructions to simplify the removal of the aspect.

Wool [15] combines two different weaving strategies. The Java Platform Debugger Architecture (JPDA) is used to insert breakpoints at all possible join points. Once an active join point has been intercepted a sufficient number of times, the hotswap mechanism is used to replace the class definition with its woven version to reduce runtime overhead.

[1] See `http://jakarta.apache.org/bcel/`
[2] See `http://asm.ow2.org/`
[3] See `http://wala.sourceforge.net/wiki/index.php/Shrike_technical_overview`
[4] See `http://perfinsp.sourceforge.net/jprof.html`
[5] See `http://kenai.com/projects/btrace/`

Runtime weaving in JAsCo [33, 36] relies on the hotswap mechanism to update class definitions at runtime. If JPDA is used, JAsCo requires preliminary load-time insertion of hooks for aspect weaving.

Similar to JAsCo, JBossAOP [6] relies on a modified class loader to prepare join point shadows and to add the corresponding auxiliary fields at load-time. Runtime weaving is supported by means of the hotswap mechanism of the *java.lang.instrument* API.

AspectWerkz [37] allows users to deploy and undeploy aspects at runtime using hotswapping. Similar to JBossAOP, AspectWerkz forces a preparation step at load-time using a customized class loader to add auxiliary fields to support different pointcuts.

Wool, JAsCo, JBossAOP, and AspectWerkz rely on Javassist [14] for bytecode manipulation.

3.4 Dynamic Program Analysis on Multicores

With the increasing popularity of multicore architectures, recent dynamic analysis tools have focused on off-loading the analysis tasks to worker threads so as parallelize base program execution and analysis to improve performance.

Approaches based on parallelized slice profiling, such as Shadow Profiling [25] and SuperPin [41], 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.

Pipelined Profiling and Analysis (PiPA) [45] 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. However, PiPA is based on a dynamic binary instrumentation system and does not provide a high-level API to ease the adoption for users.

Cache-friendly Asymmetric Buffering (CAB) [19] is a dynamic analysis framework based on lock-free ring buffers to communicate analysis data from the base program to analysis threads. CAB efficiently exploits shared caches of multicore systems but requires modifications at the JVM level.

Also the work presented in [20] decouples the execution of profiling tasks from the base program. However, communication overhead is reduced by restricting the amount of data to be communicated to the profiler, which can compute some information based on the already received data. This approach requires compiler support to determine the set of data items to be computed by the profiler.

4. APPROACH AND UNIQUENESS

We propose to add a new dimension to unit testing, allowing programmers to check for both correctness and runtime performance requirements. Developers should be able to define *analysis units*, isolating small subsets of programs to evaluate whether performance targets are met. To achieve this goal, it is necessary to provide high-level constructs to precisely define the scope of the analysis and when it must be deployed. The set of possible instrumentations should include traditional analyses, such as object allocation rate, memory leak detection, and calling-context profiling, as well as custom analyses, such as e.g. counting the number of accesses to a data structure, the dynamic type of the arguments of a method, and the number of executed bytecodes in a code section. Analysis units can be deployed on a testbed of machines with heterogeneous hardware and software characteristics. This allows developers to quickly estimate the impact of design choices and implementation alternatives on key performance metrics during the implementation phase, when the effort for code modifications is small.

To achieve our goal of improving flexibility, efficiency, and usability of dynamic program analysis tools, our research contributes to the state-of-the-art in the following domains:

- *Dynamic AOP*: realization of a high-level, dynamic AOP framework explicitly designed for rapid development of dynamic program analysis tools. Compared to existing frameworks, our solution improves runtime performance and extends the scope of the analysis to all executed bytecodes [39, 42].
- *Automatic and efficient parallelization of analysis code*: specification of a high-level construct to automatically forward the execution of specific methods to a pool of worker threads. A novel and efficient buffering strategy is applied to limit the overhead of inter-thread communication [3, 4].
- *Automatic scope refinement*: specification of a high-level API to dynamically change the scope of a dynamic analysis, limiting the observation to the most important sites for a specific metric. This novel approach is based on periodic evaluation of collected results, and on dynamic modifications to the scope of the analysis [1].
- *Data passing between instrumentation sites*: specification of a high-level construct to efficiently pass data between different instrumentation sites [5, 9, 39].

We evaluate our approach on popular dynamic analysis tools for the Java Virtual Machine (JVM), using the AspectJ [22] aspect language and weaver. However, the principles and techniques explored in our work can be applied to other virtual machines, such as the Common Language Runtime (CLR) of Microsoft's .NET Framework and the Parrot VM, and other aspect languages and weavers.

