Towards Performance Monitoring Overhead Reduction

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

Abstract—Continuous monitoring tools are designed to perform well under production workload with minimal performance overhead. Standard AOP frameworks are popular choice for development of instrumentation for Java software monitoring tools. Inevitable consequence of using these tools is the occurrence of performance overhead. The code they generate is guaranteed to be correct, but always has a slight overhead. On the other hand, if a low-level bytecode manipulation tools are chosen, they may have better performance, but development time is higher and code may not be correct. The paper demonstrates the use of a new framework that allows instrumentation of Java code using AOP-like syntax, but has less overhead than industry standard AOP tools. Already developed monitoring tool has been implemented using this new framework, with better results.

I. INTRODUCTION

The main difference between continuous monitoring and profiling is that the latter happens in a controlled environment. It does not take into account problems that can happen in the "real world". Continuous monitoring of software can provide a better picture of dynamic software behavior under production workload. In comparison to debugging, continuous monitoring allows for detection of software bugs that would be very hard to reproduce during software development. These bugs are, usually, the cause of the phenomenon known as software aging [1]. Software aging is a process where application performance degrade over time and, most often, manifests itself in production environment, after a long running time. Causes of software aging are bugs that do not immediately lead to failure. Memory leaks and unreleased locks are some of the examples.

The data obtained from monitoring can be used for various purposes. For instance, it can be used as a basis for architecture-based software optimization, visualization, and reconstruction [2]. Using this data, 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.

While profiling and debugging, developers do not care for performance overhead, it is very important issue in continuous monitoring. The monitoring system has to work using a minimal amount of resources in order not to interfere with the 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.

Our system for continuous monitoring - DPprof - uses aspect-oriented programming (AOP) [3] for instrumentation, and sends gathered data to the remote server for further analysis. After the data is analyzed, new monitoring configuration can be created in order to find the root cause of the performance problem. By monitoring only suspicious parts of software we reduce overall overhead. We have chosen to use AspectJ framework [4], since it is de facto standard in this field. The downside of this approach is the fact that resulting bytecode is not fully optimized, having the overhead slightly higher than expected. Also, the monitored system has to be restarted each time monitoring configuration changes in order to apply new monitoring parameters.

In this paper we explore using other AOP or AOP-like tools for source code instrumentation. We are looking for a tool that has lower overhead than AspectJ, which was used in previous implementation of our system. The support of runtime change of monitoring configuration is also considered.

Main contributions of this paper are following. We describe our system for continuous monitoring and how it can reduce monitoring overhead by monitoring only suspicious parts of the monitored software. We describe limitations of industry standard AOP tools when they are used for continuous monitoring. We provide an example of how the DPprof system can be used with the DiSL tool [5], instead of AspectJ, to further reduce monitoring overhead.

The remainder of this paper is structured as follows. Section 2 presents our system for continuous monitoring. In Section 3 we describe limitations of our current implementation and what is needed to improve its performance. An overview of tools that can be used with the DPprof in order to improve its performance, and related work are shown in Section 4. An example use of the DiSL tool with the DPprof is shown in section 5. Section 6 draws the conclusions and outlines future work.

II. DPprof SYSTEM FOR CONTINUOUS MONITORING

The DPprof system proposed in [6] has been developed for adaptive monitoring of distributed enterprise applications with a low overhead.

The DPprof utilizes the Kieker framework [2] for the monitoring data acquisition. The Kieker was chosen because it imposes very low overhead to a monitored system. Unlike profilers and debuggers used by software developers, the Kieker separates monitoring and data gathering from further data analysis. Monitoring is performed on one computer system using software probes. Obtained data - Monitoring Records - is stored in Monitoring Log/Streams and is available for analysis on completely different computer system.
Components of the Kieker framework are shown in Fig. 1.

To further reduce monitoring overhead, DProf introduces two architectural features. First, DProf does not send records into Monitoring Log immediately. Rather, records are buffered and sent periodically, in bulks. This way, DProf generates only periodical peaks in performance overhead. These peaks are removed in the process of analysis using statistical methods [7]. The second feature that helps in overhead reduction is an addition of adaptive monitoring.

DProf configuration specifies which of the application's call trees are going to be monitored and number of nesting levels within the call tree that will be monitored. A call tree represents calling relationships between software methods [8]. It contains the control-flow of method executions invoked by a client request. The first method is called the "root". DProf stores data in a central database, regardless of on how many computers the monitored application is being executed. It uses a mechanism integrated into the Kieker framework, which, during data gathering, uniquely identifies each method execution within a trace. Each execution is assigned a number which represents the order of execution. This allows call trees to be spread on different computers.

