Magic with Dynamo – Flexible Cross-Component Linking for Java with Invokedynamic*

Kamil Jezek[1] and Jens Dietrich[2]

1 **NTIS – New Technologies for the Information Society**
Faculty of Applied Sciences, University of West Bohemia
Pilsen, Czech Republic
kjezek@kiv.zcu.cz

2 **School of Engineering and Advanced Technology, Massey University**
Palmerston North, New Zealand
j.b.dietrich@massey.ac.nz

Abstract

Modern software systems are not built from scratch. They use functionality provided by libraries. These libraries evolve and often upgrades are deployed without the systems being recompiled. In Java, this process is particularly error-prone due to the mismatch between source and binary compatibility, and the lack of API stability in many popular libraries. We propose a novel approach to mitigate this problem based on the use of `invokedynamic` instructions for cross-component method invocations. The dispatch mechanism of `invokedynamic` is used to provide on-the-fly signature adaptation. We show how this idea can be used to construct a Java compiler that produces more resilient bytecode. We present the dynamo compiler, a proof-of-concept implemented as a `javac` post compiler. We evaluate our approach using several benchmark examples and two case studies showing how the dynamo compiler can prevent certain types of linkage and stack overflow errors that have been observed in real-world systems.

1998 ACM Subject Classification D.1.5 Object-oriented Programming D.2.13 Reusable Software D.3.4 Processors

Keywords and phrases Java, compilation, linking, binary compatibility, invokedynamic

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.12

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.2.1.5`

1 Introduction

Java and similar languages support dynamic linking to enable programs to use libraries that have been compiled separately. This makes it possible to decouple the lifecycles of a system and the libraries it uses, and to deploy new versions of libraries without reinstalling the entire system. This is particularly important for server-based systems with high availability requirements, such as services running in "24/7" mode and systems with service level agreements (SLAs). For this to work, library evolution must follow certain compatibility rules. In particular, the APIs used by the system or other libraries have to remain stable. Unfortunately, APIs do change when libraries evolve [13, 6, 11]. In Java programs, this can result in linkage errors indicating that references to library code cannot be resolved.

* This work was partially supported by Oracle Labs, Australia.

30th European Conference on Object-Oriented Programming (ECOOP 2016).
Editors: Shriram Krishnamurthi and Benjamin S. Lerner; Article No. 12; pp. 12:1–12:25

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

For instance, if the signature of a library method changes, attempts to link and run a client program using this method generate a `NoSuchMethodError`. Some other language and execution platform combinations such as C# / CLR exhibit similar behaviour.

The relevant part of the signature used to resolve method references is called the descriptor [26, sect 4.3.3]. The descriptor consists of the parameter and return types without generic type parameters. Changes to descriptors cause linkage errors. This leads to a behaviour that is very different from the compiler that reasons about the type hierarchy, associations between primitive and wrapper types, and narrowing and widening rules. In other terms, there are different types of compatibility [8]: *source compatibility* is used by the compiler when a system is compiled against a set of libraries, while *binary compatibility* is used when a system is linked against libraries that may have been compiled separately. The notion of binary compatibility is explicitly defined in the Java Language Specification with respect to linking [22, sect 13].

There are a number of empirical studies all indicating that this leads to problems [13, 6, 11, 32, 33]. It turns out that binary compatibility issues are surprisingly common when libraries evolve, and the majority of developers lack understanding of these issues [12].

There are several possible approaches to tackle this problem. First, meta-object protocols and patterns could be used. For instance, in the presence of protocols like Smalltalks `doesNotUnderstand` [21] or Ruby's `method_missing` [18], adapters can be easily integrated into library code. But this requires that the respective classes override `doesNotUnderstand`. And anyway, such a protocol is not available in Java. The second option is to change the runtime (i.e. the VM): either by instrumenting code that is being loaded or by changing the linking process in the virtual machine itself. This is possible, but expensive and invasive - the runtimes must be configured accordingly, and the instrumentation imposes a performance penalty. Moreover, the same code is sometimes accessed from different contexts (other libraries or applications), with different lifecycle dependencies on the library containing the code that is being adapted. Instrumenting library code does not support such scenarios. The third option is to access objects instantiating library classes via proxies and to use reflection within these proxy classes to resolve methods. This has two disadvantages: the proxies must either be generated or a dynamic proxy pattern must be used which requires some type abstraction. Furthermore, the use of reflection is slow.

In this paper, we suggest a simple yet elegant solution to this problem that tries to combine the advantages of the other methods discussed. Our approach is based on the idea of replacing existing invoke instructions of library methods ("cross-component invocations") by `invokedynamic` instructions. The runtime bootstrap mechanism we propose mimics the behaviour of a solution based on a `doesNotUnderstand`-like protocol. While we use a reflection protocol to locate target methods, the use of the protocol based on `invokedynamic` allows us to avoid much of the runtime overhead of traditional reflection. Bytecode is manipulated at compiletime and not when classes are loaded, thus avoiding runtime performance and configuration overhead. Also, the generated bytecode can be further optimised and analysed by the standard Java runtime (JIT, HotSpot). Due to the use of `invokedynamic`, the bytecode produced by our method is less prone to linkage errors than the bytecode produced by the standard Java compiler.

Our contributions are as follows:

- We present the *dynamo enhancer*, a bytecode manipulation framework that can be used to replace invocation instructions.
- We present the *dynamo compiler*, a Java compiler based on the dynamo enhancer.
- We present the *DynamoDSL*, a lightweight declarative language that can be used to specify when to replace invocations.