5. RESULTS AND CONTRIBUTIONS

Below we discuss the research results achieved so far. A brief description is given for each topic, and we refer to accepted publications for more detailed descriptions. The developed software has been used to implemented novel analyses presented at various conferences [6, 13, 31, 32].

5.1 HotWave

HotWave [38, 39] (standing for HOTswap & reWeAVE) is a framework for dynamic AOP in Java. HotWave is unique in reconciling three important features: dynamic AOP, compatibility with standard JVMs, and comprehensive aspect weaving with complete method coverage. HotWave is based on AspectJ and supports a wide range of standard AspectJ

[6]See `http://labs.jboss.com/jbossaop/`

constructs for dynamic cross-cutting. HotWave is portable, implemented in pure Java, and is compatible with standard, state-of-the-art JVMs. In contrast to other AOP frameworks for Java, HotWave enables aspect (re)weaving with complete method coverage, that is, aspects can be woven into any method that has a bytecode representation, including methods in dynamically generated classes or in the standard Java class library. These distinguishing features make HotWave an ideal framework for the rapid development of sophisticated software engineering tools.

5.2 SafeWeave

SafeWeave [42] is an alternative framework for dynamic AOP in Java, with dedicated support for the Dynamic Code Evolution VM (DCE VM) [43, 44]. DCE VM is a modified version of the Java HotSpot VM that supports modification of methods that are active on the call-stack of threads, as well as insertion and removal of fields and methods. To take advantage of the unique features of DCE VM, at weave-time SafeWeave inserts bytecode attributes to define safe update regions, that is, bytecodes that are present in both the original and the extended class definition. Within these regions, changes are atomic and correctness is guaranteed even though weaving happens in parallel to program execution and the system fully supports dynamic class loading. SafeWeave exposes atomic code updates through a high-level programming model based on AOP.

5.3 Buffered Advice

For many aspect-based software-engineering tools, synchronous advice execution is not a requirement. Often, the computation performed in the advice can be handled asynchronously, possibly in parallel with program execution using idle CPU cores. However, most aspect-based tools execute advice bodies upon intercepted join points in a synchronous manner. That is, each program thread synchronously invokes advice while executing woven code.

Asynchronous advice execution at the granularity of individual advice invocations rarely pays off, because the communication overhead due to passing the relevant context information from one thread to another may outweigh the benefits thanks to parallelization of program and advice code. Communication between threads involves access to a shared data structure (e.g., a shared queue for passing the context information of invoked advice) and thread-safety incurs some overhead [18].

Buffered Advice [4] mitigate the communication overhead incurred by asynchronous advice execution. Advice invocations are aggregated in thread-local storage until it pays off to execute the aggregated workload asynchronously. Upon invocation by woven code, buffered advice only store the relevant context information (e.g., references to static or dynamic join point instances, receiver or owner object, etc.) in a thread-local buffer. When the buffer is full, the workload conveyed in the buffer is processed, that is, the respective advice bodies are executed using the buffered parameters.

Buffered advice offers two important advantages: First, it allows for different buffer processing strategies in a flexible way. For example, a full buffer may be synchronously processed by the thread that has filled the buffer, or the full buffer may be passed to another thread in a pool of dedicated processing threads. In the latter case, buffered advice can be processed in parallel with program code on idle CPU cores. Second, execution of application code is less disrupted by buffered advice invocation, reducing the negative impact of advice invocation on locality in the woven code. Conversely, as the bodies of buffered advice are executed altogether when a buffer is processed, locality in the aspect code may improve.

5.4 Deferred Methods

Deferred Methods [3] extend the concept of Buffered Advice to general Java programs (i.e., not necessarily based on AOP) and add a set of novel features to reduce development time and improve runtime performance of resulting programs. Compared to existing approaches, deferred methods support custom buffer processing strategies, ease preprocessing of buffers to reduce contention on shared data structures, and offer a synchronization mechanism to wait for the completion of previously invoked deferred methods. We also introduce a novel adaptive buffer processing strategy that parallelizes the execution only when the observed workload leaves some CPU cores under-utilized. Using a profiler as case study, deferred methods with the adaptive buffer processing strategy yield an average speedup of more than factor 4 on a quad-core machine [3]. Such speedup stems both from parallelization and from reduced contention.

5.5 Self-Refining Aspects

A severe limitation of many prevailing aspect-based dynamic analysis tools is that the scope of the analysis is statically defined in the aspects. The analyzed application is usually treated as a black-box, and all potentially interesting join points are intercepted. This approach may produce biased results, as the overhead introduced by complete instrumentation could have an impact on collected metrics.

