Evaluation of Alternative Instrumentation Frameworks

Dušan Okanović, Milan Vidaković

Faculty of Technical Sciences
University of Novi Sad
Fruškogorska 11
Novi Sad, Serbia
oki@uns.ac.rs
minja@uns.ac.rs

Abstract: Our previous research focused on reducing continuous monitoring overhead by employing architectural design that performs adaptive monitoring. In this paper we explore the use of other AOP or AOP-like tools for instrumentation. The goal is to find a tool that has lower overhead than AspectJ. Support for runtime changes of monitoring configuration is also considered. Our main topic of interest is DiSL framework, because it has similar syntax as, already used AspectJ, but allows for more instrumentation options.

1 Introduction

When monitoring application performance parameters under production workload, during continuous monitoring process, everything must be done to reduce the overhead generated by monitoring system. This overhead is highly unwanted because it can have negative effect on the end user's experience. Although inevitable, this overhead can be minimized.

The DProf system [OvHVK13] for continuous monitoring uses instrumentation, and sends gathered data to a remote server for further analysis. After this data is analyzed, new monitoring configuration can be created in order to find the root cause of the performance problem. New parameters turn monitoring off, where performance data is within expected, and on, where data shows problems. This way, the overhead is reduced, because only problematic parts of software are monitored. First implementations of our tool used AspectJ[1] for instrumentation. The downside of this approach was the fact that the bytecode with weaved aspects is not fully optimized. The result was higher overhead than expected. Also, the monitored system had to be restarted each time new monitoring parameters are set.

We have explored the use of other aspect-oriented or similar tools, mainly DiSL framework [MVZ+12], with our system. The goal is to find a tool that has lower overhead than AspectJ. Support for runtime changes of monitoring configuration is also considered.

[1] http://www.eclipse.org/aspectj (October 2014)

Proc. SOSP 2014, Nov. 26–28, 2014, Stuttgart, Germany

The remainder of the paper is structured as follows. In section 2. we provide an overview of the DProf system, current implementation problems, and possible AspectJ alternatives. How to replace AspectJ with DiSL is shown is section 3. In the last section we draw conclusions and outlines for future work.

2 DProf system

The DProf system proposed in [OvHVK13] has been developed for adaptive monitoring of distributed enterprise applications with a low overhead.

For monitoring data acquisition, DProf utilizes the Kieker[2] framework [vHWH12]. The main reason for this was low overhead the Kieker imposes on a monitored system. Unlike profilers and debuggers used by software developers during development process, the Kieker separates monitoring from analysis of gathered data.

Deployment diagram of the DProf monitoring system is shown in figure 1. Monitoring probes gather data. Kieker's Monitoring Controller directs this data to the DProfWriter. DProfWriter sends data into the ResultBuffer. ResultBuffer holds the data and sends it in bulks to the RecordReceiver periodically. RecordReceiver then stores it into the database.

Analyzer performs the analysis of the data, reconstructs call trees from it, and, when required, creates a new set of monitoring parameters. New parameters are created based on configuration file, described in [OVK12]. Parameters are received by DProfManager. This component controls the work of the ResultBuffer and the AspectController. AspectController configures AspectJ framework through modifications in aop.xml configuration file.

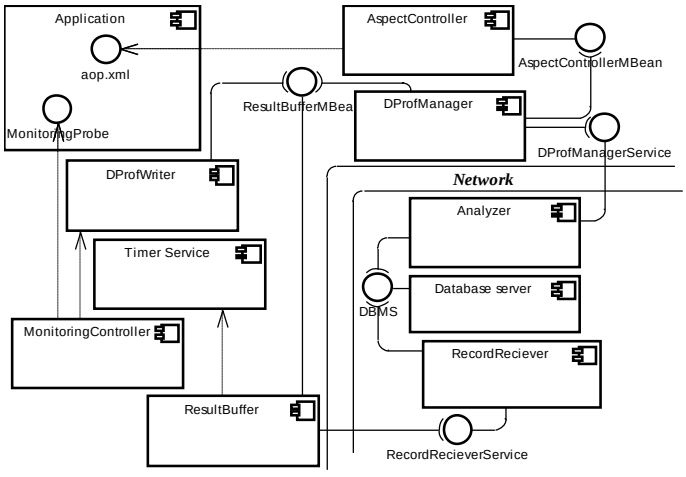

Figure 1. Deployment diagram of the DProf monitoring system

[2] At the time of implementation, Kieker 1.6 was used.