- We present a set of benchmarks used to measure the compile- and runtime performance overhead of dynamo.
- We present a case study that shows how the use of the dynamo compiler can prevent linkage errors that occur in the evolution of the *jasperreports* and *jfreechart* open source libraries.
- We present a second case study that shows how the use of the dynamo compiler can prevent stack overflow errors that can occur when methods are overridden with covariant return types.

The rest of this paper is structured as follows: first we discuss related work in Section 2 followed by a background in Section 3 where we discuss some key concepts used later. In Section 4, we present the actual compiler and its components, followed by a discussion of the runtime behaviour in Section 5. This is followed by Section 6 discussing benchmarks used for quality assurance and performance assessment, and two case studies in Section 7 extracted from problems encountered with real world systems. We finish our contribution with a brief conclusion.

2 Related Work

Our work addresses issues related to binary compatibility. The study of binary compatibility goes back to the work by Forman et al. [19], in the context of IBM's SOM object model. For Java, binary compatibility is formally defined in the language specification [22, sect 13]. Drossopoulou et al. have proposed a formal model of binary compatibility in [17].

Our work is motivated by issues that result from inconsistencies between source, binary and to some extent behavioural compatibility. These issues have been catalogued and studied by several authors, including des Rivières [10], Dietrich et al. [11, 12] and Raemakers et al. [32]. In particular, they include empirical studies [11, 32] showing that binary compatibility issues occur in practice when programs and libraries used by these programs evolve independently. Dig and Johnson [13] have conducted an API evolution case study on five real world systems (*struts*, *eclipse*, *jhotdraw*, *log4j* and a commercial application). They found that the majority of API-breaking changes were caused by refactoring, but did not distinguish between different types of compatibility. Mens et al. [27] have studied the evolution of Eclipse (from version 1.0 to version 3.3). The focus of this study was to investigate the applicability of Lehmann's laws of software evolution [25]. They found significant changes. Cosette and Walker have studied the evolution of APIs on a set of five Java open source programs [6]. They focused on generating change recommendation techniques that could then be used to give developers advice on how to refactor client code in order to adapt to API changes.

Several authors looked into how binary compatibility problems in Java programs can be avoided. Our work is somehow similar to binary component adaptation (BCA) by Keller et al. [24] as bytecode is modified in order to overcome certain compatibility problems. However, there are important differences: (1) BCA does not support changes to method descriptors, (2) changes must be specified by the user in the form of delta files while our approach is completely automated (3) BCA modifies libraries (components) whereas we transform the program itself (4) BCA can alter types and their relationships and is therefore rather invasive as it changes the semantics of client programs using reflection (for instance, when interfaces are added to classes, and the client program uses `instanceof` guards)[1]

[1] In a trivial sense, the dynamo compiler also changes the semantics of programs as method invocations that used to result in errors succeed after compilation with dynamo. But this effects are intended and local as they only impact the objects aliased at the call site, whereas changing types will affect other, unrelated parts of the program as well.

Other related works include Dmitriev's proposal to use binary compatibility checks in order to optimise build systems [16], Barr and Eisenbach's rule-based tool to compare library versions in order to detect changes that cause binary incompatibility [2], and refactoring-based approaches by Dig et al. [14] and Savga and Rudolf [7], aiming at generating a compatibility layer that ensures binary compatibility when used libraries evolve. Corwin et al. [5] have proposed a modular framework that uses a higher level API on top of the Java classpath architecture. This is somehow similar to how the OSGi framework [40] operates.

Our work relies on the use of the `invokedynamic` instruction. One of the key features of `invokedynamic` is that it adds reflection-like features to the language without incurring the performance overhead imposed by the use of traditional reflection. The low performance overhead has been confirmed by several authors including Kaewkasi [23] and Ortin and Conde [29, 3].

Within the Java standardisation process, JEP 276: Dynamic Linking of Language-Defined Object Models [38] is related to this research. The focus of JEP 276 is on supporting the compilation of object expressions often used in applications that require templating, while we suggest different compilation for standard Java call sites. The JEP 276 proposal does explicitly state that it does "not wish to provide linking semantics for operations for any single programming language or execution environment for any such language".

There is one open source project we are aware of with a similar aim – Kohsuke Kawaguchi's Bridge Method Injector [2]. The idea behind this project is to address one particular evolution problem we also consider - source-compatible changes of the return type. This is achieved by generating additional bridge methods in a post compilation step, a technique also used by the compiler when encountering co-variant return types. This will be discussed in more detail in section 7.2. The bridge method injector relies on an annotation that has to be added by the author of the library that evolves. This requires that the author understands the effects this change has on client code, and this might not always be the case [12]. On the other hand, our approach is completely automated and covers a wider range of evolution patterns.

This work is part of a wider trend towards self-adaptive software [36, 37].

3 Background

3.1 Binary vs Source Compatibility

The compiler and the linker use sets of rules to establish whether a method invocation can be resolved. These rule sets define *source compatibility* (in the case of the compiler) and *binary compatibility* (in the case of the linker). As far as Java (version 8) is concerned, binary compatibility is generally stricter than source compatibility as the compiler can reason about subtype relationships, the associations between primitive types and their respective wrapper types via auto-boxing and unboxing conversions [22, sects 5.1.7, 5.1.8] and invocations of static methods from non-static contexts [3].