Self-refining dynamic analysis aspects [1] are able to reflect on their performance at runtime, and to optimize their scope to reduce measurement overhead. Compared to traditional analysis aspects, self-refining aspects periodically reweave application classes, excluding from the scope those methods that have been found irrelevant for the analysis. Reweaving may be automatically triggered by the aspect itself after a given amount of time, or after a pre-defined number of refining requests. It is up to the user to choose which strategy to adopt. Frequent refinement operations have to be avoided to prevent excessive overhead due to reweaving. While self-refining aspects mainly focus on reducing the number of intercepted join points, it is also possible to extend the scope of the aspect, eventually restoring the interception of previously discarded join points.

5.6 Inter-Advice Communication

In AspectJ, *around* advice (in conjunction with a *proceed* statement) allow storing data in local variables before a join point, and accessing that data after the join point. Hence, one common use of *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, prevailing AOP frameworks do not provide any general support for efficiently passing data in local variables between arbitrary advice that are woven into the same method body. For instance, AspectJ does not support data passing in local variables from a "*before call*" to an "*after execution*" advice.

Inter-Advice Communication [5, 9, 39] is a novel mechanism for efficiently passing data between advice that are

woven into the same method. With the aid of annotations, the aspect programmer can declare Synthetic Local Variables [10] which have the scope of a woven method body. To implement inter-advice communication, our framework selectively inlines advice that access Synthetic Local Variables. Inter-Advice Communication is complementary to AspectJ constructs and enables important optimizations in aspect-based profiling tools.

6. CONCLUSION

This work raises the abstraction level of current frameworks for dynamic program analysis, hiding low-level implementation details and allowing programmers to rapidly specify flexible and efficient analysis tools. Such tools can instrument specific parts of a program to inspect its runtime behavior and evaluate whether custom performance targets are met. HotWave enables dynamic deployment/undeployment of instrumentations with complete method coverage. SafeWeave is an extension of HotWave that, when executing with DCE VM, guarantees atomic updates even though weaving happens in parallel to program execution. Both HotWave and SafeWeave support novel constructs to improve runtime performance of dynamic program analysis tools, such as Buffered Advice, Deferred Methods, Self-Refining Aspects, and Inter-Advice Communication. The viability and benefits of our approach are confirmed by novel analyses presented at various conferences.

7. REFERENCES

[1] D. Ansaloni. Self-refining aspects for dynamic program analysis. In *Proceedings of the tenth international conference on Aspect-oriented software development companion*, AOSD '11, pages 75–76. ACM Press, 2011.

[2] D. Ansaloni, W. Binder, C. Bockisch, E. Bodden, K. Hatun, L. Marek, Z. Qi, A. Sarimbekov, A. Sewe, P. Tuma, and Y. Zheng. Challenges for refinement and composition of instrumentations: Position paper. In *International Conference on Software Composition 2012*, pages 86–96, 2012.

[3] D. Ansaloni, W. Binder, A. Heydarnoori, and L. Y. Chen. Deferred methods: Accelerating dynamic program analysis on multicores. In *CGO 2012: Proceedings of the International Symposium on Code Generation and Optimization (10 pages, to appear)*, 2012.

[4] 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*, pages 1–12, Rennes, France, March 2010. ACM Press.

[5] D. Ansaloni, W. Binder, A. Villazón, and P. Moret. Rapid development of extensible profilers for the Java virtual machine with aspect-oriented programming. In *WOSP/SIPEW 2010: Proceedings of the 1st Joint International Conference on Performance Engineering*, pages 57–62. ACM Press, Jan 2010.

[6] D. Ansaloni, L. Y. Chen, E. Smirni, and W. Binder. Model-driven Consolidation of Java Workloads on Multicores. In *Proceedings of DSN-PDS (12 pages, to appear)*, 2012.

[7] L. D. Benavides, R. Douence, and M. Südholt. Debugging and testing middleware with aspect-based control-flow and causal patterns. In *Middleware '08: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 183–202, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[8] A. Bergel. FacetS: First class entities for an open dynamic AOP language. In *Proceedings of the Open and Dynamic Aspect Languages Workshop*, Mar. 2006.

[9] 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.

[10] W. Binder, A. Villazón, D. Ansaloni, and P. Moret. @J - Towards rapid development of dynamic analysis tools for the Java Virtual Machine. In *VMIL '09: Proceedings of the 3th Workshop on Virtual Machines and Intermediate Languages*, pages 1–9. ACM, 2009.

[11] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 83–92. ACM, 2004.

[12] F. Chen, T. F. Serbanuta, and G. Rosu. jPredictor: A Predictive Runtime Analysis Tool for Java. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 221–230, New York, NY, USA, 2008. ACM.

