ECOOP'03 workshop on Advancing the State of the Art in Runtime Inspection (ASARTI), 2003 July.

Using HotSwap for Implementing Dynamic AOP Systems

Shigeru Chiba[1] Yoshiki Sato[1] Michiaki Tatsubori[2]

[1] Dept. of Mathematical and Computing Sciences
Tokyo Institute of Technology
{chiba,yoshiki}@csg.is.titech.ac.jp
[2] IBM Tokyo Research Laboratory
mich@trl.ibm.com

1 Introduction

Practical demands on dynamic aspect-oriented programming (AOP) are getting well recognized. For example, logging functionality is a typical application of AOP but the usefulness of this functionality is limited without dynamic AOP. When we are debugging a program, we tend to want dynamically adding or removing various logging aspects without restarting the program. Suppose that the program is a Web application server and there is a bug that appears only after long product run. If the program is restarted for logging, all the internal data structures are reset and hence the context causing the bug would be lost.

Another crosscutting concern that should be dynamically woven is security fixes. Suppose that your organization is running a web application server for selling your products. Since such a web server cannot stop except scheduled maintenance time, if it turns out that the server has a security problem, a patch fixing the problem should be applied to the server without shutting it down. If a better patch is released later, the previous patch should be removed and instead the new one should be applied. The patch will be used until the server is rebooted after the bug is fixed in a clean way at scheduled maintenance time.

Developing an implementation technique for dynamic AOP systems is still a research topic especially in Java. Since one of the most important features of Java is platform independence, dynamic AOP systems are required to be built on top of the standard Java virtual machine (JVM). A modified JVM for dynamic AOP is not acceptable in practice.

This paper presents our Java-based dynamic AOP system called *Wool*. For better performance than other systems, Wool is implemented with our novel technique exploiting the *HotSwap* mechanism recently introduced by the Java2 SDK 1.4. This mechanism allows us to dynamically reload a class file to update the class definition. However, naively using this mechanism does not improve execution performance. This paper mentions how this mechanism should be used with others to really improve performance.

This work was supported in part by the CREST program of Japan Science and Technology Corp.

2 Just-in-time Hook Insertion

Like other dynamic AOP systems, Wool weaves an aspect by *hooking* the thread of control at the join points identified by pointcuts. It is currently implemented as a Java library; it dose not provide any aspect language for easily writing an aspect. The supported join points are method calls (both caller and callee sides), field accesses, object instantiation, and exception handlers. The Wool users can use before and after advice but not around advice. The advice is described as a Java method that receives a `Joinpoint` object as a parameter. The `Joinpoint` object represents the runtime context at a join point as the `thisJoinPoint` object in AspectJ. Wool does not allow introduction since the HotSwap does not allow reloading a class file to which a new method or field is appended.

First, Wool starts running a program without inserting hooks or any other code changes. If an aspect is dynamically woven during the runtime, Wool first requests the JVM through the JPDA (Java Platform Debugger Architecture) to set break points at all the join points identified by a pointcut. If the thread of control reaches one of the break point, the JPDA suspends the thread and notifies Wool that the thread reaches the join point. Then, Wool runs the advice associated with that join point.

Since the JPDA is designed for debuggers, the program that the JPDA notifies must be a process different from the JVM process. Hence Wool exists in a different JVM process and the advice is executed in this process. If the advice must refer to the runtime context at the join point, it must obtain them through a `Joinpoint` object that Wool passes to the advice. The `Joinpoint` object encapsulates details of the JPDA to access the target JVM.

Since context switches between the two processes occur whenever the thread reaches a break point, using break points as hooks implies serious performance overhead. To reduce the overheads, after executing the advice, Wool always replaces the method including the join points identified by pointcuts with the modified method in which hooks and the advice are directly embedded. The method body is modified at the bytecode level so that a hook, that is, a bytecode sequence for executing the advice is embedded at the join point contained in that method body. Wool uses the HotSwap mechanism of the JPDA for unloading the original class file and reloading a modified class file. It uses Javassist[2] for modifying a class file. After replacing the method, a break point is never set at the join points contained in the method body. Therefore, the execution overheads are minimized. Note that Wool does not immediately reload the modified class file when the aspect is woven. Instead Wool sets break points since the HotSwap mechanism is not available till the program execution is suspended by the JPDA.

There is an exceptional case that Wool has to delay substituting a method in which hooks are embedded. This is when a join point identified by a pointcut is contained in the method currently being executed. For example, suppose that a draw method in a Rectangle class is currently being executed and the activation frame associated with that method is on the execution stack. If the class file of Rectangle is reloaded with the HotSwap mechanism, however, the execution of the draw method with that activation frame is still being performed according to

the definition of the original draw method given by the old class file. Thus, the hooks contained in the new class file are not effective for that execution. On the other hand, the hooks are effective for the execution of the draw method started after the reloading. For example, if the draw method recursively call itself after the class file is reloaded, then the second call of draw is executed with the new class file. To avoid this problem, Wool does not reload the class file until the first call of the draw method finishes and the activation frame is popped from the stack.

Wool also has to be careful of the execution of a pair consisting of before and after advice woven at the same join point. If that pair is woven accidentally while the method containing that join point is executed, only the after advice will be executed at the end of that execution. The before advice will not be executed since the method execution had already been started. This behavior might cause a problem if the after advice depends on the results of the before advice. For example, the before advice might record the current time and the after advice might use that value to compute the elapsed time. In this case, the after advice must not be executed unless the corresponding before advice was executed. To solve this problem, Wool allows the programmers to select the behavior in that case. It also allows them to control precisely when an aspect is woven.