For instance, consider Listing 1[4]. This example has a client program with a class `Main` invoking `Foo.get()` defined in a library that has two versions `lib-v1.jar` and `lib-v2.jar`. `Main` can be compiled against both versions of the library. However, when `Main` is compiled against `lib-v1.jar` and then executed with `lib-v2.jar`, a linkage error

[2] `http://bridge-method-injector.infradna.com/`

[3] There are some exceptions to this rule – situations where a program is binary but not source compatible with a library. For instance, the use of erasure can cause such scenarios.

[4] Package declarations and imports are omitted.

```
// lib-v1.jar
public class Foo {
  public static java.util.Collection get() {return new java.util.ArrayList();}
}
// lib-v2.jar
public class Foo {
  public static java.util.List get() {return new java.util.ArrayList();}
}
// client program
public class Main {
  public static void main(String[] args) {
     java.util.Collection coll = Foo.get();
     System.out.println(coll);
  }
}
```

Listing 1 Specialising the return type of a method.

```
// lib-v1.jar
public class Foo {
  public static void bar(int i) {System.out.println(i);}
}
// lib-v2.jar
public class Foo {
  public static void bar(Integer i) {System.out.println(i);}
}
// client program
public class Main {
  public static void main(String[] args) {new Foo().bar(42);}
}
```

Listing 2 Boxing of a parameter type.

(`NoSuchMethodError`) is thrown as the descriptor `get()Ljava/util/Collection;` found at the call site does not match the new descriptor `get()Ljava/util/List;` and can therefore not be resolved.

Listing 2 is similar, but this time the sole parameter type of `bar()` is changed from `int` to `java.lang.Integer`. The compiler deals with this situation by applying auto-boxing. But since the descriptor changes from `bar(I)V` to `bar(Ljava/lang/Integer;)V`, this change is not binary compatible.

Finally, consider Listing 3. `Main` still compiles against `Foo` in `lib-v2.jar`, but this time the compiler has to apply two adaptation rules: auto-boxing and type generalisation. The type of the invocation must also be changed as the `static` modifier has been added to `bar`.

Evolution patterns and their impact on binary compatibility have been catalogued by ddes Rivières [10]. Some of these problems could be avoided if linking was more consistent with compilation. In particular, we are interested in the following evolution patterns that are source compatible, but not binary compatible:

1. the specialisation of a reference return type of a method
2. the narrowing of a primitive return type of a method
3. the generalisation of a reference parameter type of a method

```
// lib-v1.jar
public class Foo {
  public void bar(int i) {System.out.println(i);}
}
// lib-v2.jar
public class Foo {
  public static void bar(Object i) {System.out.println(i);}
}
// client program
public class Main {
  public static void main(String[] args) {new Foo().bar(42);}
}
```

Listing 3 Complex (but still compatible) evolution.

4. the widening of a primitive parameter type of a method
5. replacing the primitive return or parameter type of a method by the respective wrapper type
6. replacing the wrapper return or parameter type of a method by the respective primitive type
7. changing a non-static method to a static method
8. changing a class to an interface or vice-versa
9. some combinations of any number of evolution patterns from this list[5]

3.2 The invokedynamic Instruction

Prior to version 7, Java used four different bytecode instructions for method invocation. First, `invokestatic` is used to call static methods. Static methods are resolved at compile time. Next, `invokespecial` is used for special cases including constructor and private method invocations and invocations via `super`. Finally, `invokevirtual` and `invokeinterface` are used to call non-static, non-private methods. Dynamic dispatch is used here, i.e. the method which is invoked is only computed at runtime based on the actual type of the receiver.

Starting with Java 1.7, `invokedynamic` was added to the instruction set [35]. The motivation was to give programmers more control over the dispatch process, in particular to facilitate the implementation of dynamic languages like Ruby on the JVM [28]. The Java 7 JVM supports the `invokedynamic` instruction, it is not emitted by the Java 7 compiler. The Java 8 compiler uses `invokedynamic` to compile lambdas.

With `invokedynamic`, the method reference is resolved by means of a *bootstrap method*. This user-implemented method is then used to locate the actual method being invoked. This is fast as the bootstrap method is only invoked by the linker during the resolution phase [26, 5.4.3.6] and following method invocations skip the bootstrap process. At this point it is possible to implement adapters.

The bootstrap method represents the target method as an instance of a `java.lang.invoke.MethodHandle`. The method handles support transformations that can be used to achieve a linking behaviour that is similar to the behaviour of the compiler. This mapping behaviour is exposed in the API of `MethodHandle` by the `invoke` method (as opposed to the `invokeExact` method). The overall goal of this work is to use this API to perform

[5] See also section 5 for a discussion on which of those combinations are supported.

appropriate type conversions to match the call site with the method found in a library that may have evolved. The difficult part is to lookup the best method which is suitable for re-typing.

4 Compilation

The mechanism we are proposing is based on the idea of swapping invoke instructions in bytecode. Our aim is to do this at compile time, so that the bytecode produced by the compiler can be transparently used at runtime with minimal additional configuration needed. However, there is another use case: to retrofit existing code only available in compiled form. We address this use case first and introduce the *dynamo enhancer* - the tool that performs the bytecode transformations. We can then use the enhancer to design the actual *dynamo compiler*, implemented as a post compiler / wrapper around the enhancer and the standard Java compiler. This design gives the dynamo compiler almost instantly the quality attributes needed for real world application. In particular, most Java compiler optimisations are available to us. An additional benefit of this design is that the enhancer can be used as a post compiler for other compilers emitting JVM bytecode, such as `scalac`. In the final part of this section, we discuss *Dynamo DSL*, a lightweight domain-specific language that can be used to customise the dynamo-specific part of the compilation via a simple command line argument.

