USING ASPECT REWEAVING TECHNIQUES FOR APPLICATION PERFORMANCE MONITORING

Dušan Okanović, Milan Vidaković
University of Novi Sad, Faculty of Technical Sciences, Novi Sad, Serbia
{oki, minja}@uns.ac.rs

Abstract – *AspectJ and several other AOP frameworks allow for aspects to be woven with classes at compile time or at load time. While load-time weaving offers some flexibility, once weaved classes stay that way until application restarts with new aspects to be loaded. Adaptive continuous monitoring built on these frameworks requires periodical restarting of application in order for the new aspect configuration to be loaded. This approach is often very uncomfortable, sometimes even unacceptable. There are applications and systems that are not allowed to be restarted, or are allowed to be restarted very rarely. For these systems, the use of another AOP platform that allows runtime weaving and re-weaving of aspects would be of great importance.*

Keywords: Dynamic AOP, adaptive continuous monitoring

1. INTRODUCTION

Continuous monitoring provides a picture of dynamic software behavior under production workload. The data obtained from monitoring can for instance be used as a basis for architecture-based software optimization, visualization, and reconstruction [1]. Based on the data collected using continuous monitoring, it is possible to predict the behavior of the application and make a plan of further actions. This is, for example, important in capacity planning and maintenance scheduling.

Performance overhead, that a monitoring system imposes, is a very important issue. The monitoring system has to work using a minimal amount of resources in order not to interfere with performance of the monitored system. Profilers and debuggers induce significant performance overhead and are therefore unsuitable for monitoring during the operational phase. In order to achieve a reduction of monitoring overhead, it would be beneficial to automatically adapt monitoring to only monitor selected parts of the system, the ones that are suspected of causing performance issues and problems.

The DProf system proposed in [2] has been developed for adaptive monitoring of distributed enterprise applications with a low overhead. In order to do that, the Kieker [1] framework, which yields low overhead, is used for collecting the monitoring data.

Deployment diagram of the DProf system is shown in Fig. 1.

Additional components support the change of monitoring parameters at application runtime. These additional components have been developed using Java Management Extensions (JMX) [3]. The system analyzes call trees reconstructed from the gathered data and automatically creates a new monitoring configuration if needed.

A call tree represents calling relationships between software methods [4]. It contains the control-flow of method executions invoked by a client request. The first method is called the "root".

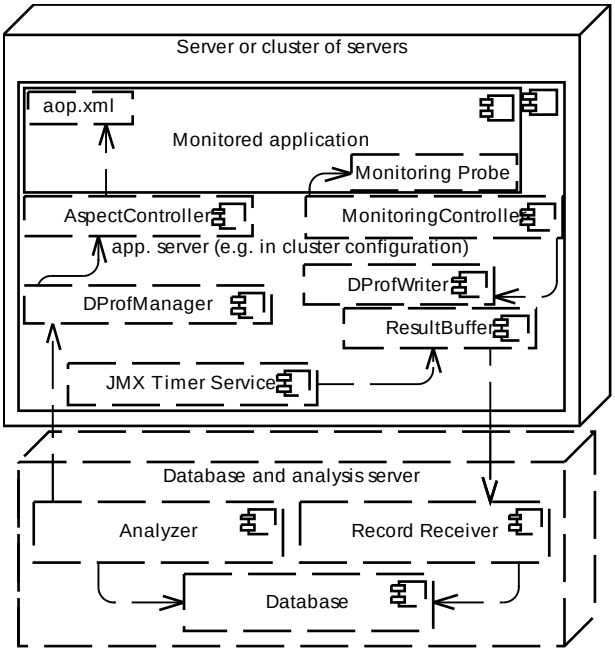

Fig. 1. Deployment diagram of the DProf monitoring system

DProf configuration parameters specify which of the application's call trees are going to be monitored and can furthermore specify nesting levels within the call tree that will be monitored. DProf stores data in a central database, regardless of on how many computers the monitored application is being executed. Using a mechanism integrated into the Kieker framework, during data gathering, each method execution within a trace is uniquely identified and is assigned a number which represents the order of execution. This allows call trees to be spread on different computers.

DProf reduces monitoring overhead by only monitoring parts of the software suspected of containing problems or deviating from expected behavior. The system starts by monitoring methods that are at the root of the call trees. If a deviation from expected results is detected in one of the trees, the DProf incrementally turns on monitoring in lower levels of that particular tree until the method that is causing the problem is determined. DProf adapts without human intervention to find the cause of the problem. This simulates the manual procedure typically employed for localizing the root cause of performance problems. Other systems perform monitoring of the whole software, regardless of the fact that other parts (i.e. other call trees) are working fine. Since DProf's additional monitoring components are implemented using JMX technology, the reconfiguration of the DProf monitoring parameters can

still be performed manually by system administrators using any JMX console.

