Controlled, Systematic, and Efficient Code Replacement for Running Java Programs *

Angela Nicoara Gustavo Alonso Timothy Roscoe

Department of Computer Science
ETH Zurich, 8092 Zurich, Switzerland
{anicoara, alonso, troscoe}@inf.ethz.ch

ABSTRACT

In this paper we present PROSE, a system that performs reversible and systematic changes to running Java applications without requiring them to be shut down. PROSE is motivated by scenarios such as hotfixes, online program instrumentation and debugging, and evolution of critical legacy applications. In PROSE, changes to running applications are performed by replacing method bodies. To select which code to replace, PROSE supports matching based on both type information and regular expressions. New code can invoke the method it replaces, facilitating code evolution. Changes are composable, and may be reordered or selectively withdrawn at any time. Furthermore, the dynamic changes are expressed as Java classes rather than through an additional programming language. We describe the architecture of PROSE, the challenges of using aggressive inlining to achieve performance, and use standard benchmarks to demonstrate code performance comparable with, or better than, compile-time systems from the Aspect-Oriented Programming community.

Categories and Subject Descriptors: D.2.11 [Software Engineering]: Software Architectures; D.3.3 [Programming Languages]: Language Constructs and Features; D.3.4 [Programming Languages]: Processors—*Run-time environments, Compilers*; C.2.4 [Distributed Systems]: Distributed applications

General Terms: Design, Languages, Measurement, Performance

Keywords: PROSE, Run-time method code replacement, Dynamic bytecode instrumentation, Run-time modification, Inlining

1. INTRODUCTION

In this paper we present PROSE, a system which performs efficient, controlled, and systematic modification of the code of running Java services, without requiring shutdown of the application.

*The work presented in this paper was supported (in part) by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

EuroSys'08, April 1–4, 2008, Glasgow, Scotland, UK.

By *controlled*, we mean that modifications can be undone at run-time as easily as they were applied in the first place, and that multiple modifications to a running system's code compose in a manner that is deterministic, predictable, and can be controlled by the programmer.

By *systematic*, we mean that modifications are automatically applied at multiple points in the code according to criteria specified by the programmer. We borrow concepts from the field of Aspect-Oriented Programming (AOP) to allow so-called "crosscutting concerns" to be addressed, for example to apply a modification at every occurrence of a service invocation, or to instrument all changes to the value of a system parameter or field.

By *efficient*, we mean that widespread crosscutting modifications to a running program can be performed with negligible interruption to service (and in any case much less disruption than that caused by restarting the program), and that the modified system continues to run with performance comparable to a fully recompiled and restarted system – indeed, our benchmarks demonstrate that PROSE introduces a negligible overhead on the JVM and has a performance comparable to tools performing code replacement at compile-time.

PROSE works by atomically replacing collections of methods in the application with new code. The methods to be replaced are identified using type signatures and regular expressions. The code that replaces a given method can also make use of the method it replaces. Moreover, multiple modifications can be applied and undone in arbitrary order, permitting orthogonal modifications of the application to be performed without interference.

Since it works purely by method replacement, PROSE cannot change Java interfaces, class schemas (the set of fields, methods, etc. that a class provides), or perform any complex refactoring of the application. However, the combination of method replacement with the ability to invoke the original code still allows significant flexibility in code evolution, replacement of functionality, and extensions or modifications to application behavior. We present usage scenarios for PROSE below.

PROSE not only allows such systematic modifications to be applied to a Java application in controlled manner, but performs such modifications at run-time. This has considerable advantages as there are many cases where recompiling and restarting might be undesirable. For instance, mission-critical applications might not tolerate downtimes of more than a few milliseconds – the design of Erlang's code replacement functionality [4] for phone switch software was motivated by this requirement, for example. Alternatively, changes might be required quickly in moments of crisis, where taking down the system would cause as much disruption as the problem itself, as might be the case with fixes

to denial-of-service vulnerabilities or modifications to deal with unexpected changes in workload characteristics.

1.1 Contributions

This paper makes three principal contributions.

First, we demonstrate that it is feasible to apply systematic changes to a running Java program, and to do so with negligible loss in run-time performance via aggressive inlining of existing method code in the replacement function. We considerably outperform other tools that work at load-time and run-time, and our performance is comparable with several compile-time approaches.

Second, the efficiency we gain allows us to layer code modifications in such a way that any given change can be applied or withdrawn independently of the others.

Finally, we show how such modifications can be expressed as Java classes themselves. This obviates the need for developers to learn a new and potentially complex language, and allows PROSE to leverage existing environments such as Eclipse [14] with minimal programming overhead.

While it has proved straightforward to port PROSE between Java Virtual Machine implementations, in its current form PROSE is certainly quite Java-specific. Nevertheless, we feel the ideas here have value in the context of other language virtual machines, such as the Microsoft CLR [13].

The rest of this paper is structured as follows. In the next section we motivate PROSE by discussing various usage scenarios for the system. Section 3 then describes the facilities we provide for modification of a running Java application. Section 4 discusses in detail how PROSE is implemented, in particular the main challenges in inlining code for performance. In Section 5 we show performance results for PROSE, and compare it with compile-time, load-time and run-time approaches in use today. Section 7 addresses safety concerns with the modifying running applications, and Section 6 situates PROSE in relation to other work. We conclude in Section 8.

2. USAGE SCENARIOS

PROSE is motivated by a number of related scenarios in running online application services, driven by the observation that most deployed code inside application servers is bespoke, and often written to short-term deadlines without longer-term evolvability in mind.

To take a concrete example, we have used PROSE to dynamically add caching to a running multi-tier Java application. PROSE is used to add new code that implements caching at either the frontend (web server tier), the application logic tier, or at the database layer. We then benchmark the different versions of the system using TPC-W [27]. The results are shown in Figure 1 where the vertical axis shows the average response time and the horizontal axis the different queries used in the benchmark. Adding such caching of intermediate results at different levels of the application is a simple concept in theory, and can lead to dramatic performance improvements, but is hard to implement in practice because the existing system is often not factored with this kind of facility in mind.

A further message of Figure 1 is that the most appropriate enhancement to the application is highly workload-dependent, and is hard to predict in advance. Rather than requiring refactoring of the system, or extensive manual annotation of the code (with the associated degradation of the system's structure), PROSE allows us to make an extension or other modification orthogonally to changes

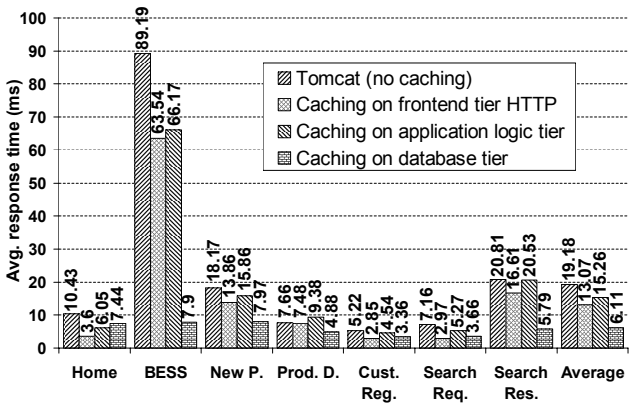

Figure 1. Adding dynamic caching strategies to a multi-tier Java application with PROSE

to the rest of the system, and with almost negligible overhead. The changes can be tried individually or concurrently, and reversed at will.

Particular usage scenarios we have in mind include:

Performance instrumentation and profiling. Online services have complex performance characteristics, and as workloads change, so different areas of a complex system may become performance bottlenecks. Identifying such bottlenecks is greatly facilitated by instrumenting code in ways that are rarely foreseen when the system was first designed, and share the same challenges as logging and debugging modifications. PROSE allows developers to dynamically introduce or remove instrumentation at very precise points in the code by, e.g., executing certain code before or after the execution of a given set of methods. It also allows programmers to test alternative methods for a given task and see which one performs better without having to shutdown and recompile the entire system for each test.

Online logging and debugging. Identifying such bottlenecks is greatly facilitated by instrumenting code in ways that are rarely foreseen when the system was first designed. However, adding such profiling (for example, tracking changes to a portion of global state) often requires extensive widespread (though locally small) changes to code. Performing these changes manually is time-consuming, and it is hard to ensure complete coverage of all applicable cases. Even with good version control, it can also be difficult to remove a coherent collection of such changes at a later date when they are no longer required, particularly if a set of unrelated modifications to the code have occurred in the meantime. PROSE provides a powerful and systematic language for specifying where to perform changes. It also provides several facilities to express, in a single piece of code, changes at several functionally different locations of the original application.

Adding feedback control loops. A recent trend in building large, online services has been the inclusion of automated control loops (either with or without an explicit control-theoretic treatment) to allow components of the service to adapt to changing external conditions. These conditions might include offered load, network conditions, network-based attacks, or changes in local policy. When automated, such control loops are at the heart of the notion of *autonomic computing*. Adding the sensing portion of such control loops can be regarded as a specialized form of

performance instrumentation, and adding the actuation parts shares similar crosscutting issues to instrumentation. Perhaps for this reason, most forms of autonomic computing to date treat programs as black-boxes, and operating solely on the execution environment (operating systems, hardware, network QoS, etc.). PROSE is convenient for implementing both the sensing and actuator parts of an autonomic system, even in systems that were not originally designed with that purpose in mind.

