Overcoming JVM HotSwap Constraints via Binary Rewriting

Dong Kwan Kim and Eli Tilevich

Dept. of Computer Science
Virginia Tech
Blacksburg, VA 24061
{ikek70, tilevich}@cs.vt.edu

Abstract

Java HotSpot VM provides a facility for replacing classes at runtime called HotSwap. One design property of HotSwap is that the signature of a replaced class must remain the same between different versions, which significantly constrains the programmer modifying a class to be replaced. Specifically, the programmer is precluded from adding new methods, constructors, or fields, or even changing the signatures of existing methods or fields. This paper presents a novel approach that overcomes these constraints of HotSwap by using *binary refactoring*, a technique that rewrites the binary representation of a program without affecting its functionality. A series of micro and macro benchmarks we conducted demonstrate that the approach is extensible and efficient. In particular, the new binary refactoring technique, which enables the approach, produces highly-efficient refactored application versions, outperforming a widely-used prior technique by as much as an order of magnitude. These initial results indicate that by rewriting the bytecode of a swapped program, one can effectively overcome constraints of HotSwap with minimal performance overhead.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming—program modification, program transformation; D.2.3 [*Software Engineering*]: Coding Tools and Techniques—object-oriented programming

General Terms Languages, Experimentation

Keywords HotSwap, Binary Refactoring, Virtual Superclass, JVM Languages,

1. Introduction

Among the advanced facilities provided by the JVM is the HotSwap API [1], which makes it possible to replace loaded classes in a running application. However, the signature of a replaced class must remain the same, and only method bodies could change. Thus, adding a new method, field, or constructor, or even changing the signature of an existing method or field will render a class invalid for HotSwap. While this design facet of HotSwap simplifies its implementation, as there is no need to update the state of objects created using an older version of a class, the programmer modifying the swapped classes is significantly constrained.

To overcome these constraints of HotSwap, this paper presents a novel approach based on bytecode rewriting and code generation, enabling the programmer to change swapped classes as required and still use HotSwap to replace the changed functionality in a running JVM. The approach uses *Binary Refactoring*, a technique we introduced earlier [15] that uses automated tools to change the binary representation of a program without affecting its functionality.

The approach uses binary refactoring to introduce indirect referencing (i.e., a Proxy pattern) to a target application. The indirection enables updated classes to comply with the restrictions of HotSwap, even if unsupported changes are present. While proxies as a mechanism to enable dynamic updates of Java applications have been proposed by Orso *et al.* [12], their approach suffers from two limitations. First, similarly to HotSwap, they disallow changes to class signatures. Second, their implementation of the Proxy pattern incurs significant performance overhead on the rewritten programs. By contrast, our approach supports changes to class signatures and implements the Proxy pattern in a novel way that offers an order magnitude speed-up on average.

The rest of this paper is structured as follows. Section 2 describes our approach that overcomes limitations of HotSwap. Section 3 details our performance results through micro and macro benchmarks as well as a case study. Section 4 compares our approach to existing state of the art.

Targets	Updates
Method	M1: Addition of a new method
	M2: Removal of an existing method
	M3: Addition of formal arguments of a method
	M4: Removal of formal arguments of a method
	M5: Change to the return type of a method
	M6: Change to method modifiers
Field	F1: Addition of a new field
	F2: Removal of an existing field
	F3: Change to the type of a field
	F4: Change to field modifier

Table 1. Limitations of HotSwap. The addressed limitations are shaded.

Section 5 summarizes our contributions and outlines future work directions.

2. Background and Approach Overview

Next we detail the constraints imposed by the standard JVM HotSwap as well as how they can be overcome using binary rewriting.

2.1 Limitations of HotSwap

Table 1 summarizes the constraints of the JVM HotSwap. Whenever programmers try to perform these updates listed in the second column using the JVM HotSwap, the JVM throws `java.lang.UnsupportedOperationException`. In short, HotSwap disallows any changes to the signature of a class. The swapped class has to contain the same set of methods and fields as the currently deployed version, and all the allowed changes must be within method bodies.