Additional components that support the change of monitoring parameters at application runtime have been developed using Java Management Extensions[3]. Use of the JMX technology allows for the reconfiguration of the DProf monitoring parameters both manually and programatically.

2.1 Limitations of this approach

In some situations the use of AspectJ imposed significant overhead that no architectural redesign of the system could reduce. While this overhead was very small in simple monitoring cases, there were cases where it rose unexpectedly. In those cases reflection and access to join-point information was used.

Another problem is the fact that AspectJ does not allow the join points to be inserted within the methods. To be more precise, they are not able to "see" into a method and check if it contains loops, since these methods are most likely to cause performance lags.

Weaving of aspects is usually performed at load-time. This means that for every monitoring configuration change, the system has to be restarted - both monitoring system and monitored application. This results in reduction of overall quality of service, because availability is reduced. Although careful planning can, as shown in [OV12], but the fact remains: run-time loading and unloading of instrumentation should be employed.

Another approach is to use proxy classes, described in [GHJV94]. All of the methods would be instrumented using aspects, but proxy classes would choose whether to accept their performance data or not. While this approach is very flexible, it adds another layer of classes and method invocations into the system, leading to even more overhead.

It is obvious that AspectJ has to replaced with another framework. This new framework will have to impose very low overhead and support run-time insertion and removal of monitoring probes.

2.2 Possible AspectJ alternatives

Since Java is a statically typed, it is very hard to manipulate code at runtime. In current Java virtual machines (JVMs), it is possible only to replace method bodies, not method signatures. This hotswapping of loaded classes is usually implemented using JVM Tool Interface[4] (JVMTI).

There are several bytecode manipulation libraries. These allow for manipulation at very low level, enabling developers to optimize instrumentation and place probes almost arbitrarily. Some of these tools are ASM[5], BCEL[6], Javassist[7] and Soot[8]. However, the resulting instrumentation code is difficult to read, maintain and debug.

[3] JMX, http://www.oracle.com/technetwork/java/javamail/javamanagement-140525.html
[4] JVMTI, http://docs.oracle.com/javase/7/docs/technotes/guides/jvmti
[5] ASM, http://asm.ow2.org/

Modern AOP tools are based on these bytecode manipulation tools, but provide higher abstraction layer for defining instrumentation. However, AOP has not been designed for profiling and continuous monitoring. AOP and modern AOP tools suffer from supporting limited set of join points, and they do not support instrumentation of basic blocks, loops, or even, byte codes. Also, bytecode generated by AOP tools is not always optimized.

Dynamic AOP [PGA02] approach enables runtime adaptation of applications, and consequently monitoring systems, by changing aspects and reweaving code in a running system. It 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 [Dim03], 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.

Existing dynamic AOP frameworks are implemented using one of the 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 [NAR08] platform has been implemented in three versions. The first uses, now obsolete, JVM Debugging Interface[9]. 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, but is not able to work with code where compiler already performed optimizations, such as method inlining. JAsCo [VSV+05] 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.

HotWave [VBAM09] uses existing industry standard AspectJ language, and generates code that can be used by hotswapping mechanism. Aspects can be woven right after JVM bootstrapping, but also while the code is executing. Previously loaded classes are

[6] BCEL, http://commons.apache.org/proper/commons-bcel/
[7] Javassist, http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist
[8] Soot: a Java Optimization Framework, http://www.sable.mcgill.ca/soot
[9] http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/architecture.html

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. This is acceptable for continuous monitoring because we do not want to change program behavior. Inter-advice communication allows the creation of synthetic local variables that can be passed between any advice. This is something AspectJ and other AOP frameworks do not support. HotWave has never been fully developed, and remained only a prototype.

Another approach similar to HotWave is shown in [WWS10]. DCE VM is an extension of standard JVM. It allows the classes within to be changed while JVM executes. This tool has also been only a prototype.

Domain specific language for instrumentation (DiSL [MVZ+12]) has been developed to counter some of the problems that occur when using AOP for Java software monitoring. Considering the level of abstraction, DiSL is somewhere between low-level tools like ASM, and high-level tools like AspectJ.