Hotfixes and security patches. Finally, devising fixes for problems that emerge after the code is operational can be hard, particularly for security issues which can exhibit the same crosscutting properties as our other examples above. PROSE can be used for example to modify the arguments passed to a method to avoid a known bug for certain value ranges, or substitute a method for a more secure or better performing version of the code.

3. PROGRAMMING EXTENSIONS

In this section we describe PROSE as seen by a programmer developing run-time extensions, deferring implementation details to Section 4. Here we focus on the practical use of PROSE to give a better idea of its purpose and scope. PROSE's key functionality is *method code replacement at run-time*: replacing the bytecode of a method on a running application without interrupting it. Changes are specified as Java objects using normal Java syntax. In what follows we describe the basic programming elements and procedures, the scope of the changes PROSE facilitates, how changes are expressed in Java, how to specify what to change, and the interface to PROSE.

3.1 Terminology and Basic Concepts

Although PROSE is implemented differently and has different goals, for convenience we borrow terms from the field of compile-time *Aspect-Oriented Programming* [19].

We refer to the changes or extensions to an application as *aspects*. An aspect consists of one or more *crosscuts*. A crosscut contains a piece of *advice* code and an optional pattern expression called a *pointcut*. The advice is the new code to be inserted into the application. PROSE implements modification by replacing the original code by the code in the advice. Due to the constraints imposed by operating at run-time, this is done only for readily and efficiently identifiable blocks of code, namely methods. Hence, upon insertion of an advice, a call to the original method results in the advice being called instead, with the same parameters as the original method.

The advice can choose to call the original method using the `proceed` statement. We sometimes say that the advice *redefines* the original method. If several advices redefine the same method, each `proceed` calls the advices in order according to a user-defined priority. The last advice in the resulting *proceed chain* will call the original method by executing `proceed`. An advice may call `proceed` any number of times.

A crosscut allows the same advice pattern to be applied to multiple methods. Precisely which methods a crosscut applies to is specified by a combination of the advice and the pointcut: the advice uses a special signature that defines a list of methods potentially affected, and the pointcut refines this list by applying a more powerful regular expression.

Aspects provide an abstraction for bundling a set of crosscuts into a single, coherent compound modification to the application. The changes specified by an aspect are inserted or withdrawn from the program as a unit.

3.2 Scope of Modifications

In PROSE, modifications affect only methods, not method invocations – what an advice redefines is a method, not the calls to that method. PROSE can thus implement any change that involves extending functionality (doing something before or after executing a method, e.g., to add logging, timers, or retries), replacing functionality (modifying arguments passed to a method to avoid a known bug for certain value ranges, substituting a method for a better version of the same code), and varied forms of code evolution. PROSE does not support *schema changes* such as replacing the interface of a method, *refactoring* of an application, or the addition of new class members (i.e., methods, fields) to a Java class in the original code. These limitations are largely imposed by PROSE's requirement to operate at run-time, after all potential classes are loaded and linked by the JVM.

In spite of these limitations, PROSE can be used to perform a wide variety of useful changes on running applications. Using `proceed` calls allows advice code to be inserted both before and after original method invocations, and the ability to call `proceed` multiple times (with different arguments, and from multiple places in the advice) allows, for example, complex early exception handling or retry logic to be added to particular areas of an application after it has been deployed.

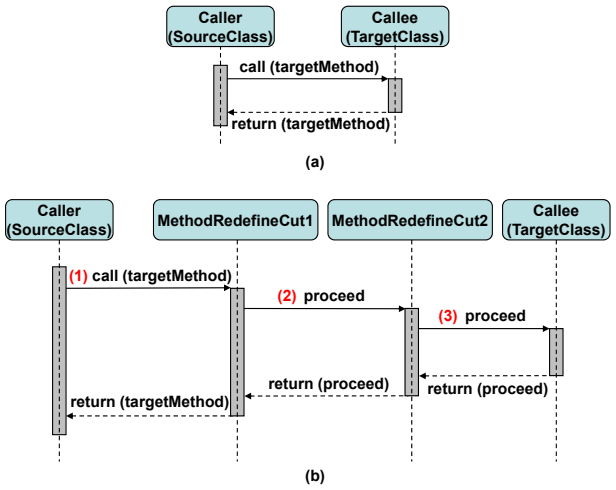

Figure 2. Control: **(a)** before weaving, **(b)** after weaving

To illustrate `proceed` calls further, Figure 2(a) shows a `Caller` class that invokes the `targetMethod` of a `Callee` class. Figure 2(b) shows the result of applying two aspects, `MethodRedefineCut2` and `MethodRedefineCut1`, in that order. Inserting the aspects causes `targetMethod` to be redefined twice. A call to `targetMethod` invokes the last inserted advice instead (*step 1*). Inside this advice, `proceed` calls the second advice (*step 2*). When this second advice executes `proceed`, the original method is called (*step 3*).

3.3 Programming Extensions in Java

An important and unique feature of PROSE is that changes are specified entirely in Java. This is fundamentally different to AOP systems where programmers usually employ a different language for aspects [6, 10, 15].

An aspect in PROSE is a Java class containing one or more crosscut objects. Figure 3 shows an example aspect that implements a cache for expensive computations. The new advice checks

the cache to see if the result is there. If it is, it returns the result from the cache, otherwise it calls the original method using `proceed`. The aspect, called `CachingAspect`, contains a single crosscut and, like all aspects, extends the `DefaultAspect` base class. The crosscut `checkCache` is an instance of `MethodRedefineCut`, and contains an advice (lines 8-19) and a pointcut (lines 21-23), both implemented as methods. The advice is called `METHOD_ARGS` and takes a set of arguments which form the signature used to define potential methods to be replaced. The arguments in the example (line 8) indicate that the advice will match all methods of all object types (`ANY target`) that have an object as a first parameter (`Object arg1`) followed by any number of additional parameters (`REST arg2`). This set of methods is further qualified by the pointcut, which in this case, specifies (line 22) that the advice applies only to methods called `expensiveComputation`.

```
1  public class CachingAspect extends DefaultAspect {
2     public static CachingService service;
3
4     public Crosscut checkCache = new MethodRedefineCut() {
5        public Object proceed (Object arg1, REST arg2) { return null; }
6
7        // advice: match '*.*(Object,..)'
8        public Object METHOD_ARGS(ANY target, Object arg1, REST arg2) {
9              Object result;
10             if (!service.lookUp(arg1)) {
11                // store the value in the cache when the original method is called
12                result = service.writeInCache(arg1, proceed (arg1, arg2));
13                return result;
14             }
15             else {
16                // return the value from the cache
17                return service.accessCache(arg1);
18             }
19       }
20       // specialization: match '*.expensiveComputation(..)'
21       protected PointCutter pointCutter() {
22             return (Within.method("expensiveComputation"));
23       }
24    }
25 }
```

Figure 3. An aspect implementing a cache

3.4 Specifying the Methods to Replace

The methods to be replaced by an advice are specified using both the advice and the pointcut. In the advice, the expression is derived from the method signature of the advice itself (the advice automatically gets passed the parameters of the original method so it does not really need a signature). The signature of the advice method (`METHOD_ARGS`) has a first parameter called `target`. The type of `target` is used to match object types and defines a list of classes where the advice might potentially apply. With the `target` parameter it is possible to use the `ANY` wildcard: `target` will then match all object types. The second and remainder parameters of the advice have names of the form `arg1`, `arg2`, `arg3`, and so on. They are used to identify the methods to be replaced by matching their parameters. For these parameters it is possible to use the `REST` wildcard that matches any set of parameters of arbitrary types, and also the `ANY` wildcard that matches all object and primitive types.

The signature pattern of the advice is, however, not precise enough in all cases. This is why a crosscut includes a pointcut to refine the match. The pointcut can take several qualifiers: `within`, `this` and `target`. The `within` qualifier exists in four versions. The first three are: `Within.type(A)`, `Within.subType(A)`, `Within.superType(A)`, that match methods that are in a class `A`, a subclass of `A`, or a superclass of `A`, respectively. The fourth version is `Within.method (pattern)`, where `pattern` is a regular expression that will be matched against method names. The `this` qualifier allows matching of methods based on various properties of the `this` object: `This.type(A)`, `This.subtypeOf(A)`, `This.supertypeOf(A)` matches methods that are of type `A`, a subtype of `A`, or a supertype of `A`. `This.isSameObject(B)` matches methods in the same object as `B`. The `target` qualifier matches methods based on properties of the target object: `Target.subtypeOf(A)` matches methods if the current target is a subclass of `A`.

3.5 The `proceed` Facility

The requirement that PROSE aspects must be well-formed Java classes requires a special treatment of the `proceed` statement. The approach we have followed in PROSE is to implement `proceed` statements as a method call, the corresponding method being defined as part of the crosscut class. An example can be seen in Figure 3, line 5, which defines the `proceed` statement used in the advice (in line 12).