Human testers usually monitor only parts of application causing problems, while other parts of the system are not monitored and they perform without any monitoring overhead. Testers turn monitoring on and off in parts of the system in order to localize the method causing performance problems. DProf mimics this 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. Lower nodes of other trees are not monitored, keeping the monitoring overhead at bare minimum. Other systems usually perform monitoring of the whole software, regardless of the fact that other parts (i.e. other call trees) are working fine.

DProf monitoring probes are usually inserted into application code using aspect-oriented programming (AOP). Inserting methods directly into software code is also possible, but it is not recommended. AOP reduces, 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. Kieker’s reference implementation uses AspectJ, so we adopted this approach for the DProf.

Monitoring probes are inserted using annotations. Kieker’s original annotation @OperationExecutionMonitoringProbe is used most often, but custom annotations can be used, as well. Aspects are written to intercept these annotated methods and store monitoring data. In usual scenario, aspects use around advices. After the time is measured, proceed() is invoked in order to execute intercepted method. After it completes, the time is measured again.

Kieker’s Monitoring Record has been extended in order to support not only temporal data, but also other performance parameters - memory consumption, CPU load, etc.

Deployment diagram of the DProf monitoring system is shown in Fig. 2.

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 analyzes call trees reconstructed from this data, automatically creates a new set of monitoring parameters, and sends them to the DProfManager, if needed. DProfManager is the component that controls the work of the ResultBuffer and the AspectController. AspectController configures AspectJ framework i.e. "tells" which classes should be weaved.

These additional components that support the change of monitoring parameters at application runtime have been developed using Java Management Extensions (JMX) [9]. Use of the JMX technology allows for the reconfiguration of the DProf monitoring parameters manually, by system administrators using any JMX console.
III. CURRENT IMPLEMENTATION LIMITATIONS

As stated in the previous chapter, DProf uses AspectJ framework for source code instrumentation. The main problem we have encountered while using AspectJ is the fact that it imposes higher overhead in some situations. Although this overhead is very small in simple monitoring cases, use of reflection and access to join-point information can be time consuming.

Another problem is the fact that AspectJ works at the method level. Aspects cannot intercept anything within the methods. For example, they are not able to look beyond the method signature and check if the method contains loops, since these methods are most likely to cause performance lags.

A more difficult problem is weaving of aspects. It is usually performed at load-time. The main problem with this implementation is the fact that every time monitoring configuration changes, complete system restart is required. Long running systems’ monitoring also suffers because of this, since their availability is reduced, thus reducing the overall quality of service. In case of these systems, careful planning is required, as shown in [10], but the fact remains: run-time loading and unloading of instrumentation should be employed.

One approach would be through proxy classes [11]. 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 allows for type-safe reflective dispatch of invocations on objects, it adds another layer of classes and method invocations, which leads to even more overhead.

In order to solve these issues, it is clear that another AOP, or AOP-like, framework is required. It needs to impose very low overhead and support run-time insertion and removal of monitoring probes.

IV. RELATED WORK

As Java is a statically typed language, runtime code manipulation is very hard to achieve. In current Java virtual machines (JVMs), dynamic changes in classes are possible through the hotswapping mechanism [12]. However, it is possible only to change method bodies, not method signatures. It is impossible to add and remove fields and methods at runtime, change inheritance, etc. Hotswapping of loaded classes is usually implemented using JVM Tool Interface (JVMTI) [13].

There are several bytecode engineering libraries. These allow for bytecode manipulation at very low level, enabling developers to optimize instrumentation and place probes almost arbitrarily. Some of these tools are ASM [14], BCEL [15], Javassist [16] and Soot [17]. The downside of this approach is that the resulting implementation code is difficult to read, maintain and debug.

Modern AOP tools usually use one of these tools, but provide higher abstraction layer for defining instrumentation. The problem is the fact that AOP has not been designed for profiling, let alone continuous monitoring. AOP languages provide support for limited set of join points. There is no support for instrumentation of basic blocks, loops, or even, byte codes. Also, bytecode generated by AOP tools is not always optimized.

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 [18], 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 [19] platform has been implemented in three versions, each using one of the aforementioned approaches. The first uses, now obsolete, JVM Debugging Interface [JVMDI] 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 already performed optimizations, such as method inlining.

JAsCo [20] 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.

HotWave [21] uses existing industry standard AspectJ 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. 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 [22]. 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 [5]) 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 bytecode 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.

V. USING DiSL WITH DPROF

The use of the DiSL with the DProf will be shown on the example shown in [6]. In this example we monitor the software configuration management application presented in [23].