HotSwap constraints are particularly restrictive, because one cannot assume a one-to-one correspondence between source files and their compiled binary class versions. To further clarify this point, consider how Java inner classes are commonly translated. To allow inner classes to access non-public members of their enclosing classes, the Java compiler silently adds `synthetic` access methods to the enclosing classes. Thus, the programmer will be completely unaware that an unrelated change in a method body of an inner class has caused the compiler to add a method to another class. Further, HotSwap will unexpectedly fail trying to update such an enclosing class due to the restriction on adding new methods.

2.2 Virtual Superclass Binary Refactoring

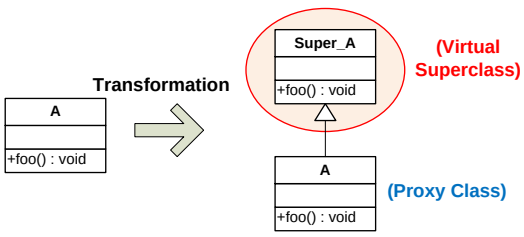

Figure 1. Virtual superclass binary refactoring

To address limitations of HotSwap described in Section 2.1, our approach changes direct references into proxy references. A commonly-used implementation of indirect referencing is described by a binary refactoring that we call *Virtual Interface*.[1] The Virtual Interface binary refactoring transforms a class into proxy, interface, and implementation classes. The bytecode instrumentor does not change the client part of the target version, but makes it refer to the proxy class in the refactored version. However, the virtual interface style proxy indirection can incur as much as 30% performance overhead, according to our measurements [14].

Thus, to alleviate this performance overhead, we created a new technique for introducing indirect referencing that we call *Virtual Superclass*. Similar to Virtual Interface, Virtual Superclass introduces an indirection to a target class, but it does so without incurring the often prohibitive overhead of Virtual Interface. Each application class initially loaded into the JVM, first goes through the Virtual Superclass refactoring at the bytecode level; Figure 1 describes the refactoring transformations involved. This refactoring changes class `A` to extend a *virtual* superclass `Super_A`. In other words, the virtual superclass is inserted into the class's inheritance hierarchy. The original class `A` becomes a proxy, and the virtual superclass contains all the original functionality, including method bodies and fields.

The performance efficiency of Virtual Superclass stems from the sophisticated optimization capabilities of modern JVMs. Among their advanced optimization facilities is the ability to inline delegating method calls, if the delegation does not involve dynamic dispatch. In Figure 1, the call to `super.foo` is translated into the `invokespecial` bytecode instruction, reserved for invoking constructors and methods in superclasses. Modern JVMs can effectively inline this call, completely eliminating any indirection overhead in most cases. By contrast, Virtual Interface uses the `invokeinterface` instruction in its implementation. While the performance of `invokeinterface` has been improved significantly in modern JVMs [3], this instruction implements a form of dynamic method dispatch, and as such cannot be safely inlined. Thus, it is the JVM method invocation instructions used that explain the performance differences of these two implementations of the Proxy Design pattern.

2.3 Translating Updates for HotSwap Conformance

Our approach still uses the standard HotSwap to replace classes in a running JVM, but all the deployed classes are first enhanced with additional capabilities at the bytecode level. It is these capabilities that enable a full-range of changes to be made to the replaced classes, without violating constraints of the HotSwap API.

[1] The adjective *virtual* emphasizes the fact that the introduced interface is not seen by the client program and is only used as an implementation artifact. The client code never accesses the introduced "virtual" interface directly.

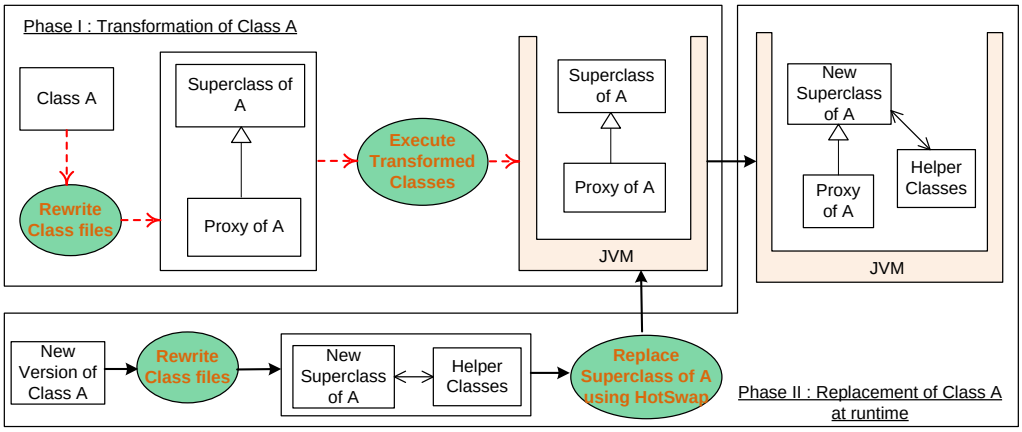

Figure 2. Supporting a full range of dynamic updates using HotSwap.