Since `proceed` stands in for a call to either another advice or to the original method, there is the question of how to handle the parameters and the return of the call. For this purpose, PROSE offer two options: *predefined proceed statements* and *custom proceed statements*.

Predefined `proceed` statements are a collection of `proceed` statements where the return type of the `proceed` call is fixed. These statements are defined within the crosscut class and are thus available to the advice. By default, the predefined `proceed` statements automatically include the arguments of the original call. The main purpose of the predefined `proceed` statements is to make things easier for the programmer. They are used for invoking advices or original methods without having to declare a `proceed` method on the crosscut class. They are also useful for automatically autoboxing the return value to an appropriate type. The complete list of predefined `proceed` statements is shown in Table 1. Predefined `proceed` statements are declared to be `static` for performance reasons, `final` to prevent them from being overridden, and `protected` to prevent them from being called from outside of the crosscut.

void **proceed**();	
boolean **proceedBoolean**();	boolean[] **proceedBooleanArray**();
byte **proceedByte**();	byte[] **proceedByteArray**();
char **proceedChar**();	char[] **proceedCharArray**();
double **proceedDouble**();	double[] **proceedDoubleArray**();
float **proceedFloat**();	float[] **proceedFloatArray**();
int **proceedInt**();	int[] **proceedIntArray**();
long **proceedLong**();	long[] **proceedLongArray**();
Object **proceedObject**();	Object[] **proceedObjectArray**();

Table 1. Predefined `proceed` statements

As an example of how to use predefined `proceed`, consider a method that does some calculation returning a `Float` and keeps track of how many values have been computed by storing them in a database. The method turns out to be inaccurate for small values and slightly off for larger values. To correct the problem, an advice can be used as follows. The advice checks the value passed to the method, if the values are small, the advice calls `proceed` () so that the original method can keep count but it does the computation itself. By using the predefined `proceed` with a `void` return value, the return parameter of the original method is ignored. If the value is large enough, the advice calls `proceedDouble()` to get the computation done and recorded by the original method. The

returned value is cast into a `Double` and can then be plugged into an expression that produces the correct value.

Predefined `proceed` statements work best when it is very clear what the return value and the parameters involved are. There are many cases, however, where it is advantageous to write a single advice to replace a relatively large collection of methods that might not all have the same signature. In such cases, custom `proceed` statements are used. A custom `proceed` allows the programmer to generate arbitrarily complex signatures, particularly when combined with *signature patterns* (e.g., `ANY`, `REST`). The only constraint is that the signature must match (be assignable to) all methods that will be replaced by the advice. If a `proceed` without return value is declared as `void`, the value returned by the original method is not accessible in the advice. A custom `proceed` also has the slight performance advantage that it avoids the casting of the `Object` returned by `proceedObject()` to the class instance returned by the advice.

Custom `proceed` statements are a very powerful construct that must be used with care. However, when judiciously used, they significantly increase what can be expressed in PROSE. For instance, custom `proceed` makes it possible to write a single advice for a collection of related but different methods.

3.6 Working with PROSE

PROSE is installed on a JVM and runs as a service in its own thread. It can be accessed locally or remotely over a TCP connection. Remote and local access is supported by an Eclipse plug-in that greatly facilitates its use, as well as a number of visual tools to allow insertion and removal of extensions by drag and drop. Other tools help to visualize the methods to be replaced, compare the affected methods of different aspects, and synchronously activating them on multiple remote JVMs. PROSE supports transactional semantics for insertion and removal so that it can be safely used in distributed applications.

Inserting an aspect involves instantiating the aspect, setting its insertion priority, and telling the system to insert it. The following code fragment shows how local insertion takes place:

```
1.  CachingAspect asp = new CachingAspect();
2.  asp.setPriority(10);
3.  ProseSystem.getAspectManager().insert(asp);
```

An instance of the class `CachingAspect` called `asp` is created on line 1, given a priority (line 2) that determines the order in which it will be applied relative to other aspects, and then inserted in the aspect manager (line 3). Alternatively, aspect instances may be sent to PROSE using the remote interface of the aspect manager. In such cases, the aspect instances are generated and initialized outside the JVM in which they will be inserted.

4. IMPLEMENTATION

To be practically useful on real systems, applications which have been modified by PROSE must exhibit minimal performance degradation after modification. The approach adopted by existing systems which implement Java method replacement is typically to interpose a new method invocation (containing the advice), and implement `proceed` by performing a method invocation on the original code. This is accomplished by either adding new methods to the invoked class, creating a new, anonymous class as an intermediary, or by wrapping the original method code in a so-called closure object. However, we show in Section 5 that this results in a significant run-time performance penalty.

PROSE achieves good performance by creating no new methods, classes, or closures. Instead, we replace the original method code entirely, and implement `proceed` using aggressive inlining of the original code in the body of the redefined method. This is a delicate business: there is a tension between the expressivity and generality of PROSE's code modification interface, and the complexity of handling all of Java's language facilities within the modification.

In the rest of this section we first give an architectural overview of PROSE, and how it interfaces to two different virtual machine implementations. We then go on to describe in detail how an individual method is rewritten to incorporate changes. The principal challenges are (1) transforming method bytecode into a code sequence suitable for inlining into the new advice; (2) avoiding an explosion in code size when multiple crosscuts and/or `proceed`s are applied to the same method; (3) handling the complexity of Java's call conventions in the form of exceptions, variable argument lists, and autoboxing; and (4) composing multiple aspects in a controlled manner. We proceed by starting with a simplified description of the process, and progressively filling in the gaps.

4.1 System Overview

The first step of any changes performed with PROSE is to weave an advice into a running application. Upon insertion, PROSE parses the aspect and extracts the information necessary to determine which methods to change (assume for simplicity the aspect has only one crosscut, otherwise the procedure is repeated for each one of them). It then generates the bytecode for the advice, inlining the original method when necessary (see below). The actual weaving can be implemented in different ways, depending on the underlying JVM. PROSE has been implemented so far in two JVMs: Sun's HotSpot JVM [25] and the IBM Jikes Research Virtual Machine (RVM) [3]. Most of the code of PROSE is VM-independent and identical in both systems, what varies is the technique used for weaving. These weaving techniques are both well known and orthogonal to PROSE.

The HotSpot backend for PROSE requires no changes to the underlying virtual machine, and uses the HotSwap mechanism [12] of the Java Platform Debugger Architecture (JPDA), which allows a class to be reloaded at run-time, effectively replacing all methods at the same time. PROSE interfaces to HotSpot as a JVM plugin. For weaving an aspect, it rewrites the corresponding class with the new method and reloads the class. This results in the original method being replaced by the advice. The safety of the update is guaranteed by HotSwap, which takes care of pausing the JVM at a safe point, doing the change, and resuming execution.

The Jikes implementation of PROSE uses dynamic weaving support to change a class at run-time in the RVM [21]. It is based on making the JIT recompile the original method by changing the corresponding entry in the method table. PROSE triggers the recompilation but what is compiled is the advice rather than the original method. This way, calls to the original method immediately become calls to the advice. This approach is more general as it can be used in any JVM with a JIT. It has the drawback, however, that certain compiler optimizations interfere with the ability of PROSE to find methods: if the bytecode compiler inlines a method, PROSE will not be able to find it.

Based on our implementation experience so far, PROSE should be relatively easy to retarget for other JVMs. There is also nothing in the techniques used that would not make the general idea also applicable to other kinds of language virtual machines, such as Microsoft's CLR [13].

4.2 Basic Method Replacement

The simplest non-trivial case for method replacement is a single instance of a crosscut where the advice (i.e. the replacement code) has a single, conventional `proceed` call. This `proceed` invocation in the advice code must be replaced with an inlined version of the original method.

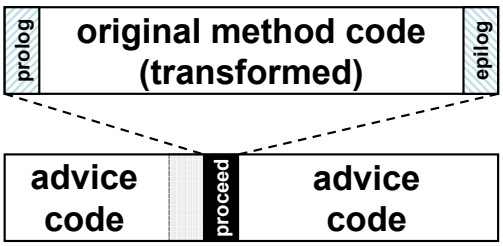

Figure 4. Single `proceed` inlining

The process is shown in Figure 4. The first challenge arises from the use of `ANY` and `REST` patterns in the signature of the new advice, which allows the same crosscut to apply to multiple methods, with potentially different signatures. For each method which will be replaced by the new advice code, the advice bytecode must be modified so that its arguments match those of the original method.

The second issue is that both the new advice code and the original method code use local variables, which the Java compiler allocates to slots in the current stack frame. The slot numbers used by local variables in the original method code must therefore be redefined so that they are allocated after the slots used for the new advice code. The exception to this is the slot reserved for the implicit "`this`" value, i.e. the current object. The bytecode of the original method is rewritten to relocate these local references, and the prolog and epilog of the original method are rewritten to obtain argument values from advice local variables and store return results back.

Finally, all `return` instructions in the original method are replaced with `gotos` to the instruction following the transformed code.

4.3 Handling Multiple `proceeds`

The basic scheme above generates highly efficient code for cases where a single `proceed` call exists in the advice clause. However, it is not uncommon for an advice code to call the original method from more than one place – in particular, when installing functionality to retry an unreliable operation some number of times, perhaps with modified arguments.