4.1 The Dynamo Bytecode Enhancer

The bytecode enhancer is used to transform the bytecode emitted by a standard Java compiler, and replaces selective `invokestatic`, `invokevirtual`, `invokeinterface` and `invokespecial` instructions (from hereto referred to as *classical invocations*) by `invokedynamic`. The intention of the selection is to only replace invocations where the target is located in a library that may have a separate update cycle. This is achieved through *filters*, described in more detail below.

Any classical invocation can be expressed by a tuple $t = (opcode, C, m, desc)$ where *opcode* represents one of the classical invocations, C is an owner class containing the method, m is the method's name and *desc* is the methods descriptor consisting of the formal parameter types and the return type of the method. To convert a method invocation to `invokedynamic`, the original instruction from the tuple t must be changed to fit the `invokedynamic` call site specifier [26, sect 4.7.23]. In particular, a reference to a bootstrap method must be provided that is then used at link time to locate the actual target of the invocation.

Given a tuple t containing call site information, a dynamic call site can be created as follows. An `invokedynamic` instruction is created that has an index pointing to a constant pool entry of the type `CONSTANT_InvokeDynamic`. This entry defines a bootstrap method with the following three parameters: (1) the `MethodHandles.Lookup` factory used to locate and check access to methods, (2) the method name, (3) and a `MethodType` representing the descriptor. Any number of additional user parameters may follow.

The JVM Specification provides information about descriptors used by the classical invocations and the descriptors required by method handles [26, sect 5.4.3.5]. By combining these two parts of the specification, we infer the transformation rules listed in Table 1.

As discussed earlier, C and m represent the method owner type and its name, respectively. Furthermore, $A*$ is a set of input parameter types, T represents the return type, and V represents the `void` type (used in the descriptors of constructors). For virtual and interface invocations, the owner type is prepended to the list of argument types in the target descriptor.

Table 1 Translation from method descriptors to method handles.

opcode	source C,m,desc	target desc	parameters
`invokevirtual`	`C,m,(A*)T`	`(C,A*)T`	
`invokestatic`	`C,m,(A*)T`	`(A*)T`	`C`
`invokeinterface`	`C,m,(A*)T`	`(C,A*)T`	
`invokespecial` (`new`)	`C,<init>,(A*)V`	`(A*)C`	
`invokespecial` (`super`)	`C,m,(A*)T`	`(C,A*)T`	

```
1 bipush    10
2 istore_1
3 ...    // new lib/Foo()
4 iload_1
5 invokevirtual lib/Foo.setValue:(I)V
```

Listing 4 Bytecode for `new Foo().setValue(10)` before enhancement.

```
1 bipush    10
2 istore_1
3 ...    // new lib/Foo()
4 iload_1
5 invokedynamic setValue:(Llib/Foo;I)V
```

Listing 5 Enhanced bytecode for code in listing 4.

This is necessary to check the type of the `this` reference passed as the first argument to the respective method handle. For constructors, C becomes the return type. The descriptors of method handles for static invocations do not contain owner types at all. For this reason, we pass the owner type as an additional, user defined, parameter to the bootstrap method.

The translation of the first three instructions in Table 1, `invokevirtual`, `invokestatic` and `invokeinterface` is straightforward as they all have similar semantics. For this reason the transformation only requires to swap instructions and customise descriptor according to the table. See Listings 4 and 5 for an example presenting simplified byte-code instructions. These instructions may all be used in cross-component invocations and thus each occurrence is a candidate for replacement by the bytecode enhancer.

The situation is less obvious for `invokespecial`. In this case, the instruction has several usages, including the invocation of private methods, the invocation of methods via the `super` keyword, and the invocation of constructor using the `this` keyword [26, sect 6.5]. We support the transformation of `invokespecial` used with `super` and constructors [6]. For `invokespecial (super)`, the transformation is equivalent to the transformations applied to `invokevirtual` and `invokeinterface`.

The transformation of `invokespecial` used in conjunction with the `new` keyword at object allocation sites is less straightforward. In bytecode, the respective methods invoked all have a special name `<init>`. A correct transformation requires the detection of a usage pattern that consists of a certain sequence of instructions, the so-called *bytecode behaviour*. The pattern for constructors consists of the following three instructions (1) `new C` (2) `dup` (3) `invokespecial C.<init>:(A*)V` [26, sect 5.4.3.5], possibly with some intermediate instructions. The semantics of this sequence is: (1) create a new object of type C and push it onto the stack, (2) duplicate the object on the top of the stack and (3) invoke the constructor. The object on the top of the stack is duplicated because one element is consumed (popped) by the constructor invocation and the object must remain on top of the stack so that the value can be assigned to a variable (usually using an `astore` instruction). To

[6] Invocations of private methods are out of scope as they can not be used for cross-component invocations.

```
1 new lib/Foo
2 dup
3 bipush 10
4 invokespecial lib/Foo."<init>":(I)V
```

Listing 6 Bytecode for `new lib.Foo(10)`.

```
1 bipush 10
2 invokedynamic C$D:(I)Llib/Foo;
```

Listing 7 Enhanced bytecode for code in listing 6.

```
1 java -cp dynamo<..>.jar org.dynamo.compiler.Compiler -sourcepath ./src -d ./bin
```

Listing 8 Basic use of dynamo to compile all Java source code files in the `src` folder and store the .class files in the `bin` folder.

replace this sequence, it must be changed to a standard method invocation, which means that the instructions `new C` and `dup` must be dropped and `invokespecial` must be replaced by `invokedynamic`. Note that `invokedynamic` returns a value, this value is pushed onto the stack and effectively replaces the two dropped instructions. If the constructor contains parameters, the transformed sequence will retain the instructions to push these parameters onto the stack before the `invokedynamic` instruction is executed. An example transformation is shown in Listings 6 and 7.