At the first phase illustrated in Figure 2, the updating system refactors all the loaded classes at the bytecode level and generates their corresponding virtual superclasses. The virtual superclasses have actual methods implementing application-specific logic and are swapped by the updating system. Therefore, the output of the bytecode rewriting is updateable software, which is structurally different from the original version and can be deployed on a virtual machine. When the initial program is changed, the programmer gives the changed classes as input to the updating system, which refactors them into virtual superclasses and special helper classes. HotSwap can then replace older class versions of virtual superclasses with newer versions, as they have the same schema. Helper classes make the updates conform to the HotSwap API when new methods or fields are added. The new members are added to helper classes, so that the signatures of virtual superclasses remain the same.

Our prototype uses the Javassist library[8] and consists of a class differencing module and code generators for proxy, virtual superclass, and helper classes. The differencing algorithm operates at the bytecode level, and its output parameterizes the code generators and the bytecode rewriter. The rewriter translates newly-added methods, constructors, and fields to helper classes as follows:

Adding new methods/constructors: A special `invoke` method is added to all the instrumented classes as a facility to invoke newly-added methods without changing the updated class's signature. Each new method is translated into a method in a helper class, whose invocation logic is added to the body of the `invoke` method. Each call site of a newly added method becomes a call to `invoke`, with the added method name as the first argument.

Figure 3 shows an example of adding a new method; the newer version of `A` has a new method `bar`. The first and second columns in Figure 3 illustrate class diagrams representing classes and their relationships at the source code and the corresponding bytecode, respectively. The special helper class `HelperClass` contains the new method `bar` and each proxy class contains the `invoke` method. Each invocation of `bar` is translated to invoke `invoke` instead.

Each new constructor is translated into an invocation of a "do-nothing" constructor and a special initialization method that contains the added constructor's logic.

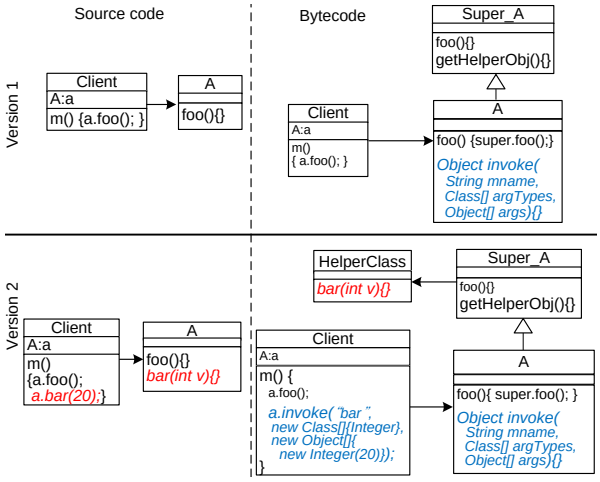

Figure 3. Adding a new method

Adding new fields: New fields are translated according to two approaches, one optimized for performance, while the other for space. The first uses a separate helper class for the new fields whenever a class is replaced with a newer version. The second uses a single class that contains a mapping data structure that represents all the added fields for all classes.

Object state update: One complication of using HotSwap for updating running applications is that it can only update classes–HotSwap has not facilities for upgrading objects created from an older version of a class to a newer version. In dynamic update systems, this operation is called *Object State Update*. Our approach also can efficiently transfer state between old and new objects, enabling instances of different versions of a class to coexist in the running application. Our system updates the state between old and new helper objects for new fields, based on their respective version numbers.

Specifically, the update system checks if the version of a helper object is older than the latest version. If so, a special helper object is instantiated for the newly added state (i.e., extra fields). The values of the fields in the older helper object then are copied to the corresponding fields in the newer helper object.

3. Experimental Results

We show the efficiency and extensibility of our approach through benchmarks and a case study updating a server-side RMI application.