Performing direct inlining in the case where `proceed` functions are called multiple times from the advice code poses a challenge. Naive inlining would increase the code size considerably, apart from being conceptually inelegant. Furthermore, such a code size explosion would increase the risk of running up against Java's 64kB limit on the bytecode size of each method.

An alternative approach would be to call the original method code as a subroutine by moving it into a separate method (either a new class method, instance method, or a method of an entirely new class), and indeed some compile-time systems do just this (e.g. [6]). However, the most efficient method call in Java (calling a static method on the same class) is still a surprisingly heavyweight operation.

Instead, in cases where we would otherwise need to inline the original method more than once, we adopt the (perhaps surprising)

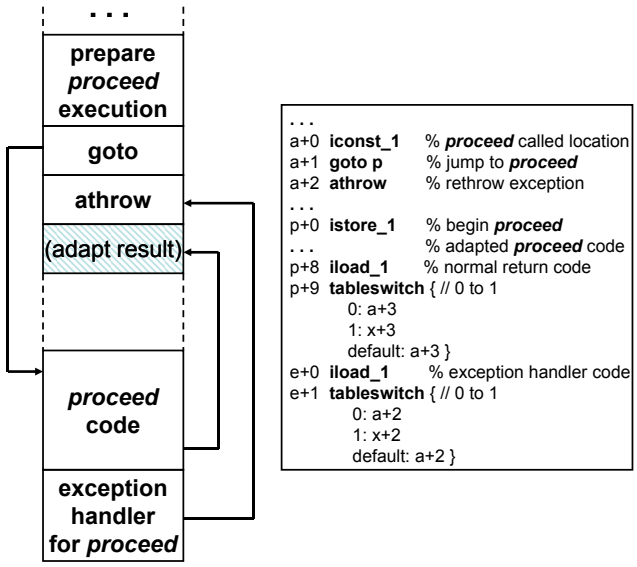

Figure 5. Multiple `proceed` inlining

approach of *simulating* subroutines within the new, replacement method. The (transformed) original method code is placed at the end of the code block for the new method, and each time it needs to be invoked, an integer value is pushed onto the operand stack indicating where the call is being made from (note that the stack is not used to pass arguments at this point, since all arguments are by now local variables). This instruction is then followed by a `goto` to the original method code.

The epilog of the original method is followed by code to pop this integer from the stack, and a Java `tableswitch` bytecode instruction implementing a jump table to calculate the correct address to return to. Figure 5 shows the complete structure of the resulting code; exceptions are also handled by a similar jump table, a scheme we describe in more detail in Section 4.7.

As we show in Section 5, the code resulting from this transformation has remarkably good performance, while keeping the size under control. Note that PROSE can still run up against Java's method size limit, however. In practice we have not found this to be a problem so far, but were it to become so we would have to "spill" code to a newly created method or class and accept the associated performance penalty.

4.4 Method Arguments and Return Values

As explained in Section 4.2, PROSE generates code ahead of an inlined original method to copy method arguments to the newly allocated local variable slots.

These arguments are of two kinds. The first kind are explicitly passed to the `proceed` call in the advice code written by the user, in which case the advice bytecode contains instructions to push them on the operand stack just before the point in the code where the `proceed` call instruction occurs (and which will be replaced by the inlined original method).

The second kind are implicit. For example, the programmer can simply write `proceed` () and have all the original arguments passed from the advice code to the original method. These arguments will already reside in local variable slots, since they will have been pulled off the argument stack at the start of the advice code – indeed, it is acceptable for the advice code to change their values.

These implicit arguments must be copied into the original method's local variable slots.

Return values from the original method call are handled in a similar way. Where a plain `proceed` statement (declared `void`) is employed for a method with a return value, the epilog discards this value from the stack.

The shuffling around of arguments and return values theoretically results in suboptimal performance; one could imagine adopting a more sophisticated approach to allocating local variable slots which borrowed techniques like register-coloring from compilers to eliminate the redundant memory and stack operations. We do not currently explore that possibility with PROSE, in part because the current implementation is easier to check for bugs than one performing more complex transformations in code, partly because we do not expect the overhead to be too high, but also because we feel that a good just-in-time bytecode compiler may well be able optimize out many of these operations anyway. Our optimism is borne out by the performance results we report in Section 5.

4.5 Autoboxing

Recent versions of Java (J2SE 5.0 and after) support a feature known as "autoboxing" – implicit conversions between concrete types like `int` and `long` and their object counterparts like `Integer` and `Long`.

PROSE supports autoboxing of both arguments and results of `proceed` calls. Since the types of the actual and real parameters of the inlined method call are known when the new method is being generated, PROSE inserts appropriate instructions where necessary to perform boxing and unboxing of concrete type values (integers and floats of various sizes, plus characters and booleans).

These instruction sequences themselves can perform implicit casting so, for example, if the boxed type to be converted to an integer is not `java.lang.Integer` but one which is castable to it, the code will still work. As a (minor) added bonus, our autoboxing of arguments and results also works on older JVMs whose compiler does not support general autoboxing. PROSE performs a typecheck via the Java reflection model before applying the aspect, and attempts to autobox between incompatible types will generate an exception when inserting the new aspect, causing the insertion transaction to fail.

4.6 Varargs

As well as autoboxing, version 5.0 of Java J2SE introduced variable-length argument lists to functions along the lines of the C `varargs` facility. A Java method can be declared with a varargs parameter as its last argument, denoted by the type for its entries followed by an ellipsis and the name of the argument, for example "`proceed(Object... extra_args)`". The Java compiler converts this into an array which is passed in a single argument to the callee for each point that the method is called from.

PROSE provides support for varargs arguments in `proceed` calls. However, efficiently inlining a varargs call invocation turns out to be difficult, because of the code generated by Java compilers to implement the calling conventions for varargs functions. Figure 6 shows the code sequence generated for a `proceed` call with variable arguments: first, an array is created on the stack, then each argument is created on the stack and inserted into the array, and finally `proceed` is called with the array argument.

The challenge is to eliminate the performance penalty of the redundant creation and subsequent deletion of the argument array when inlining the original method in place of the `proceed` call.

```
...
iconst <size>          % creation of the array
anewarray              % pushes the arrayref onto the stack

dup                    % insertion of the first entry
iconst <0>             % (1) pushes the index onto the stack
...                    % (2) instructions creating the entry
aastore                % (3) stores the entry into the array

...                    % additional array entries

dup                    % insertion of the last entry
iconst <max index>     % (1) pushes the index onto the stack
...                    % (2) instructions creating the entry
aastore                % (3) stores the entry into the array

invokevirtual proceed(Object[] obj)  % proceed invocation
```

Figure 6. Bytecode instructions for a call to a method with variable arguments

This is difficult since PROSE is working from the bytecode, and while it is clear where `proceed` is called from, it is less clear where the argument marshalling starts. Furthermore, the marshalling code for each argument contains arbitrary instructions to calculate the argument value before inserting it in the array.

PROSE works backwards from the `proceed` call instruction, and identifies the code blocks for marshalling each argument by keeping track of how each instruction affects the current stack size (since every Java bytecode instruction affects the stack size by a constant amount). The code for each argument is identified by the presence of an `aastore` instruction where the stack size is the same as when `proceed` is later called. Eventually, the code to create the argument array itself is identified, and only at this point can the number of arguments to the call be found.

4.7 Exception Handling

Perhaps the biggest challenge in efficiently inlining method calls when implementing crosscuts is the handling of exceptions. Both original methods and advice code can throw and catch arbitrary exceptions – indeed, a common use case for applying fixes to software on-the-fly is installing exception handlers to catch errors at different levels in the system. PROSE needs to preserve the behavior of exception handlers when inlining original method code into new advice.

Figure 7 shows the simple case of adding a new exception handler around an existing method invocation, and where the existing method is invoked only once inside the advice. In this case, the inlining is straightforward: the new handler is placed after the inlined method body, and the exception table for the new method created accordingly.

The situation is more complex when the new advice code invokes the original method more than once, via multiple `proceed` calls. Recall from Section 4.3 that PROSE uses values pushed on the stack, `gotos`, and a jump table at the end of the inlined code to efficiently invoke the `proceed` code multiple times. Obviously, exceptions thrown within this single body of inlined code must be handled differently depending on where in the new method the code was invoked from. The situation is further complicated by the fact that a `proceed` call might be called from within the exception handler surrounding a previous `proceed`, for example to retry a failed operation.