Since `invokedynamic` cannot use `<init>` as the name parameter for the bootstrap method [26, sect 4.10], an artificial name set to `C$D`, meaning "constructor dynamic", is used instead. The use of the $ sign in method names is legal, but according to the language specification "The $ sign should be used only in mechanically generated source code or, rarely, to access pre-existing names on legacy systems" [22, sect 3.8]. By complying with this rule it is very unlikely that this name choice will create conflicts with user code.

The actual bootstrap methods referenced in the modified bytecode are defined as static methods in the class `com.dynamo.rt.DynamoBootstrap`. There is one method for each instruction type (`bootstrapInterface`, `bootstrapStatic`, etc). The semantics of these methods is described in section 5.

4.2 The Dynamo Compiler

The dynamo compiler is implemented as a post compiler. It is a wrapper around the dynamo enhancer, combined with the standard Java compiler accessed via the compiler API (JSR-199) [1]. More explicitly, the dynamo compiler first uses the standard Java compiler through JSR-199 to compile compilation units in memory. The resulting bytecode is represented using a map that associates fully qualified class names with byte arrays. The compiler then uses the enhancer to apply bytecode transformations using information from (1) the compiler runtime parameters specifying the classpath, (2) compiler runtime parameters specifying filters.

The classpath information is used to determine component boundaries. The filters are described in the next section, the options of the compiler command line interface (CLI) are described in appendix B.

Figure 1 shows the design of the dynamo compiler; Listing 8 shows the basic usage of the compiler via its CLI.

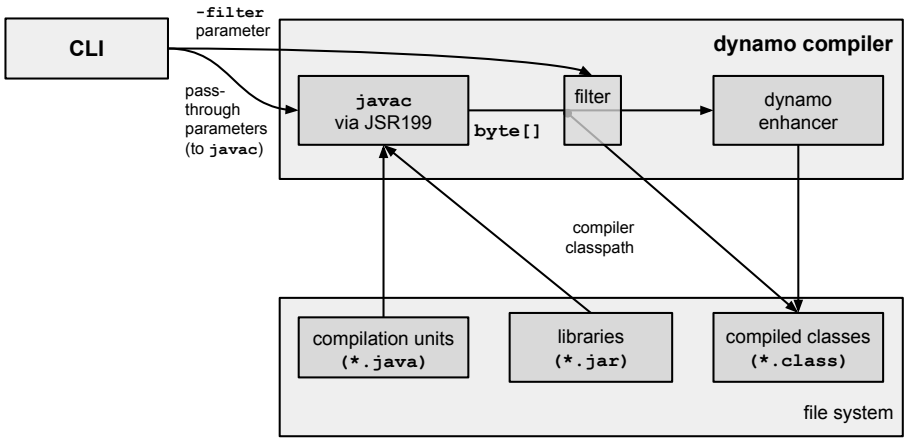

Figure 1 Compiler Design.

4.3 The Dynamo Filter DSL

It is often desirable for users to retain fine-grained control over the compilation process. One of the more common use cases is not to use `invokedynamic` when invoking functionality from Java platform libraries, due to the strong emphasis on backward compatibility in these libraries [8, 9].

For this reason, we have developed a lightweight domain-specific language that can be used to filter the methods and call sites where `invokedynamic` is used. These filters are used to select *invocation records* consisting of two methods, the method containing the call site, and the target method invoked before enhancement takes place.

The method filter is composed of individual filters that all have the following properties:

1. a *kind* (`+`, `-`) defining whether a filter is an include or an exclude filter
2. a *role* (`callsite`, `target`) defining whether a filter applies to the client method that has the call site, or to the target method to be invoked
3. a *class pattern* defining the classes to which the filter applies. Class names are fully qualified, with a dot used as package separator
4. an optional *method name pattern* defining the methods to which the filter applies
5. an optional *descriptor pattern* defining the descriptors to which the filter applies. This allows for the discrimination of overloaded methods. Descriptor patterns use the syntax defined in the JVM specification [26, sect 4.3.3] plus optional wild card characters.

Filters can be defined using a simple domain specific language; the grammar of this language is given in appendix A. The `*` and `?` wild cards can be used in class names, method names and descriptor patterns. The *default filter* is defined using the patterns listed in Table 2.

The intention of this filter is to exclude all invocations of target methods defined in the Java Development Kit. A user-defined filter can modify the default filter by using additional exclude and include patterns. The main use case for include patterns is to selectively permit the enhancement of JDK method invocations. With additional exclude patterns, users can manually specify that enhancements for certain targets are not required since the libraries in which the respective methods are defined are known to be API stable. In case additional include and exclude filters conflict, we resolve this as follows: invocation records are accepted if they are accepted by at least one of the include filter and not accepted by any of the exclude filters. This resolution is consistent with how include and exclude patterns are handled in

Table 2 The default filter.

Filter definition	Description
`+callsite *,+target *`	include all invocation records

`-target java.*`
`-target javax.*`
`-target com.sun.*`
`-target sun.*`
`-target com.oracle.*`
`-target org.ietf.*`
`-target org.omg.*`
`-target org.w3c.*`
`-target org.xml.*`
	then exclude all packages included in the Java development kit.