[13] L. Y. Chen, D. Ansaloni, E. Smirni, A. Yokokawa, and W. Binder. Achieving Application-Centric Performance Targets via Consolidation on Multicores: Myth or Reality? In *Proceedings of the 21th International Symposium on High Performance Distributed Computing (12 pages, to appear)*, HPDC, 2012.

[14] S. Chiba. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer Verlag, Cannes, France, June 2000.

[15] S. Chiba, Y. Sato, and M. Tatsubori. Using Hotswap for Implementing Dynamic AOP Systems. In *1st Workshop on Advancing the State-of-the-Art in Run-time Inspection*, 2003.

[16] 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.

[17] 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.

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

[19] 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*, pages 155–174, Orlando, Florida, USA, October 2009. ACM Press.

[20] G. He and A. Zhai. Improving the performance of program monitors with compiler support in multi-core environment. In *IPDPS '10: IEEE International Parallel and Distributed Processing Symposium*, pages 1–12, Washington, DC, USA, April 2010. IEEE Computer Society.

[21] R. Hirschfeld. AspectS - Aspect-Oriented Programming with Squeak. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures,*

Services, and Applications for a Networked World, pages 216–232, London, UK, 2003. Springer-Verlag.

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353. Springer-Verlag, 2001.

[23] 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, *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242, Jyväkylä, Finnland, June 1997. Springer-Verlag.

[24] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. DiSL: a domain-specific language for bytecode instrumentation. In *AOSD '12: Proceedings of the 11th International Conference on Aspect-Oriented Software Development*, pages 239–250, 2012.

[25] 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*, pages 198–208, San Jose, CA, USA, March 2007. IEEE Computer Society.

[26] A. Nicoara and G. Alonso. Making applications persistent at run-time. *International Conference on Data Engineering*, 15:1368–1372, 2007.

[27] A. Nicoara, G. Alonso, and T. Roscoe. Controlled, Systematic, and Efficient Code Replacement for Running Java Programs. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 233–246, New York, NY, USA, 2008. ACM.

[28] 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.

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

[30] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, New York, NY, USA, 2002. ACM Press.

[31] 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.

[32] D. Röthlisberger, M. Härry, A. Villazón, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret. Augmenting static source views in IDEs with dynamic metrics. In *ICSM '09: Proceedings of the 25th IEEE International Conference on Software Maintenance*, pages 253–262, Edmonton, Alberta, Canada, 2009. IEEE Computer Society.

[33] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM.

[34] Q. M. Teng, H. C. Wang, Z. Xiao, P. F. Sweeney, and E. Duesterwald. Thor: a performance analysis tool for java applications running on multicore systems. *IBM J. Res. Dev.*, 54:456–472, September 2010.

[35] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.

[36] W. Vanderperren, D. Suvée, B. Verheecke, M. A. Cibrán, and V. Jonckers. Adaptive Programming in JAsCo. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 75–86, New York, NY, USA, 2005. ACM.

[37] A. Vasseur. Dynamic AOP and Runtime Weaving for Java – How does AspectWerkz address it? In *Dynamic Aspects Workshop (DAW04)*, Lancaster, England, Mar. 2004.

[38] A. Villazón, D. Ansaloni, W. Binder, and P. Moret. HotWave: Creating Adaptive Tools with Dynamic Aspect-Oriented Programming in Java. In *Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*, GPCE '09, pages 95–98. ACM, Oct. 2009.

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

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

[41] 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*, pages 209–217, San Jose, CA, USA, March 2007. IEEE Computer Society.

[42] T. Würthinger, D. Ansaloni, W. Binder, C. Wimmer, and H. Mössenböck. Safe and atomic run-time code evolution for java and its application to dynamic aop. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 825–844, New York, NY, USA, 2011. ACM.

[43] T. Würthinger, W. Binder, D. Ansaloni, P. Moret, and H. Mössenböck. Applications of enhanced dynamic code evolution for java in gui development and dynamic aspect-oriented programming. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 123–126, New York, NY, USA, 2010. ACM.

[44] T. Würthinger, C. Wimmer, and L. Stadler. Dynamic code evolution for java. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 10–19. ACM, 2010.

[45] 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*, pages 185–194, Boston, MA, USA, April 2008. ACM Press.

[46] Y. Zheng, D. Ansaloni, L. Marek, A. Sewe, W. Binder, A. Villazón, P. Tuma, Z. Qi, and M. Mezini. Turbo DiSL: partial evaluation for high-level bytecode instrumentation. In *TOOLS 2012 – Objects, Models, Components, Patterns*, pages 354–369, 2012.