PROSE handles this by surrounding the inlined original method with a general exception handler which catches all exceptions. This handler then uses a second jump table to transfer execution to an

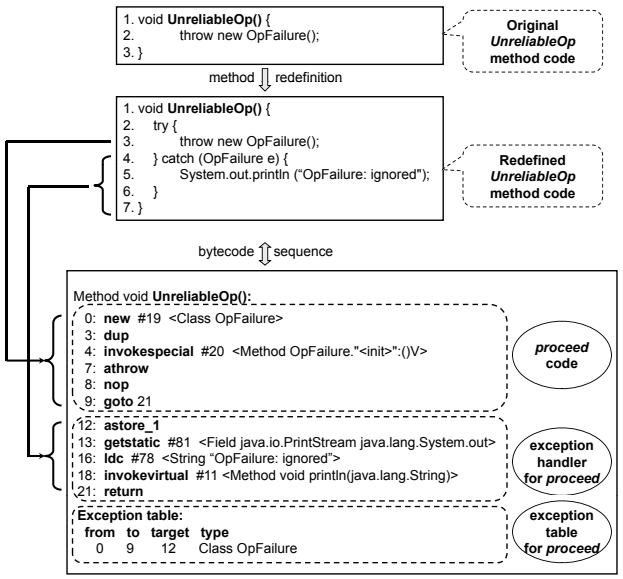

Figure 7. Exception handling for single `proceed` inlining

appropriate `athrow` instruction in the advice code. The exception table for the new method is then constructed so that each of these `athrow` instructions causes the correct exception handler (if any) to be invoked. The resulting code layout is shown in Figure 8.

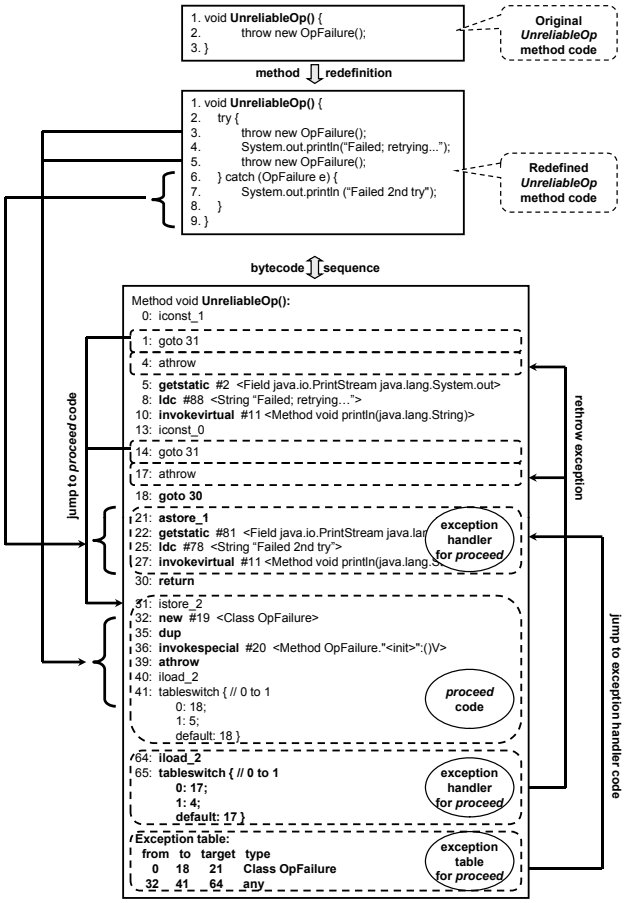

Figure 8. Exception handling for multiple `proceed` inlining

4.8 Multiple Method Redefinitions

It is highly likely in a real system modified with PROSE that more than one crosscut will redefine the same method. PROSE handles the composition of multiple aspects (and crosscuts within a single aspect) by nesting the method redefinitions according to a simple ordering specified when the aspects are introduced: each aspect has an associated "priority". This provides a deterministic ordering of redefinitions across multiple insertions and removals.

When inserting and removing method definitions, PROSE simply generates new code for the method every time, derived from the original method and the new combination of applicable advice sections. This is conceptually simpler and less bug-prone than trying to incrementally unpick an advice section from a method, and is still computationally relatively cheap. Furthermore, this recalculation does not affect the time taken to hotswap the method, since it is effectively done out of band until the point where it is swapped.

5. EVALUATION

In this section we present performance measurements of PROSE. Our aim is to determine the overhead of PROSE when installed in a JVM, the efficiency of the code that PROSE generates when replacing a method at run-time, and the cost of inserting a new aspect into a system.

5.1 Experimental Methodology

All experiments were performed on an Intel Pentium M-based system at 1.73 GHz with 512 MB RAM running Linux 2.6.17. The Java implementations used are Sun's Java SDK 1.5.0 JVM and IBM's Jikes RVM 2.3.0.1. In the case of Jikes, adaptive inlining is turned off (see the problem with inlined methods described above).

For our measurements we use three benchmarks: Java Grande [18], SPECjvm98 [24], and JAC [17]. SPECjvm98 consists of several tests that measure the efficiency of the JVM, just-in-time (JIT) compiler, and operating system interface. It includes applications for text compression, MPEG decoding, compilation speed, graphics, and database processing. The Java Grande benchmark tests the performance of Java for scientific computations and covers three areas: low-level benchmarks that test general language features and operations, scientific and numerical application kernels, and full-scale science and engineering applications. We use the scientific and numerical application kernels, short codes which reflect the type of operations that might be found in the most computationally intense parts of numerical application. JAC is a benchmark for Aspect-Oriented Programming systems, and consists of a set of public methods with different method signatures and simple method body implementation. We use JAC to compare PROSE to commonly used Aspect-Oriented Programming systems.

5.2 PROSE Overhead

Our first set of experiments measure the overhead of having PROSE reside in the JVM without performing any method replacements. We run the SPECjvm98 and Java Grande benchmarks on a JVM with and without PROSE. For Java Grande, we report the average times for one hundred runs, each run completed in a separate VM. For SPECjvm98, we report the average execution times of one hundred runs, all run in a single JVM execution, with an input size of 100 (large). In addition, for Jikes we show results with