Table 3 Filter examples.

Filter definition	Description
`-callsite com.foo.Bar`	exclude invocations from call sites within `com.foo.Bar`
`-target com.foo.*`	exclude invocation of targets in methods in packages starting with `com.foo`
`+target java.lang.String#substring`	include invocations of the `substring` methods defined in `String`
`+target java.lang.String#substring(I)*`	include invocations of the `substring(int)` method defined in `String`

popular build tools, and we assume that developers are familiar with the practice. Some examples of filters are given in Table 3.

The definition of the default filter is relatively coarse. For instance, the filter would also include a package with the prefix `com.oracle` that is not part of the Java developer kit, such as certain JDBC drivers. In order to override this behaviour, users must use custom filters.

5 Linking

In this section, we describe the bootstrap process. We first discuss the strategy we are using to locate and select the target method. In a nutshell, we use an algorithm that aims at aligning linking behaviour with compile time behaviour to facilitate program comprehension by developers. We then describe the complexity of the algorithm, and some implementation issues.

5.1 Resolution

At runtime, we need to resolve the reference in the bootstrap method and locate an actual target method. This method must be *adaptable* to the original method. We only look for methods that are non-abstract, visible from the call site and have the same name and arity, but may have a different descriptor or are defined in a super type. This is similar to the problem the compiler has to solve when it selects the most specific method [22, sec 15.12.2.5]. However, we also have to take the return type into account. Not only can the return type change as we allow specialisation or narrowing of the return type, but we potentially also

have to deal with return type overloading, a situation that can only arise in bytecode. In particular, the standard Java compiler uses synthetic [26, sec 4.7.8] bridge methods to deal with co-variant return types. This means that any runtime method resolution has to be able to deal with a situation where there are multiple methods with the same name, the same parameter types but different return types within one class. Since return type overloading is not supported by the Java language, we can make the assumption that if this situation arises, only one of these methods is non-synthetic.

We refer to the selection of an adaptable method as *runtime resolution*, or in this context just *resolution* for short. Resolution is defined with respect to adaptations that can be performed by method handles [35]. Different resolution strategies are possible: (1) a *greedy strategy* that locates any adaptable method and selects it, (2) an *optimal strategy* that tries to find the "best match" according to some metric and (3) an *unambiguous best match strategy* that tries to find a method that is not only the best match in the sense of being optimal with respect to some order, but must also be strictly better than other candidates with respect to this order.

As one of our objectives is to address inconsistencies between source and binary compatibility, we decided to use strategy (3) to mimic the compiler behaviour of choosing the most specific method [22, sect 15.12.2.5]. While the overall aim is somehow similar to how the compiler processes method invocation expressions [22, sect 15.12], there are important differences caused by the differences in representation of language features such as generic types, static imports and varargs in source code and bytecode.

For a concise formulation of the algorithm, we propose a simple model, the *type conversion graph (TCG)*. This graph captures subtype relationships between classes and interfaces, boxing and unboxing relationships between wrapper types and their respective primitive types, and widening conversion relationships between primitive types [26, sect 5.1.2]. We build the TCG for a program by applying the following rules:

1. All (primitive and reference) types except annotation types but including array types that occur in the program are added as vertices to TCG.
2. If T_1 and T_2 are (non-generic) class, interface or array types and T_1 is a direct subtype of T_2 as defined in [22, sects 4.10.2, 4.10.3] then an edge $T_1 \to T_2$ with a label r is added to TCG.
3. If T_1 and T_2 are primitive types and there is a widening conversion from T_1 to T_2 as defined in [22, sect 5.1.2] then an edge $T_1 \to T_2$ with a label p is added to TCG.
4. If T is a primitive type, and Cl is the wrapper type of T, then an edge $T \to Cl$ with a label b (for boxing) and an edge $Cl \to T$ with a label u (for unboxing) are added to TCG.

Figure 2 shows an example TCG. The intention behind the TCG is that paths between types represent valid (potentially composite) conversions. However, the compiler imposes additional constraints. For instance, a widening primitive conversion followed by a boxing conversion is not permitted in assignment and invocation contexts [22, sects 5.2, 5.3]. As our motivation is to align compilation and linking behaviour, we impose the same restrictions. We do so by defining a *valid path* between two types considered as vertices in the TCG as a path such that the labels of the edges in this graph spell a word defined by the following simple regular grammar: `p|r*|br*|up`. This grammar is derived from the rules used in [22, sects 5.2, 5.3][7].

[7] The identity conversion is represented by a path of length 0. The primitive widening conversion rules in [22, sect 5.1.2] use the transitive closure for primitive widening conversions, we can therefore use `p` instead of `p*` in the grammar

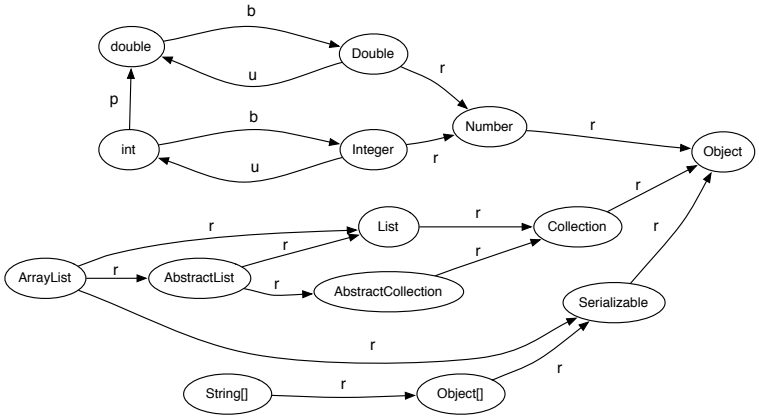

Figure 2 Example Type Conversion Graph.