3 Preliminary Experiments

To evaluate effects of our implementation technique, we measured the execution time of the `jess` benchmark program from SPECjvm98. The program was run with the HotSpot Client JVM (Java2 SDK 1.4.0) on Sun Blade 1000 (dual UltraSPARC III 750MHz and 1 GB memory).

We executed `jess` to solve two problems (the number puzzle and the monkey banana). `jess` is an expert system, which receives a problem description and solves the problem. The `null` before-advice was woven in a method body in one or four classes when the program started. For comparison, we also measured the execution time of the program statically woven by AspectJ and the program in which advice was woven only with breakpoints but without the HotSwap mechanism.

Table 1 lists the results. Compared to the implementation only with breakpoints, the hooks embedded by the HotSwap mechanism significantly improved execution performance despite extra overheads due to the HotSwap. The only exception is the monkey banana in which advice was woven only in one method body; since the frequency of the execution of that advice was relatively low, the improvement by the HotSwap did not exceed the overheads.

We also broke down the execution time measured with Wool. The tuples indicated by † represent the elapsed time for handling break points, for reloading a class file with HotSwap, and for the rest of the computation. Note that Wool first sets break points since the program execution must be suspended at break points before using the HotSwap. The times for handling break points were equal among the four tests since they are independent of the solved problems, which are

Table 1. Elapsed Time of `jess` (msec)

input	num. of advice	AspectJ (static)	breakpoint only	Wool (HotSwap)
Number puzzle	4	8,590	7,388,812	19,638 (1,680 + 4,057 + 13,901)†
	1	8,522	23,307	11,832 (1,680 + 806 + 9,346)†
Monkey banana	4	1,063	45,817	11,003 (1,680 + 4,057 + 5,266)†
	1	1,003	3,833	3,993 (1,680 + 806 + 1,507)†

†(handling breakpoints + HotSwap + the rest)

input data, and the numbers of advice bodies. The time for reloading a class file includes the time for modifying the class file. It is approximately proportional to the number of modified class files, which is equal to the number of advice bodies in our experiment. It is independent of the input data.

Unfortunately, the execution performance of Wool was still lower than that of AspectJ even if the time for handling break points and reloading a class file is excluded. This is mainly due to constructing a `Joinpoint` object, which represents runtime context. On the other hand, AspectJ constructs a `thisJoinPoint` object only if advice needs it. Since the advice we used for this experiment does not need it, the execution performance of AspectJ is better than Wool. We chose the current design of Wool since we want to provide the same interface to advice no matter how hooks are implemented, by break points or the HotSwap. If hooks are implemented by break points, advice needs a `Joinpoint` object for accessing the context at the join point in a target JVM process.

Another overhead of Wool is that it runs in debug mode although AspectJ runs in normal mode. The overhead due to the debug mode is about 5% or less according to our other experiment using the Java 2 SDK 1.4 and SPECjvm98.

These results of our experiments show that the cost of embedding hooks in a program by using the HotSpot is not negligible. To make this cost relatively small, the program must run long after dynamically weaving an aspect. Also, the current design of Wool with respect to the `Joinpoint` object should be revised to reduce the overhead of the embedded hooks. Our new technique described in another paper [3] would help this.

4 Related Work

There are several implementation techniques for dynamic AOP that have been proposed so far. The most naive technique is to modify the JVM but it is not acceptable for Java since portability is significant in Java. PROSE [5] uses breakpoints for hooking the thread of control. This technique, however, implies sub-

stantial performance penalties as we mentioned above. JAC [4] and HandiWrap [1] statically transform a program at compilation time so that hooks are embedded at all the join points. Since the hooks are embedded even at the join points that are not identified by pointcuts, performance penalties due to the hooks are not acceptable.

5 Summary

This paper proposes our new implementation technique for dynamic AOP systems. It also shows *Wool*, which is our dynamic AOP system for Java based on the proposed technique. It integrates a technique using breakpoints provided by the debugger interface of the JVM and a technique using the HotSwap mechanism, which allows us to reload an class file that has been already loaded. The results of our preliminary experiments showed that our technique can achieve good execution performance against other implementation techniques of dynamic AOP. Our technique would allow software developers to use dynamic AOP for practical applications.

References

1. Baker, J. and W. Hsieh, "Runtime aspect weaving through metaprogramming," in *Proc. of Int'l Conf. on Aspect-Oriented Software Development (AOSD'02)*, pp. 86–95, ACM Press, 2002.
2. Chiba, S., "Load-time structural reflection in Java," in *ECOOP 2000*, LNCS 1850, pp. 313–336, Springer-Verlag, 2000.
3. Chiba, S. and M. Nishizawa, "An Easy-to-Use Toolkit for Efficient Java Bytecode Translators," in *Proc. of Generative Programming and Component Engineering (GPCE '03)*, (to appear), 2003.
4. Pawlak, R., L. Seinturier, L. Duchien, and G. Florin, "JAC: A Flexible Solution for Aspect-Oriented Programming in Java," in *Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, LNCS 2192, pp. 1–24, Springer, 2001.
5. Popovici, A., T. Gross, and G. Alonso, "Dynamic Weaving for Aspect-Oriented Programming," in *Proc. of Int'l Conf. on Aspect-Oriented Software Development (AOSD'02)*, pp. 141–147, ACM Press, 2002.