3.1 Performance Evaluation

To measure the performance overhead of the Virtual Superclass binary refactoring, we conducted micro and macro benchmarks. All the measurements were performed on a workstation with an Intel Pentium 4 (3.6GHz) processor, 1GB RAM, running Ubuntu Linux 7.10 (Gutsy Gibbon), JDK version 1.5.0_14.

Figure 4 shows the overhead of indirecting a single method invocation. The cost of indirection depends on the amount of computation performed by the indirected method. In this benchmark, the indirected method performed two, four, and eight multiplications, increments, and test operations. Each invocation is repeated $1 * 10^9$ times. The maximum overhead of less than 2% makes this refactoring applicable for introducing indirection to most performance-sensitive applications.

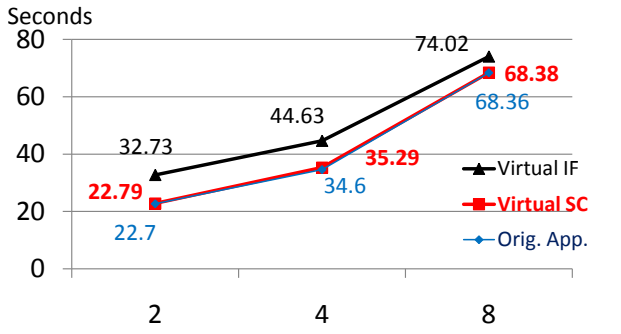

Figure 4. Overhead of binary refactoring microbenchmark. (**SC**–SuperClass; **IF**–Interface)

To assess the overhead imposed by the virtual superclass indirection in more realistic programs, we used five different programs from the SpecJVM98 benchmark [2]. The suite contains a set of test programs, with some of them being real applications, such as _201_compress and _202_jess. Figure 5 shows the overhead of bytecode rewriting using the virtual superclass for the SpecJVM98 test programs. Similar to the numbers obtained for the microbenchmark, the indirection overhead for all benchmark programs never exceeds 2%, as shown in Figure 5.

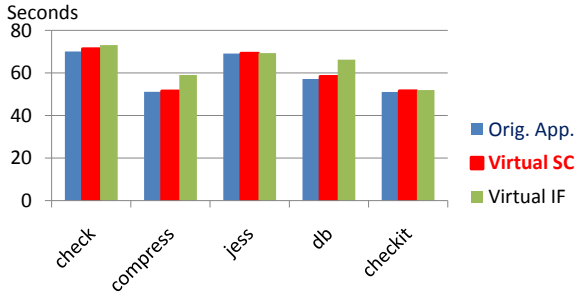

Figure 5. Overhead of binary refactorings on SpecJVM98. (**SC**–SuperClass; **IF**–Interface)

Changes	Implementations
Add methods	Add `transfer(float, Account)`
Change method signatures	deposit(float) → deposit(*boolean*, float) withdraw(float) → withdraw(*boolean*, float) transfer(float, Account) → transfer(*boolean*, float, Account)
Add fields	Add a new field `coowner` and getter/setter
Call sites	*.transfer(..) → *.invoke("transfer", ...)

Table 2. Dynamic updates to the Banking program

3.2 Case Study: Updating an RMI-based Application Dynamically

As a case study, we used a Remote Method Invocation (RMI)-based program as an example of a long-running server application that can benefit from being updated at runtime. The program consists of a remote interface `Account` and its implementation `AccountImpl`. The update cases are illustrated in Table 2, which are adding a method, changing the signature of a method, and adding a new field.

To add a new method, `transfer`, our updating system generates a proxy class with the `invoke` method and a helper class with the `transfer` method. All invocations to `transfer` are replaced with invocations to `invoke` method of the proxy class.

To modify the signatures of remote methods, we added a new argument `access` that checks the user's permission.

To add a new field, `coowner` which holds the value of a co-owner's name, our system generates a special helper class containing the new field. The virtual superclass will return the helper class when any object tries to access the field.

4. Related Work

Due to space constraints, we will focus only on closely related research on dynamic updates for Java applications. Existing state of the art includes program transformation, custom virtual machines or runtime libraries, and special programming models.