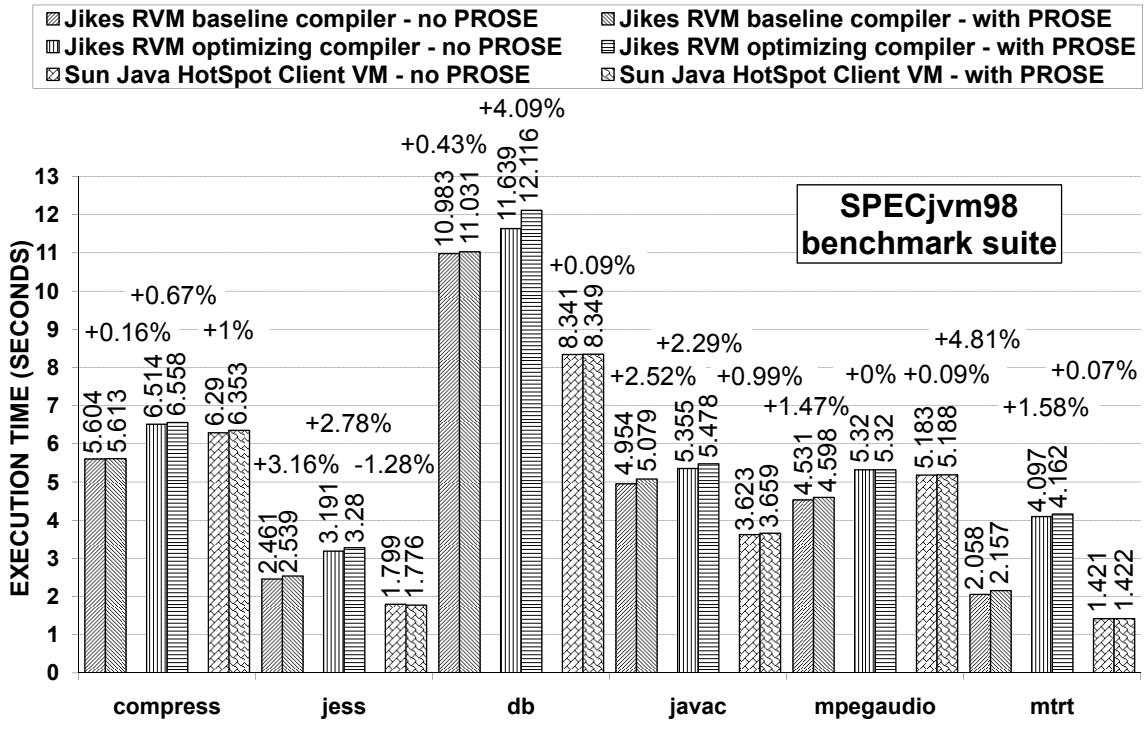

Figure 9. Relative overhead of PROSE before method redefinition

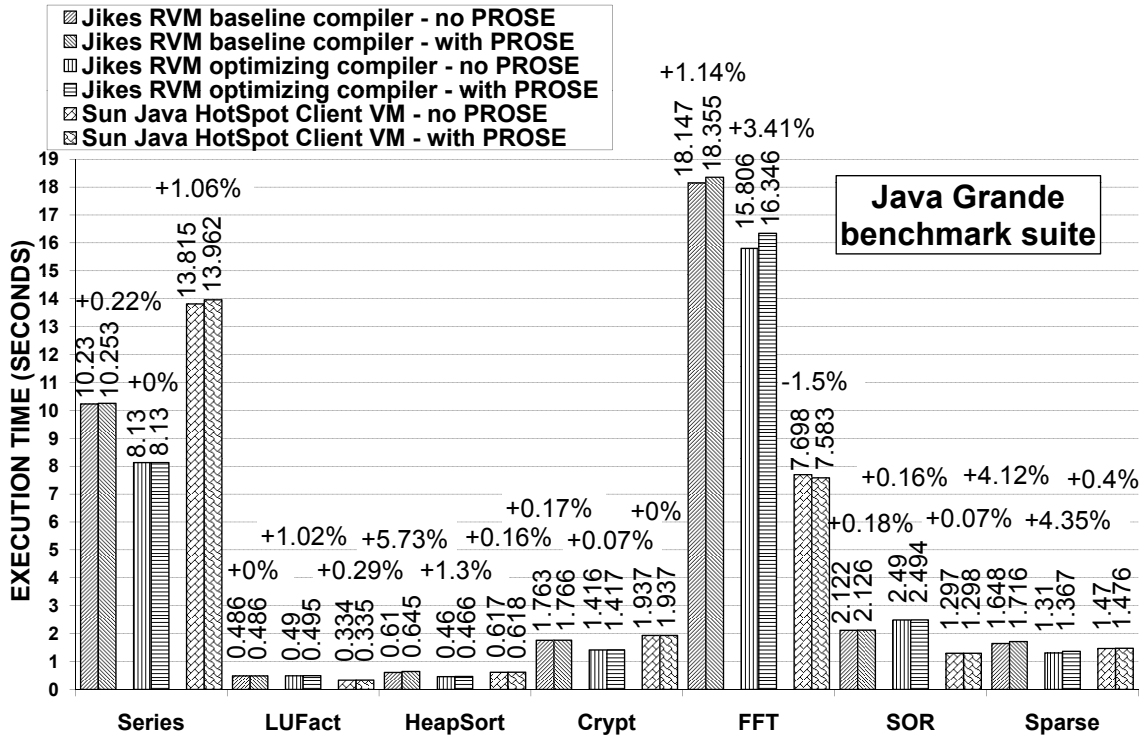

Figure 10. Relative overhead of PROSE before method redefinition

Method type	Arguments	Execution time (ns)					
		zero `proceed` avg / std	one `proceed` avg / std	two `proceed` avg / std	three `proceed` avg / std	four `proceed` avg / std	five `proceed` avg / std
PROSE/ Jikes RVM - Optimizing compiler							
invokevirtual	()	5.217 0.53%	5.228 1.44%	15.746 1.83%	19.113 6.48%	24.467 0.9%	30.233 0.52%
sync invokevirtual	()	64.703 0.68%	68.260 0.68%	73.474 2.93%	76.733 5.73%	78.422 2.38%	82.289 0.94%
invokestatic	()	4.678 3.93%	5.220 3.57%	15.766 1.58%	19.298 2.22%	25.116 1.22%	32.630 0.77%
invokevirtual	(Object,Object)	5.963 0.61%	6.663 3.19%	16.357 3.14%	20.34 6.74%	25.169 1.21%	31.575 1.19%
sync invokevirtual	(Object,Object)	65.889 0.53%	67.061 0.5%	72.671 2.97%	75.754 5.19%	78.147 0.51%	82.329 0.44%
invokestatic	(Object,Object)	4.684 3.81%	5.262 4.63%	17.469 0.85%	21.963 6.16%	27.459 1.71%	34.147 1.85%
PROSE/ Jikes RVM - Baseline compiler							
invokevirtual	()	21.807 0.78%	25.637 0.76%	85.377 2.71%	111.713 3.55%	142.199 0.76%	168.742 1.01%
sync invokevirtual	()	101.653 0.6%	109.768 0.31%	156.456 0.53%	184.107 2.02%	214.166 5.62%	245.679 0.43%
invokestatic	()	17.741 2.7%	21.170 1.45%	67.961 2.15%	95.828 3.94%	121.297 2.16%	146.189 2.71%
invokevirtual	(Object,Object)	24.540 0.95%	36.048 0.84%	95.998 0.35%	132.249 2.15%	162.311 2.82%	201.086 0.42%
sync invokevirtual	(Object,Object)	105.735 1.1%	112.609 0.34%	166.377 2.04%	204.118 0.63%	239.631 0.47%	273.037 0.61%
invokestatic	(Object,Object)	22.549 1.33%	29.184 2.14%	82.501 2.29%	117.150 4.08%	148.693 7.72%	175.783 1.2%
PROSE/ Sun JVM - Java Hotspot Client VM							
invokevirtual	()	26.220 1.38%	28.435 1.37%	40.981 9.86%	43.815 5.25%	49.934 0.7%	55.275 3.9%
sync invokevirtual	()	41.105 0.77%	41.888 0.59%	49.953 2.95%	54.282 3.29%	59.881 2.17%	66.418 0.72%
invokestatic	()	26.987 1.57%	28.713 1.66%	37.913 0.7%	43.879 1.44%	48.674 0.69%	54.021 0.98%
invokevirtual	(Object,Object)	28.354 2.14%	32.084 0.94%	44.963 2.89%	50.399 1.55%	56.574 0.98%	63.976 0.62%
sync invokevirtual	(Object,Object)	42.755 0.49%	47.006 0.43%	55.099 4.18%	60.818 1.22%	68.177 1.98%	75.944 0.79%
invokestatic	(Object,Object)	25.379 1.09%	27.978 1.1%	41.642 8.74%	45.105 4.88%	52.036 5.04%	59.225 0.9%

Table 2. Microbenchmarks for PROSE

and without compiler optimizations enabled, leading to three pairs of measurements for each constituent benchmark.

The results are shown in Figure 9 and Figure 10. In all cases we observe that the overhead of PROSE is small – on average 2.09% for SPECjvm98 and 1.65% for Java Grande – and well below the standard deviation for these experiments (itself below 7%).

5.3 Performance of Inlined Methods

Next, we measure the overhead of replacing a trivial method (which assigns a single variable value) with a new one which calls the original up to 5 times, incrementing a counter before all `proceed` calls. We look at three different kinds of methods: `public` (i.e. a virtual or instance method), `public synchronized`, and `static` (i.e. a straight function call). We also obtain results for two different method signatures: one with no arguments, and one with two arguments of type `Object`. We perform the experiment on three different JVM configurations (Jikes with and without optimization, and Sun HotSpot). We also precede the measurements with a dummy run to ensure all classes are loaded and compiled.

Table 2 summarizes the results. Figures are averaged over 20 experimental runs of 100,000 method calls each; standard deviation for all results is less than 10%.

For the case where only a single `proceed` call is made, the cost of execution is around 28ns for Sun HotSpot. For comparison, both the original function and the advice without any `proceed` calls take around 26ns, implying that the overhead of virtual method invocation is around 24ns. Baseline Jikes gives similar figures, although slightly better. The message from these measurements is that inlining a single instance of an original method is dramatically more efficient than a nested method call would be.

Optimized Jikes shows less dramatic results except for `synchronized` calls, where the benefits of fewer mutex operations are clear. The compiler is clearly generating efficient machine code to invoke both virtual and static methods.

Table 2 also shows the cost of moving from a single inlined `proceed` call to multiple `proceed`s with simulated subroutines. Once again, the results are most dramatic with the Sun JVM, where we pay a penalty of between 8ns and 14ns to go from a single `proceed` to the subroutine implementation with two `proceed`s. From the rest of the data the cost of further `proceed` calls is between 6ns and 8ns.

Interestingly, the penalty of simulating subroutines with

optimized Jikes is much higher than with the Sun JVM, even though the code as a whole runs much faster. This suggests that there may be room for improvement in the Jikes implementation of jump tables for switch constructs.

5.4 Overhead Comparison with AOP Systems

We now compare PROSE's performance on microbenchmarks with prominent Aspect-Oriented Programming systems which operate at compile-time (AspectJ5 [6], ABC [5]), load-time (AspectJ5 again), and run-time (Steamloom [10]). The procedure is much as before, except that we only show results up to 3 `proceed` statements per advice clause.

Many AOP systems distinguish between replacing a method implementation, and replacing all *invocations* of that method with inlined code to execute the advice. Which of these two approaches results in faster modified code is a subtle question, particularly when the advice code uses `proceed` calls to invoke the original method implementation.

Table 3 shows our results for various systems modifying code at compile-time, load-time, and run-time. For the AOP systems, "call" modifications replace every invocation of the method with the advice, and "method" modifications replace only the method's implementation. PROSE only performs method implementation replacement. AspectJ provides two alternate commands for load-time weaving of aspects; we show results for both (`aj` and `aj5`) here.