DProf monitoring probes are usually inserted into application code using aspect-oriented programming (AOP) [5].

AOP removes clutter created when additional concerns, such as monitoring instrumentation, are implemented in the same module as the, so called, business logic concern. Using aspects that intercept the execution of program logic at defined points (so-called join points), developers can add additional behavior (defined in advices).

However, currently available AOP frameworks do not allow for runtime changing of aspects. Whenever DProf has to change monitoring configuration, i.e. to change join-points or behavior defined in advices, the monitored application has to be restarted.

This behavior of monitoring system can be problematic, even unacceptable. There are applications that cannot be simply stopped and restarted, as they have high availability demands. Some examples of these applications are stock market and banking systems and life support systems.

In this paper we explore the possibility of using other AOP frameworks that allow us to change aspects at runtime, i.e. to weave new and remove already woven aspects.

The remainder of this paper is structured as follows. Section 2 presents the concept of dynamic AOP and frameworks that support it (Section 2.2). Main characteristics of the HotWave framework for dynamic AOP are shown in the Section 2.3. An example use of the HotWave with the DProf is shown in section 3. Section 4 draws the conclusions and outlines future work.

2. DYNAMIC AOP

Dynamic AOP enables runtime adaptation of applications, and consequently monitoring systems, by changing aspects and reweaving code in a running system. In the domain of designing application monitoring tools, dynamic AOP enables creation of tools where developers can refine the set of dynamic metrics of interest and choose the application components to be analyzed while the target application is executing. Such features are essential for analyzing complex, long-running applications, where the comprehensive collection of dynamic metrics in the overall system would cause excessive overheads and reduce developers' productivity. In fact, state-of-the-art profilers, such as the NetBeans Profiler [6], rely on such dynamic adaptation, but currently these tools are implemented with low-level instrumentation techniques, which cause high development effort and costs, and hinder customization and extension.

2.1 Code Hotswapping

Lisp and Smalltalk languages are dynamically typed and allow for runtime manipulation of program code - code hotswapping.

In Java and other statically typed languages, this is much harder to achieve. Hotswapping of loaded classes is usually implemented using JVM Tool Interface (JVMTI) [7]. This approach imposes several constraints. JVMTI requires the use of native code, besides Java. Class interface cannot be changed. New methods and fields are not allowed, and old are not allowed to be changed or removed. Only method bodies can be modified.

Some authors propose extensions to existing JVMs, but none of these extensions [8] has been incorporated in any standard JVM release so far.

2.2 An Overview of Dynamic AOP Frameworks

In standard AOP frameworks two types of weaving are supported.

Compile-time weaving (CTW) is performed when application source code is available. Aspect weaver uses aspect sources or their binary form and weaves them with application classes. Weaver can also work with application binaries in a process known as post-compile weaving. The result of this process are new application binaries.

Load-time weaving is post-compile weaving deferred until class is loaded into JVM. Upon class loading, the class is weaved with aspects, and then it is loaded.

Modern AOP frameworks do not have built-in support for aspect un-weaving or re-weaving at runtime. Dynamic AOP is available only in several experimental frameworks.

Existing dynamic AOP frameworks are implemented using one of three following approaches.

The first approach uses pre-runtime instrumentation to insert hooks - small pieces of code - at locations that can become join-points. These locations are determined using pre-processing, and applied using load-time instrumentation or on just-in-time compilation.

Another approach is to implement runtime event monitoring using low-level JVM support to capture events - method entry/exit and field access.

The most challenging approach is to implement runtime weaving. It can be implemented with customized JVM or using JVM hotswapping support.

PROSE [9] platform has been implemented in three versions, each using one of the aforementioned approaches. The first uses, now obsolete, JVM Profiling Interface [10] to receive notification that application execution reached one of the join points. The second is implemented based on the IBM Jikes Research Virtual Machine and has very large overhead. The third version is implemented for HotSpot and Jikes JVMs. This version is not able to work with code where compiler performed optimizations, such as method inlining.

JasCo [11] introduces new AOP language and concepts of aspect beans and connectors. Aspect beans are used to define join-points and advices. Connectors deploy aspect beans in a concrete component context. The development of JAsCo technology has been stalled for some time now, although it showed promising results.

JBoss AOP [12] and ApsectWerkz [13] support deploying and undeploying of aspects using hotswapping. The downside of both is that they require the knowledge of their own AOP language, based on the XML. Like in JAsCo, bytecode manipulations are performed using Javassist [14].

2.3 HotWave

HotWave [15] uses existing industry standard AspectJ [16] language. It leverages AspectJ compiler and weaver tools. The resulting code conforms to restrictions imposed by hotswapping mechanism. Aspects can be woven right after JVM bootstrapping. Weaving can take place while code is executing. Previously loaded classes are hotswapped with classes woven with new aspects. If the class was already weaved with aspects, new weaving uses original class bytes, not those from previously weaved version.