Our approach is based on refactoring transformations, which change the structure of a program without affecting its functionality. Orso *et al.*'s technique [12] also transforms the code to enable its dynamic updates. The Virtual Inter-

face refactors each class into wrapper, implementation, interface, and state classes. As discussed above, Virtual Interface incurs significant performance overhead, making it inapplicable for performance-sensitive applications. In addition, their technique does not support changes to the signatures of swapped classes. Bialek *et al.*'s system [4] also rewrites the updated software at the source or bytecode levels to enable its dynamic replacement, but it does not use HotSwap, and thus, does not need to overcome its constraints.

Several approaches [9, 11, 13] have introduced custom virtual machines to support dynamic updates of Java applications. These approaches, however, require installing a custom JVM, which may have limited functionality and interoperability.

Some approaches [10, 7, 16, 5] introduce new languages features, middleware systems, or require that software developers abide by specific component models or programming rules. Warth *et al.* presents Expanders [16], a programming language construct that allows adding new methods, fields, and interfaces. Expanders enable the programmer to express new methods and fields to be added to an existing program.

Bierman *et al.*'s UpgradeJ [5] is a Java-like language for dynamically upgrading classes with guaranteed type-safety. UpgradeJ allows for safe co-existence of classes with different versions by including an explicit version number as part of a class signature. UpgradeJ supports adding methods and fields, changing type hierarchies, and modifying method bodies. An upcoming implementation of UpgradeJ will also use HotSwap as the underlying technology.

5. Future Work and Conclusions

As future work, we plan to apply our approach to facilitate the updates of programs written in JVM-based, high-productivity languages such as X10 [6]. High performance computing applications are often long-running, and their parts may have to be updated without stopping the JVM. The efficiency of our approach makes it promising for this domain.

We presented a new binary rewriting approach that can overcome constraints of HotSwap. The micro and macro benchmarks we conducted demonstrate that our approach is viable and does not incur unreasonable performance overhead. Furthermore, as dynamic reconfiguration and maintenance have become an indispensable part of modern software system evolution, the techniques that we introduced can benefit the broader maintenance community, and our approach can become a valuable tool in the toolset of programmers using JVM-based languages.

References

[1] Java HotSwap, `http://java.sun.com/j2se/1.4.2/docs/guide/jpda/enhancements.html`.

[2] Specjvm98 benchmarks, `http://www.spec.org/jvm98/`.

[3] B. Alpern, A. Cocchi, S. Fink, and D. Grove. Efficient implementation of Java interfaces: Invokeinterface considered harmless. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 108–124, 2001.

[4] R. P. Bialek. Dynamic updates of existing Java applications. *Ph.D. Thesis, the University of Copenhagen*, pages 1–216, June 2006.

[5] G. Bierman, M. Parkinson, and J. Nob. UpgradeJ: Incremental typechecking for class upgrades. In *European Conference on Object-Oriented Programming (ECOOP)*, July 2008.

[6] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An object oriented aproach to non-uniform cluster computing. *Conference on Object Oriented Programming Systems Languages and Applications*, June 2005.

[7] X. Chen. Extending RMI to support dynamic reconfiguration of distributed systems. *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 401–408, 2002.

[8] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. *Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE'03)*, pages 364–376, 2003.

[9] B. Gharaibeh, D. Dig, T. N. Nguyen, and J. M. Chang. dReAM: Dynamic refactoring-aware automated migration of Java online applications. *Technical Report, Iowa State University*, August 2007.

[10] Y.-F. Lee and R.-C. Chang. Java-based component framework for dynamic reconfiguration. *IEE Proceedings - Software*, 152(3):110–118, June 2005.

[11] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. *Proceedings of the 14th European Conference on Object-Oriented Programming*, 1850:337–361, June 2000.

[12] A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating of Java software. *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, October 2002.

[13] T. Ritzau and J. Andersson. Dynamic deployment of Java applications. In *Java for Embedded Systems Workshop*, London, May 2000.

[14] E. Tilevich and Y. Smaragdakis. J-Orchestra: Enhancing Java programs with distribution capabilities. *ACM Transactions on Software Engineering and Methodology*. Accepted for publication.

[15] E. Tilevich and Y. Smaragdakis. Binary refactoring: Improving code behind the scenes. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 264–273, May 2005.

[16] A. Warth, M. Stanojević, and T. Millstein. Statically scoped object adaptation with Expanders. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 37–56, 2006.