When?	Modification type	System / JVM	Execution time (ns)		
			one `proceed`	two `proceed`	three `proceed`
Compile Time	call	AspectJ5 / Sun	50.349	53.086	56.271
		AspectJ5 / Jikes	318.316	332.895	365.053
		ABC / Sun	29.376	31.781	33.432
		ABC / Jikes	38.402	51.942	66.214
	method	AspectJ5 / Sun	37.054	40.278	42.476
		AspectJ5 / Jikes	64.058	79.283	93.935
		ABC / Sun	36.285	38.422	39.402
		ABC / Jikes	64.78	76.375	89.843
Load Time	call	AspectJ5 / Sun (aj5)	307.809	379.293	425.375
		AspectJ5 / Sun (aj)	302.723	349.875	369.687
	method	AspectJ5 / Sun (aj5)	238.633	272.169	333.612
		AspectJ5 / Sun (aj)	173.486	218.398	261.83
Run Time	call	Steamloom / Jikes	927.569	3427.003	3879.797
	method	Steamloom / Jikes	2798.317	3255.978	3861.674
		PROSE/ Sun	28.435	40.981	43.815
		PROSE/ Jikes	25.637	85.377	111.713

Table 3. Microbenchmarks for PROSE, AspectJ5, ABC and Steamloom

As expected, compile-time AOP systems (AspectJ and ABC) produce significantly faster code than load-time ones (AspectJ `aj` and `aj5`). Furthermore, both are significantly faster than Steamloom's run-time implementation.

However, PROSE shows performance for run-time modifications which is comparable with that of AspectJ and ABC at compile-time, 6 to 10 times faster than both of AspectJ's load-time mechanisms, and two orders of magnitude faster than Steamloom using the Sun JVM[1].

Steamloom's overhead is due to the way it deals with `proceed`. Steamloom creates a *closure object* for each method in the application where the advice is to be applied; `proceed` is implemented as calls to that closure object. PROSE's inlining of the original method within the advice code reduces the number of indirect references and is largely responsible for the performance gain.

It is more surprising that PROSE slightly outperforms even compile-time approaches such as AspectJ. PROSE's advantage here is gained by having its generated code bypass the JVM calling conventions even for inlined code.

5.5 Comparison using JAC

We also present comparisons using the JAC application benchmark for AOP systems. The benchmark consists of a set of 8 public methods with different method signatures and simple method body implementation. We run 20 experiments of 100,000 iterations each and average the results. The standard deviation for all these experiments is less than 10%.

Table 4 shows the results of (a) applying an aspect to all 8 methods in the benchmark, and (b) applying the aspect to only one[2]. With some variation, the results are consistent with the microbenchmarks, and show PROSE has run-time performance comparable with compile-time AOP approaches.

5.6 Modification Performance

Our final experiment compares the time taken to apply an aspect to a system at run-time using PROSE with the time taken to apply it at load-time with AspectJ. The method used is `public`, takes a single `int` argument and returns an `int`. The advice contains a simple field operation and calls `proceed` up to three times.

Figure 11 shows the average times for twenty runs. We only show figures for the Sun JVM, since AspectJ does not support load-time modification for Jikes. Standard deviation for these figures is less than 4%.

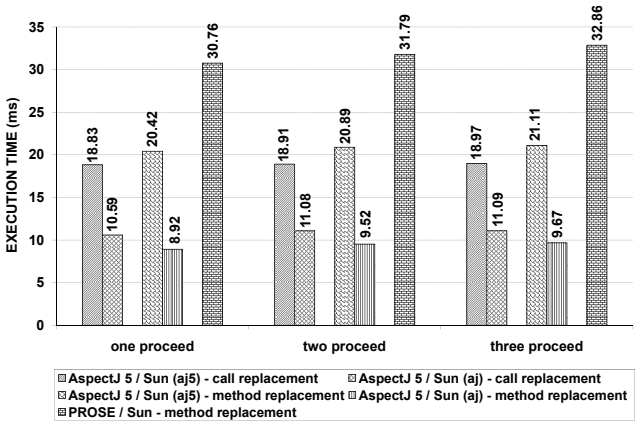

Figure 11. Modification speed of PROSE and AspectJ5

[1]Note: these figures are obtained starting with the baseline Jikes compiler, rather than the optimizing compiler. Steamloom does not function with an optimized Jikes compiler and RVM, but AspectJ and ABC will. Performance for PROSE improves by roughly a factor of 5 when starting with the optimizing Jikes compiler (the precise figures are 5.228ns, 15.746ns, and 19.113ns for 1, 2, and 3 `proceed`s) and we expect similar improvements for the other systems, which will not significantly change our overall conclusions.

[2]We show incomplete results for Steamloom, since its current implementation cannot successfully apply aspects to 8 public methods. We are working with the implementors to resolve this problem.

(a) Modifying all 8 methods

When?	Modification type	System / JVM	Execution time (ms) one proceed	two proceed	three proceed
Compile Time	call	AspectJ5 / Sun	7.3	13.95	16.05
		AspectJ5 / Jikes	20.2	22.1	25.7
		ABC / Sun	3	3	3
		ABC / Jikes	20	21.9	25.55
	method	AspectJ5 / Sun	7.25	13.05	15.05
		AspectJ5 / Jikes	18.05	18.6	19.05
		ABC / Sun	4	4	4
		ABC / Jikes	13.75	14.6	15.65
Load Time	call	AspectJ5 / Sun (aj5)	148	221.85	296.9
		AspectJ5 / Sun (aj)	152.05	227.3	302.35
	method	AspectJ5 / Sun (aj5)	137.8	207.9	278.3
		AspectJ5 / Sun (aj)	141.75	213.45	284.75
Run Time	call	Steamloom / Jikes	-	-	-
	method	Steamloom / Jikes	-	-	-
		PROSE/ Sun	3	10.35	12.2
		PROSE/ Jikes	21.8	24.1	26.15

(b) Redefining a single method

When?	Modification type	System / JVM	Execution time (ms) one proceed	two proceed	three proceed
Compile Time	call	AspectJ5 / Sun	2.05	3.95	4
		AspectJ5 / Jikes	9.9	10.8	10.85
		ABC / Sun	2	2	2
		ABC / Jikes	9.15	9.25	9.6
	method	AspectJ5 / Sun	2	3.95	4
		AspectJ5 / Jikes	9.85	9.9	9.95
		ABC / Sun	2	2	2
		ABC / Jikes	8.05	8.15	8.35
Load Time	call	AspectJ5 / Sun (aj5)	15	20.05	25.85
		AspectJ5 / Sun (aj)	15.3	21	26.05
	method	AspectJ5 / Sun (aj5)	13.95	18.05	23
		AspectJ5 / Sun (aj)	14	18.95	23.8
Run Time	call	Steamloom / Jikes	1225.95	2312.35	3394.05
	method	Steamloom / Jikes	1226.1	2307.25	3390.95
		PROSE/ Sun	2	2	2
		PROSE/ Jikes	9.25	10.85	11.95

Table 4. Measurements with PROSE, AspectJ5, ABC and Steamloom

We see that PROSE takes about 31ms to find the method to be replaced, generate the bytecode including the inlined original method, and install the new code in the VM, and furthermore incurs an overhead of about a millisecond for each extra `proceed`. This is about 55% slower than AspectJ's load-time operation, but we feel that given PROSE's superior performance and ability to modify the application at run-time, this penalty is acceptable.

6. RELATED WORK

PROSE borrows its notation from Aspect-Oriented Programming, but applies the concepts to the problem of modifying running applications instead of introducing crosscutting changes as part of the software development process. As such, and unlike current AOP efforts, PROSE should not be seen as a software development tool but rather as a run-time tool that can be used for a wide range of purposes: from instrumentation to providing high availability.

We have already introduced the major AOP systems in Section 5. The best-known compile-time AOP system is *AspectJ* [6, 16], which implements `proceed` calls in advice using closure objects. A closure is created for each point where the advice applies, and `proceed` calls invoke the closure. AspectJ will inline code in the absence of `proceed` calls. Recently AspectJ's compile-time system has been merged with the load-time functionality of AspectWerkz [7].

The *AspectBench Compiler (abc)* [5, 8] uses a more efficient approach whereby the `proceed` code from all modified methods of a class is moved to a static `proceed` method in the class. Integer indices are used at run-time to identify the class from which advice was called, and to invoke the correct original method.

Steamloom [10, 15] performs run-time code replacement as an extension to the IBM Jikes RVM. Steamloom uses a sophisticated closure-passing scheme to provide considerable flexibility on where an advice can be applied, but has a number of restrictions on advice implementation: it cannot exploit the optimizing Jikes compiler, does not at present allow composition of aspects, and cannot apply aspects to methods that throw exceptions. It also uses its own programming language to describe aspects.

Run-time replacement of code or application subsystems has also been explored in a variety of contexts.

The Erlang language [4] was explicitly designed to facilitate code replacement at run-time. Erlang is a strict functional language, with all program state consequently passed on the stack (through tail calls in the case of message-handling loops). Individual functions can therefore be replaced at the point where they are next called. The high-level ability to replace functionality that cuts across an Erlang application is left to the system designer.

An idea similar in spirit to PROSE but with rather different techniques and context was recently proposed as a way to dynamically instrument an operating system kernel [22]. The approach adds instrumentation to code blocks by copying them to an instruction cache, adding instrumentation on the way, before executing. Jumps and traps (the standard techniques for inserting instrumentation) are avoided by having the instrumentation directly inlined into the original code. Unlike PROSE, this technique can only add code, not replace the functionality. It is also unclear how such modifications could, in the more general-purpose case, handle variable arguments and exceptions.