Using DiSL guarantees that the monitored software will never change its behavior, something that ASM does not. DiSL developer has to deal with some details of byte code manipulation, but much less than when using tools like ASM. Code generated using AspectJ will always pass bytecode verification, while with DiSL, developer has to ensure that it passes. Unlike AspectJ, it allows for instrumentation to be inserted within methods.

DiSL uses similar pointcut/advice model as AspectJ, and even similar syntax, but it removes some of the constructs. One of the omitted constructs is around advice. This advice is often used by developers of dynamic analysis tools. Instead of it, a combination of before/after advices can be used, with the addition of synthetic local variables for inter-advice communication. DiSL constructs are transformed into code snippets that are inlined before or after indicated bytecode sections. The omission of around is not a problem when constructing monitoring tools. Around advices intended use is changing of program's behavior, something that monitoring should not do.

Performance evaluation of monitoring tools developed with DiSL showed less overhead than AspectJ implementation of such tool, while providing more functionality (e.g. basic block analysis). Code generated by DiSL weaver is smaller, thus making the memory footprint of the monitoring tool smaller.

3. Using DiSL With DProf

Replacing of the AspectJ with the DiSL will be shown on the example already shown in [OvHVK13]. In this example we monitor the software configuration management application.

So far, we have used the monitoring probe implemented as aspect shown in our previous papers. This aspect intercepts execution of annotated method, and in around advice we perform measurements.

Since DiSL does not support around advices, we have to implement two new advices - `@Before(...)` and `@After(...)`. New aspect is shown in Listing 1. Synthetic local variable startTime (annotated with `@SyntheticLV`) 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, and in after advice we take end time, and create and store monitoring record, in the same way as in original aspect.

```
1 public class ExecutionTimeMonitoring {
2   @SyntheticLocal public static long startTime;
3   @Before(marker = BodyMarker.class, guard =
                      DProfAnnotatedGuard.class)
4   static void onMethodEntry() { startTime = System.nanoTime();}
5
6   @After(marker = BodyMarker.class, guard =
                                  DProfAnnotatedGuard.class) {
7   static void onMethodExit(MethodStaticContext msc) {
8      double endTime = System.nanoTime();
9      DProfExecutionRecord dProfExec =
          new DProfExecutionRecord(..., endTime - startTime, ...);
10     MonitoringController.getInstance()
          .newMonitoringRecord(dProfExec);
11  }
12 }
```

Listing 1. DiSL aspect for execution time monitoring

This class is weaved with application classes and is used to monitor call tree shown in Fig. 2.

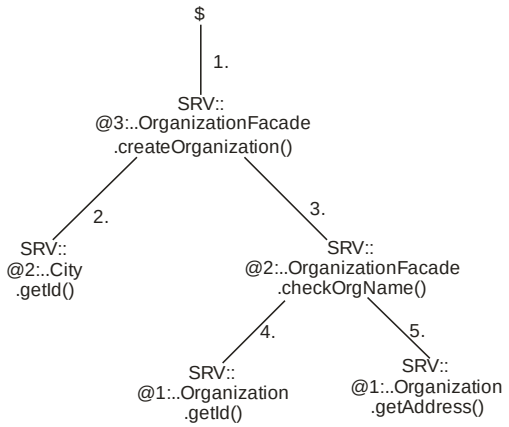

Figure 2. Monitored call tree

In the first pass only the root method - was weaved and monitored. When deviation from expected performance was detected, another level of nodes was included into monitoring. In the next pass, no deviation from expected values was detected in execution of `City.getId()` method, while there was a deviation in `OrganizationFacade.checkOrgName()` results. `City.getId()` was un-weaved, and methods invoked from `OrganizationFacade.checkOrgName()` were weaved. In the last pass, no deviation was detected in the results for methods in the lowest level of the call tree, and `OrganizationFacade.checkOrgName()` was declared to be the cause of the problem.

In the background, Analyzer analyzed the obtained data. Based on the analysis results, it issued commands to the DProfManager. Based on these commands, DiSL performed aspect weaving.

The main goal of this experiment was to measure performance overhead ogenerated by monitoring. The overhead was measured by repeatedly invoking monitored method (monitoring configuration was fixed). On our test platform, implementation that uses DiSL yielded approximately 1.2% less overhead then AspectJ implementation.