HotWave lacks support for *around* advices. The workaround is to use a pair of *before* and *after* advices and inter-advice communication. Inter-advice communication allows the creation of synthetic local variables that can be passed between any advice. This is something even AspectJ does not support.

3. USING HOTWAVE WITH DPROF

The use of the HotWave with the DProf will be shown on the example shown in [2]. In this example we show how to monitor the software configuration management application presented in [17].

In that example we used the monitoring probe implemented as aspect shown in Listing 1.

This aspect has one pointcut - `monitoredMethod()`. This pointcut intercepts execution of any method annotated with Kieker's original `@OperationExecutionMonitoringProbe` annotation (line 3). In `around` advice for this pointcut (line 5) we perform measurements. First we measure time before execution of the intercepted method. Using `proceed()` we invoke the intercepted method. After the execution and end time measurement, new monitoring record is created and stored using monitoring controller. The record holds information about execution of the intercepted method and data required for call tree reconstruction.

```
1 public aspect ExecutionTimeMonitoringAspect {
2   // ...
3   pointcut monitoredMethod() :
              execution (@OperationExecutionMonitoringProbe * *(..));
4
5   around() : monitoredMethod() {
6     // ...
7     double startTime = System.nanoTime();
8     try {
9       Object retVal = proceed();
10    } catch (Exception e) {
11      throw e;
12    } finally {
13      double endTime = System.nanoTime();
14      long executionTime = endTime - startTime;
15      DProfExecutionRecord dProfExec =
              new DProfExecutionRecord(..., executionTime);
16      MonitoringController.getInstance().newMonitoringRecord(dProfExec);
17      //...
18      return retVal;
19    }
20  }
21 }
```

Listing 1. Aspect used for monitoring execution time when monitoring using AspectJ

Instead of `@Around("monitoredMethod()")` advice, we have to implement two new advices - `@Before("monitoredMethod()")` and `@Around("monitoredMethod()")`. New aspect is shown in Listing 2.

This aspect has the same pointcut - `monitoredMethod()`. Synthetic local variable `startTime` (annotated with `@SyntheticLV` annotation) holds the value of the method execution start time between `before` and `after` advice execution.

In `before` advice we take time when method execution starts. In `after` advice we take end time, and create and store monitoring record, in the same way as in previous example.

```
1 public aspect ExecutionTimeMonitoringAspect {
2
3   // ...
4   @SyntheticLV
5   public static long startTime;
6
7   pointcut monitoredMethod() : execution
```

```
      (@OperationExecutionMonitoringProbe java.lang.Object *(..));
 8
 9  before() : monitoredMethod() {
10     double startTime = System.nanoTime();
11  }
12
13  after() : monitoredMethod() {
14     double startTime = System.nanoTime();
15     double endTime = System.nanoTime();
16     DProfExecutionRecord dProfExec = new DProfExecutionRecord(...);
17     MonitoringController.getInstance().newMonitoringRecord(dProfExec);
18  }
19}
```

Listing 2. Aspect used for monitoring execution time when monitoring using HotWave

This new aspect is used to monitor call tree shown in Fig. 2. In the `OrganizationFacade.checkOrgName()` we placed time delay to simulate performance lag in this method.

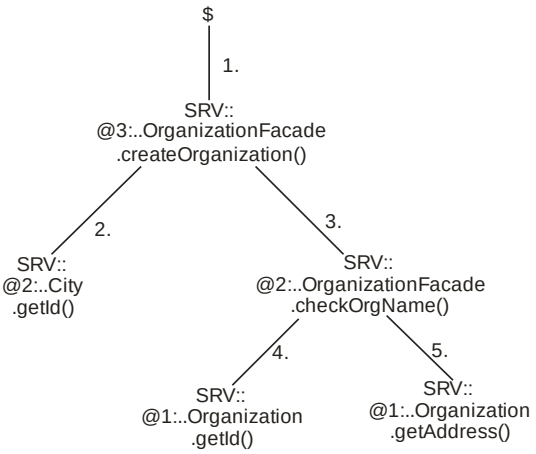

Figure 3. Monitored call tree

In the first pass only root method - `OrganizatonFacade.createOrganization()` - would be weaved and monitored. If deviation from expected performance is detected, another level of nodes is included into monitoring. In the next pass, if no deviation from expected values is detected in execution of `City.getId()` method, and there is a deviation in `OrganizationFacade.checkOrgName()` results, `City.getId()` is unweaved, and methods invoked from `OrganizationFacade.checkOrgName()` were weaved. In the last pass, if no deviation is detected in the results for methods in the lowest level of the call tree, `OrganizationFacade.checkOrgName()` is pronounced the cause of the problem.