When we try to locate an adaptable method, we only consider methods that are non-synthetic[8]. Given the TCG, the adaptability relationship between types $\sqsubseteq_{type} \subseteq T \times T$ can be easily defined as $t_1 \sqsubseteq_{type} t_2$ iff there is a valid path from t_1 to t_2 in the TCG. In order to identify, compare and select adaptable methods, we need to analyse their (1) defining type (2) parameter types and (3) return types. We define an order between methods based on their *extended descriptor* (*XD*) comprising the type where the method is defined, followed by the parameter types and the return type. We use the syntax `<defining_type>(parameter_type*)return_type` for XDs.

The relation $\sqsubseteq_{type}$ can be easily promoted to a relationship between XDs of the same arity: $T^1(A_1^1,..,A_n^1)R^1 \sqsubseteq_{desc} T^2(A_1^2,..,A_n^2)R^2$ iff $R^2 \sqsubseteq_{type} R^1$ and $T^1 \sqsubseteq_{type} T^2$ and $A_i^1 \sqsubseteq_{type} A_i^2$ for each i. Note that the direction of $\sqsubseteq_{type}$ is reversed ("contravariant") for return types. By treating the defining type as a virtual first parameter, we include possible target methods defined in super types. For instance, this allows adaptations in cases where methods have been pushed up the type hierarchy by refactoring[9].

We can then define a simple disambiguation algorithm as follows: given an extended descriptor $\Delta = T(A_1,..,A_n)R$ and a set of adaptable descriptors $\{\Delta^i\}$, where $\Delta \sqsubseteq_{desc} \Delta^i$, we chose a descriptor Δ^k from $\{\Delta^i\}$ if $\Delta^k \sqsubseteq_{desc} \Delta^i$ for all $i \neq k$. This captures the intention of selecting the *most specific method* that is adaptable. In case there is more than one minimal XD with respect to $\sqsubseteq_{desc}$, and the respective methods have the same parameter types, we chose the method with the more specific return type following the compiler [22, sect 15.12.2.5]. If resolution fails to detect a unique XD, the process will result in a linkage error (instance of `java.lang.NoSuchMethodError`).

An example where this occurs is when a method `Object foo(java.util.ArrayList)` is replaced by two methods `String foo(java.util.AbstractList)` and `Object foo(java.io.Serializable)`[10] within the same class. Both methods are suitable targets, but disambiguation fails to detect a best method that is strictly better than the other one.

A side effect of the resolution algorithm just described is that the only possible targets for replaced constructor invocations are constructors defined within the same class. This

[8] This will remove ambiguity if several methods with the same name and parameter type but different return types are present. This is discussed in more details in section 7.2

[9] The standard JVM runtime method resolution already includes superclass lookup [26, sect 5.4.3.3]

[10] The class `ArrayList` extends `AbstractList` and implements the interface `Serializable`.

follows from the fact that we only look for targets with the same name as the original target (i.e., `<init>`). While we also consider constructors from super classes as possible targets, these methods have a more general return type, and therefore do not qualify. It follows further that the type returned by the `invokedynamic` invocation is the same as the type of the original (`invokespecial (new)`) invocation.

5.2 Algorithm Complexity

Adaptability is defined with respect to reachability in the TCG. Reachability is a potentially expensive operation. The worst case complexity of standard shortest path algorithms used to query adaptability between two types is quadratic [15], while the pre-computation of a reachability index that enables constant time queries is super-quadratic but sub-cubic [4]. However, the paths that need to be traversed by the algorithm are generally short because of the flat hierarchies found in most Java programs [39], and the complexity of standard shortest path and many reachability algorithms is near linear for sparse graphs.

We provide further evidence of the performance of our approach in the evaluation section.

5.3 Implementation Issues

The runtime component is implemented as a small library that must be included in the class path of applications compiled or enhanced with dynamo. The method lookup algorithm is implemented in `org.dynamo.rt.DynamoBootstrap`. This class has five static bootstrap methods corresponding to the different instructions (`bootstrapVirtual`, `bootstrapStatic`, `bootstrapInterface`) and the two variants of `invokespecial` supported (`bootstrapSpecialNew`, `bootstrapSpecialSuper`). These methods differ slightly in terms of decoding and interpreting method types according to Table 1. They all invoke a common method which finds the most suitable method as described in section 5.1.

We use reflection to gather information about types, their relationships and members, and to reason about this information. Although reflection is not ideal as it may trigger some unnecessary class loading if classes are analysed for members that are not used at the end, it facilitates the implementation of bootstrap methods. The actual target finder algorithm produces an instance of `java.lang.reflect.Method` which is easily converted to a method handle using the `MethodHandles.Lookup.unreflect()` protocol and its variants for constructors and special methods. The actual conversion of types is then easily performed by `MethodHandle.asType()`. At the end of this process, a constant call site wrapping this handle is instantiated.