Some performance peaks were detected, in both cases, but only when weaving was initiated, on application restart. After reweaving, as when JVM restarts, new classes are loaded, linked and just-in-time compiled. This causes longer execution times at the beginning of each DProf cycle.

4 Conclusion

This paper presented the use of the DiSL framework with the DProf system in order to reduce performance overhead. The result is a tool that can be used for continuous monitoring of any kind of applications, including distributed applications. Because DiSL uses similar syntax to AspectJ, and is fully Java based, learning curve for this new tool is not an issue.

The evaluation of DProf/DiSL combination was performed by monitoring the sample software configuration management application, which was monitored using DProf on AspectJ platform in our previous work. The comparison of the results shows that the generated overhead is slightly less when using DiSL.

Further work depends on the DiSL development. FRANC [AKZ+13] framework, built on DiSL, will provide the possibility to implement runtime changing of monitoring parameters.

References

[AKZ+13] Ansaloni, D.; Kell, S.; Zheng, Y.; Bulej, L.; Binder, W.; Tůma, P.: Enabling modularity and re-use in dynamic program analysis tools for the java virtual machine. In

Proceedings of the 27th European conference on Object-Oriented Programming (ECOOP'13). Springer-Verlag, Berlin, Heidelberg, p. 352-377. 2013.
[BLC02] Bruneton, E.; Lenglet, R.; Coupaye, T.: ASM: A Code Manipulation Tool to Implement adaptable systems. http://asm.ow2.org/current/asm-eng.pdf. 2013.
[Dim03] Dimitriev, M.: Design of JFluid. Technical Report: SERIES13103. Sun Microsystems Inc., USA. 2003.
[GHJV94] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. 1994.
[MVZ+12] Marek, L.; Villazón, A.; Zheng, Y.; Ansaloni, D.; Binder, W.; Qi Z.: DiSL: a domain-specific language for bytecode instrumentation. In Proceedings of the 11th annual international conference on Aspect-oriented Software Development (AOSD '12). p. 239-250. 2012.
[NAR08] Nicoara, A.; Alonso, G.; Roscoe, T.: Controlled, Systematic, and Efficient Code Replacement for Running Java Programs. SIGOPS Operating Systems. v. 42, n. 4. p. 233-246. 2008.
[OV12] Okanović, D.; Vidaković, M.: Software Performance Prediction Using Linear Regression. In Proceedings of the 2nd International Conference on Information Society Technology and Management. Kopaonik, Serbia. p. 60-64. 2012.
[OVK12] Okanović, D.; Vidaković, M.; Konjović, Z.: Monitoring of JEE Applications and Performance Prediction. Journal of Information Technology and Applications, v.1, n.2. p. 136-143. 2012.
[OvHKZ13] Okanović D.; van Hoorn, A.; Vidaković, M.; Konjović, Z.: SLA-Driven Adaptive Monitoring of Distributed Applications for Performance Problem Localization, Computer Science and Information Systems (ComSIS), vol. 10, no. 1, pp. 25-50, 2013.
[PGA02] Andrei Popovici, Thomas Gross, and Gustavo Alonso. 2002. Dynamic weaving for aspect-oriented programming. In Proceedings of the 1st international conference on Aspect-oriented software development (AOSD '02). ACM, New York, NY, USA, p. 141-147
[VBAM09] Villazón, A.; Binder, W.; Ansaloni, D.; Moret, P.: Advanced Runtime Adaptation for Java. SIGPLAN Not. v. 45, n. 2. p. 85-94. 2009.
[vHWH12] van Hoorn A.; 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. p. 247-248. 2012.
[VSV+05] Vanderperren, W.; Suvée, D.; Verheecke, B.; Cibrán, M. A.; Jonckers, V.: Adaptive Programming in JAsCo. In Proceedings of the 4th international conference on Aspect-oriented software development (AOSD '05). p. 75-86. 2005.
[WWS10] Würthinger, T.; Wimmer, C.; Stadler, L.: Dynamic Code Evolution for Java. 8th International Conference on the Principles and Practice of Programming in Java, Vienna, Austria. 2010.

Acknowledgement

The research presented in this paper was supported by the Ministry of Science and Technological Development of the Republic of Serbia, grant III-44010, Title: Intelligent Systems for Software Product Development and Business Support based on Models.