Another system for dynamic instrumentation is Pin [20]. When attached to an application, Pin uses a JIT compiler that takes as input native code and partly recompiles the code plus the added instrumentation. Pin regains control over the application with every branch and generates the necessary code from the original code plus the instrumentation. Compared with PROSE, and like many similar dynamic instrumentation systems, Pin can only add, not replace, code. However, it does operate on native code in real time.

Alternative techniques have been proposed to safely upgrade parts of a distributed application by exploiting the (hopefully) inherent fault-tolerance of such a system to perform micro-reboots [11] with new code. We see techniques like these as largely complementary to our own: we provide a way to apply changes across the systems as a whole, and can do it without shutting down an application VM if needed.

Existing research on mobile, pervasive and autonomic computing has proposed several approaches to the problem of application recovery in case of system failures caused by the operating system's device drivers [26], software failures in large-scale Internet systems [11], or database connections in multi-tier applications [9, 23]. Several techniques have been used (e.g., shadow drivers, micro-rebooting of partial system

components) to adapt systems and allow applications to continue executing despite software failures.

Finally, we recognize that in a distributed context, PROSE is only addressing only part of the wider problem of evolving a large application. For example, we do not address the long-term versioning and operations issues addressed in [1, 2].

7. DISCUSSION

Modifying a running application on-the-fly is, of course, a potentially dangerous operation, and the power of PROSE in replacing collections of methods at run-time comes with associated caveats about unrestrained usage. Several concerns about "safety" (in the broad sense) come to mind; in this section we attempt to address them in turn.

Firstly, the ability of PROSE to change arbitrary program functionality via method replacement says nothing about whether such changes will result in a program that will crash or enter an inconsistent state. There is a tension between the safety of such operations and their scope in changing the behavior of the application, and it is unclear whether an "inherently safe" mechanism could implement useful changes at all.

Our response to this dilemma is not to limit PROSE's power, but instead to provide abstraction mechanisms to make it easier for program maintainers to do the right thing. In particular, the ability to express (and perform) crosscutting changes in one go, and furthermore the bundling of multiple crosscuts into a single unit (the aspect) to be applied atomically, make it easier to transition an application from one state to a new one which is hopefully desirable.

Nevertheless, PROSE can be used in ways that are not as dramatic as radically changing how an application behaves. For instance, an advice can be used to test if a modification actually works. The advice runs both the modification and the original code through `proceed`. It then compares the results (which could also be compared in terms of performance) and keeps a log. The advice always returns the value provided by the original method. The application has not changed in principle but developers have now a useful facility for testing new code on the running system. It is relatively easy to develop a tool above PROSE that only permits such testing extensions, and we are interested in further pursuing the possibilities of building software engineering tools above PROSE.

A second concern is that of "feature interaction": the unexpected results of applying two, individually unproblematic, changes to the same application. This problem is not unique to aspect-oriented program modifications, indeed it is well known to anyone who has engineered a large system over an extended period.

As before, our philosophy is not to attempt to rule out the problem by limiting the expressivity of PROSE, but instead to give users better tools to manage the problem themselves. Hence PROSE provides a mechanism for keeping track of the combined semantics of multiple aspects by means of the ordering arguments. Aspects can be reordered and reapplied atomically, should the need arise, and individual aspects withdrawn from the system at any time.

A third potential problem with PROSE is source code divergence: over time, as more and more changes are applied, the running system diverges from the source code available to the programmer. This is of course a problem with any patching system, but whereas patches are typically localized modifications, PROSE changes affect many areas of an application (and indeed this is one of their principle advantages). While a programmer would presumably have at her disposal the complete set of applied aspects as well as the original source code, this does not necessarily make the job of understanding the program as a whole any easier.

We see this is a legitimate question of appropriate usage and methodology. We can view uses of PROSE as falling between two ideal poles: applying new functionality which is completely orthogonal to the correctness of the program (such as tracing information), and changing the behavior of the running system completely (by replacing whole subsystems). The source code divergence problem arises most when the application contains many applied changes which fall somewhere in the middle.

Since PROSE is based on controlled replacement of methods, one promising methodology for prolonged system evolution is as follows: firstly, short-term, urgent changes to a system are implemented as aspects with `proceed` calls to the original methods. These prototype aspects can be replaced later with aspects that contain the entire new logic and consequently have no need for `proceed` calls. Such "finished" aspects also enable synchronization with the source code. Finally, scheduled downtime or microreboots at safe times can be used to replace the system with a fresh build from source.

We have liberally borrowed concepts from Aspect-Oriented Programming, and some of the above criticisms have long been levelled at AOP in general. Our position is that AOP is not a panacea, and indeed in adopting our dynamic run-time approach we are explicitly *not* advocating AOP as a methodology for designing and building systems. Instead, we are arguing in this paper that AOP's concepts are a useful way to get purchase on the problem of modifying an existing (and, in our case, running) system.

8. CONCLUSION

PROSE demonstrates the feasibility of systematically modifying a running Java application without shutting it down, by building a powerful facility above the existing low-level code hot-swapping features of Java virtual machines.

Crosscuts allow changes to be applied to areas of program functionality that are not localized in particular components, objects, or other program areas. Furthermore, aspects allow related and interdependent crosscuts to be gathered together and applied (or withdrawn) as a unit. PROSE also provides controlled composition of aspects via a well-defined order on modifications.

Finally, PROSE shows how aggressive inlining can be generic enough to support different designs of Java virtual machine, yet provide performance at run-time considerably superior to the existing load-time AOP-based systems, and competitive with the best compile-time AOP systems.

The source code for PROSE, together with the code for all the benchmarks in this paper, will be available in due course on the project web site at `http://prose.ethz.ch/`.

9. ACKNOWLEDGMENTS

We would like to thank Thomas Gross for his valuable comments and advice on how to improve this paper.

References

[1] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Scheduling and Simulation: How to Upgrade Distributed Systems. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, Hawai, May 2003.

[2] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular Software Upgrades for Distributed Systems. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'06)*, Nantes, France, July 2006.

[3] Bowen Alpern, C. R. Attanasio, John J. Barton, Michael G. Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn-Hummel, Derek Lieber, Vassily Litvinov, Mark F. Mergen, Ton Ngo, James R. Russell, Vivek Sarkar, Mauricio J. Serrano, Janice C. Shepherd, Stephen E. Smith, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeno virtual machine. In *IBM System Journal, 39(1)*, 2000.

[4] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, 2nd edition, 1996.

[5] AspectBench Compiler (abc) website. `http://abc.comlab.ox.ac.uk/`.

[6] AspectJ website. `http://www.eclipse.org/aspectj/`.

[7] AspectWerkz website. `http://aspectwerkz.codehaus.org/`.

[8] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, Chicago, USA, 2005.

[9] Roger Barga, David Lomet, and Gerhard Weikum. Recovery Guarantees for General Multi-Tier Applications. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'02)*, San Jose, California, 2002.

[10] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual Machine Support for Dynamic Join Points. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD'04), pages 83-92*, Lancaster, UK, 2004.

[11] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot - A Technique for Cheap Recovery. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, California, USA, 2004.

[12] Mikhail Dmitriev. Application of the HotSwap Technology to Advanced Profiling. In *Proceedings of the ECOOP'02 Workshop on Unanticipated Software Evolution*, 2002.

[13] ECMA International. *Standard ECMA-335, Common Language Infrastructure (CLI)*, 4 edition, June 2006.

[14] Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plug-Ins*. Addison-Wesley, 2004.

[15] Michael Haupt. *Virtual Machine Support for Aspect-Oriented Programming Languages*. PhD thesis, Darmstadt University of Technology, 2006.

[16] Erik Hilsdale and Jim Hugunin. Advice Weaving in AspectJ. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, Lancaster, UK, 2004.

[17] JAC website. `http://jac.objectweb.org`.

[18] Java Grande Forum. Java Grande benchmark suite. `http://www.javagrande.org`, 2006.

[19] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97), vol. 1241 of LNCS, pag. 220-242, Springer-Verlag*, Jyvaskyla, Finland, 1997.

[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, Chicago, Illinois, USA, 2005.

[21] Angela Nicoara and Gustavo Alonso. Dynamic AOP with PROSE. In *Proceedings of the International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA'05) in conjunction with the 17th Conference on Advanced Information Systems Engineering (CAiSE'05)*, Porto, Portugal, June 2005.

[22] Marek Olszewski, Keir Mierle, Adam Czajkowski, and Angela Demke Brown. JIT Instrumentation - A Novel Approach To Dynamically Instrument Operating Systems. In *Proceedings of the ACM EuroSys 2007 Conference (EuroSys'07)*, Lisbon, Portugal, 2007.

[23] Phoenix Project, Database Group, Microsoft Research. `http://www.research.microsoft.com/db/phoenix/`.

[24] Spec - Standard Performance Evaluation Corporation. `http://www.spec.org/osg/jvm98/`.

[25] Sun Microsytems. Java 2 Platform, Standard Edition (J2SE). `http://java.sun.com/j2se`, 2006.

[26] Michael Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering Device Drivers. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, California, USA, 2004.

[27] TPC-W Benchmark Specification. `http://www.tpc.org/tpcw/specs.asp`.