In the author's previous work 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 intercepted method and data required for call tree reconstruction.

Since DiSL does not support around advices, we have to implement two new advices - @Before(...) and @After(...). New aspect is shown in Listing 2.

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. DProfAnnotatedGuard class (not shown here) is used to filter only methods annotated with @OperationExecutionMonitoringProbe annotation.

This class is weaved with application classes and is used to monitor call tree shown in Fig. 3.

In the first pass only the root method - OrganizatonFacade.createOrganization() - 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

\[\text{DiSL uses AspectJ-like annotations for pointcuts and advices}\]
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 pronounced the cause of the problem.

```java
public aspect ExecutionTimeMonitoringAspect {
  // ...
  pointcut monitoredMethod() :
    execution (@OperationExecutionMonitoringProbe * *(..));

  around() : monitoredMethod() {
    // ...
    double startTime = System.nanoTime();
    try {
      Object retVal = proceed();
    } catch (Exception e) {
      throw e;
    } finally {
      double endTime = System.nanoTime();
      long executionTime = endTime - startTime;
      DProfExecutionRecord dProfExec = new DProfExecutionRecord(..., executionTime);
      MonitoringController.getInstance().newMonitoringRecord(dProfExec);
      // ...
      return retVal;
    }
}
```

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

```java
public class ExecutionTimeMonitoring {
  // ...
  @SyntheticLocal public static long startTime;

  @Before(marker = BodyMarker.class, guard = DProfAnnotatedGuard.class)
  static void onMethodEntry() {
    startTime = System.nanoTime();
  }

  @After(marker = BodyMarker.class, guard = DProfAnnotatedGuard.class)
  static void onMethodExit(MethodStaticContext msc) {
    double endTime = System.nanoTime();
    DProfExecutionRecord dProfExec = new DProfExecutionRecord(..., endTime - startTime, ...);
    MonitoringController.getInstance().newMonitoringRecord(dProfExec);
  }
}
```

Listing 2. Monitoring using DiSL

In the background, Analyzer analyzed the obtained data. Based on the analysis results it issued commands to the DProfManager. These commands contained nodes that are to be weaved or un-weaved. DProfManager translated commands into the configuration applied by the AspectController. Based on these configuration parameters, DiSL was reweaving classes after each pass.

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

We have measured overhead in both implementations by repeatedly invoking monitored method (monitoring configuration was fixed). The obtained results are shown in Table 1.

Overhead peaks were detected, in both cases, on application start. In both cases, underlying tool performs weaving of the instrumentation code into application classes. After weaving, as is the case with JVM starting, new classes are loaded, linked and just-in-time compiled. This causes longer execution times at the beginning of each DProf cycle. After that, the application resumes normally.

Implementation that uses DiSL yielded approximately 1.2% less overhead then AspectJ implementation, as
shown in Table 1. The first test cycle was excluded from average time calculation, since it includes class loading and linking.

![Figure 3. Monitored call tree](image)

### Table I. Execution times in nanoseconds

<table>
<thead>
<tr>
<th>Test cycle</th>
<th>AspectJ</th>
<th>DiSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1908021</td>
<td>2350077</td>
</tr>
<tr>
<td>2</td>
<td>153781</td>
<td>151523</td>
</tr>
<tr>
<td>3</td>
<td>152677</td>
<td>151628</td>
</tr>
<tr>
<td>4</td>
<td>152739</td>
<td>151391</td>
</tr>
<tr>
<td>5</td>
<td>153192</td>
<td>151284</td>
</tr>
<tr>
<td>6</td>
<td>152760</td>
<td>151152</td>
</tr>
<tr>
<td>7</td>
<td>152116</td>
<td>152520</td>
</tr>
<tr>
<td>8</td>
<td>152806</td>
<td>150627</td>
</tr>
<tr>
<td>9</td>
<td>153604</td>
<td>151810</td>
</tr>
<tr>
<td>10</td>
<td>153905</td>
<td>151409</td>
</tr>
<tr>
<td>Average</td>
<td>153064.4</td>
<td>151452.7</td>
</tr>
</tbody>
</table>

### VI. CONCLUSION

This paper presented the use of the DiSL framework with the DPProf 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. Because DiSL uses similar syntax to AspectJ, and is fully Java based, learning of this new tool is not an issue.

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

Further work depends on the DiSL development. FRANC [24] framework, built on DiSL, will provide the possibility to implement runtime changing of monitoring parameters.

### ACKNOWLEDGMENT

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.

### REFERENCES