In the background, *Analyzer* analyzes obtained data. Based on the analysis results it issues commands to *DProfManager*. These commands contain nodes that are to be weaved or unweaved. *DProfManager* translates commands into *aop.xml* clauses. Based on these clauses HotWave is reweaving classes after each pass.

HotWave should not incur any additional overhead, compared to the results presented in [2].

Overhead peaks[8] are expected only when reweaving is initiated. After reweaving, as is the case with JVM starting, new classes are loaded, linked and just-in-time compiled. This would cause longer execution times at the beginning of each DProf cycle. However, this performance lag is acceptable. As stated in [1, 18], if AspectJ or some other AOP framework is used with DProf, application needs to be restarted. Complete restarts would yield performance lag at the beginning, but also add, very often unacceptable, down time.

4. CONCLUSION

This paper presented the possible use of the HotWave with the DProf for continuous monitoring without the need to restart monitored applications. The result of this integration would be a tool that could be used for continuous monitoring of any kind of applications, including distributed. There is no need to use specific JVM, with dynamic AOP support, or to learn additional AOP language. Applications don't have to be restarted, when monitoring parameters are changes. All changes in monitoring configuration are applied at runtime. The fact that it utilizes industry standard AspectJ platform and language means that only minor changes in existing monitoring probes are required.

The use of the DProf with the HotWave main advantage lies in the fact that monitored application does not have to be restarted when monitoring parameters change. The performance evaluation of both DProf and HotWave guarantees very little overhead. Increased overhead would be experienced only during reweaving. This overhead increase is acceptable considering the fact that restarting the whole application is the only alternative.

Further work depends on the HotWave and DProf development.

5. REFERENCES

[1] Hoorn, A. v., Hasselbring, W., Waller, J., *Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis*, Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012). ACM, Boston, Massachusetts, USA. (2012)

[2] Okanović, D., Hoorn, A. v., Konjović, Z., Vidaković, M., *SLA-Driven Adaptive Monitoring of Distributed Applications for Performance Problem Localization*, Computer Science and Information Systems (in print)

[3] *Java Management Extensions Technology*, Oracle, [Online] http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/
[4] Binder, W., *Portable and Accurate Sampling Profiling for Java*, Software – Practice & Experience, v.36 n.6, p.615-650, 2006.
[5] Kiczales, G., *Aspect-Oriented Programming*, ACM Computing Surveys (CSUR), v.28 n.4. (1996)
[6] *NetBeans: The NetBeans Profiler Project*, Web http://profiler.netbeans.org/
[7] *Java Virtual Machine Tool Interface (JVMTI)*, Oracle, [Online] http://download.oracle.com/javase/6/docs/technotes/guides/jvmti
[8] Wurthinger, T., Ansaloni, D., Binder, W., Wimmer, C., Mossenbock, H., *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 2011, pp. 825–844. ACM, New York (2011)
[9] Nicoara, A., Alonso, G., Roscoe, T., *Controlled, Systematic, and Efficient Code Replacement for Running Java Programs*, In: Eurosys 2008: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008, pp. 233–246. ACM, New York (2008)
[10] *Java Virtual Machine Profiler Interface (JVMPI)*, Oracle, [Online] http://download.oracle.com/javase/1.4.2/docs/guide/jvmpi/jvmpi.html
[11] Suvee, D., Vanderperren, W., Jonckers, V., *JAsCo: An Aspect-Oriented Approach Tailored for Component Based Software Development*. In: AOSD 2003: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, pp. 21–29. ACM, New York (2003)
[12] *JBoss: Open Source Middleware Software*, [Online] http://labs.jboss.com/jbossaop/
[13] Vasseur, A., *Dynamic AOP and RuntimeWeaving for Java – How does AspectWerkz address it?*, In: Dynamic Aspects Workshop (DAW 2004), Lancaster, England (March 2004)
[14] Chiba, S., *Load-Time Structural Reflection in Java*, In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 313–336. Springer, Heidelberg (2000)
[15] Binder, W., Hulaas, J., Moret, P., *Advanced Java Bytecode Instrumentation*, Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java, Lisboa, Portugal. p. 135-144 (2007)
[16] *AspectJ*, [Online] http://www.eclipse.org/aspectj/
[17] Okanović, D., Vidaković, M., *One Implementation of the System for Application Version Tracking and Automatic Updating*, Proceedings of the IASTED International Conference on Software Engineering, Innsbruck, Austria. p. 80-85. (2008)
[18] Okanović, D., Hoorn, A. v., Konjović, Z., Vidaković, M., *Towards Adaptive Monitoring of Java EE Applications*, In Proceedings of the 5th International Conference on Information Technology - ICIT. Amman, Jordan. CD. (2011)

ACKNOWLEDGMENTS

Results presented in this paper are partialy funded as the research conducted within the Grant No. III-44010, Ministry of Science and Technological Development of the Republic of Serbia