The method `MethodHandle.asType()` supports all of the type conversions we use and therefore effectively performs the transformation of both return and parameter types. Since we replace classic invocations with `invokedynamic`, we automatically support cases where interfaces are converted to classes or vice-versa. Finally, changes where non-static target methods have evolved to static methods are handled by ignoring the first parameter when the method is invoked. This way, the descriptor described in Table 1, row 2 is produced. The transformation is achieved by using the `MethodHandles.dropArguments` API. The (simplified) implementation of a bootstrap method is shown in Listing 9.

6 Benchmarks

In this section we discuss sets of benchmarks used to test various dynamo components, and to assess the performance overhead induced by dynamo.

```
import java.lang.invoke.*;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;

...
CallSite bootstrapVirtual(Lookup caller, String name, MethodType type) {
   Class<?> owner = type.parameterType(0);
   Method method = find(owner, name, type); // use resolution with TGC
   MethodHandle handle = lookup.unreflect(method);
   if (Modifier.isStatic(method.getModifiers())) {
      handle = MethodHandles.dropArguments(handle, 0, method.getDeclaringClass());
   }
   return new ConstantCallSite(handle.asType(type));
}
...
```

Listing 9 Bootstrap method implementation example (simplified).

6.1 Compiler Benchmarks

The compiler benchmarks are based on a comprehensive set of unit tests we have developed to quality assure dynamo. Tests are based on scenarios, each scenario consists of three classes:

1. `version1/lib/Foo.java` - the source code of version 1 of a class `lib.Foo` providing a method
2. `version2/lib/Foo.java` - the source code of version 2 of a class `lib.Foo` providing a modified method
3. `client/Main.java` - the source code of the client program using the method provided by `Foo`

During testing, the following sequence is executed for each scenario:

- Compile both versions of `Foo` and package them in different libraries (`lib-v1` and `lib-v2`). This should succeed.
- Compile `Main` with javac against `lib-v1`. This should succeed.
- Run the compiled class `Main` with `lib-v1`. This should succeed.
- Run the compiled class `Main` with `lib-v2`. This should fail.
- Recompile `Main` with dynamo against `lib-v1`. This should succeed.
- Run the recompiled class `Main` with `lib-v2`. This should succeed, except for the last two "ambiguous" benchmarks designed to produce linkage errors.

Scenarios are identified by self-descriptive unique names. For instance `invokeinterface_narrow_ret` is a scenario that contains an `invokeinterface` instruction with a reference to a method that evolves by narrowing the return type. Table 4 shows an overview of the scenarios used for testing, the first column describes the change pattern, the other columns show the type of invocation that is being converted. These scenarios were designed to obtain full coverage of all possible combinations for common scenarios, and good partial coverage for more exotic cases such as chained conversions (like boxing and then generalising a parameter type). A check mark in the row-column intersection means that there is such a scenario, `n/a` indicates that such a scenario is impossible (example: return type conversions for constructor invocations) or does not make sense (example: converting an invocation of a certain type to an invocation of the same type without adapting parameter or return types).

Table 4 Benchmark overview.

	`invoke-static`	`invoke-virtual`	`invoke-interface`	`invoke-special (super)`	`invoke-special (new)`
narrow return type	✓	✓	✓	✓	n/a
specialised return type	✓	✓	✓	✓	n/a
specialised array return type	✓	✓	✓	✓	n/a
box return type	✓	✓	✓	✓	n/a
unbox return type	✓	✓	✓	✓	n/a
widen parameter type	✓	✓	✓	✓	✓
generalise parameter type	✓	✓	✓	✓	✓
generalise array parameter type	✓	✓	✓		
generalise array parameter to `Object`			✓		
generalise array parameter to `Serializable`			✓		
generalise array parameter to `Cloneable`			✓		
box parameter type	✓	✓	✓	✓	✓
unbox parameter type	✓	✓	✓	✓	✓
multiple methods w/ generalised parameters	✓				
convert to `invokestatic`	n/a	✓	✓	n/a	n/a
convert to `invokeinterface`	n/a	✓	n/a	n/a	n/a
convert to `invokevirtual`	n/a	n/a	✓	n/a	n/a
unbox and widen parameter type	✓	✓	✓		✓
box and narrow return type	✓	✓	✓		n/a
box and generalised parameter type	✓	✓	✓	✓	✓
method ambiguous parameter		✓			
method ambiguous two parameters		✓			

We use the test scenarios as compiler micro-benchmarks. However, as we do not test linking at this stage, for some scenarios the code compiled with dynamo (`Main.java`) is identical. For instance, this is the case for the various "generalise array parameter .. " scenarios. We remove these scenarios from the set of 65 scenarios listed in Table 4 in order to avoid double counting, the result is a set of 49 scenarios we can use for benchmarking.

Performance measurement in Java is not straightforward as Java uses runtime optimisation (JIT, HotSpot). To account for this, repeated invocations and JVM warm-up runs are necessary to produce statistically meaningful results [20]. For this reason, we used JMH,[11] a tool that provides the respective features. For each experiment, we executed 15 warm-up and 30 trial runs. For the experiments, we used the Java(TM) SE Runtime Environment (build 1.8.0_20-b26) with a Java HotSpot(TM) 64-Bit Server VM (build 25.20-b23, mixed mode) on a MacBook with OSX 10.5.5, an 2.8 GHz Intel Core i7 processor, 16 GB 1600 MHz DDR3 and SSD disk.

Table 5 summarises the performance results. We compare the dynamo compiler with the standard Java compiler accessed through JSR199. To achieve a fair comparison, we compile in memory in both cases and try to measure the net effect of the post compilation step when bytecode is manipulated. We report average, standard deviation and confidence intervals, all in ms. The fourth column contains the number of benchmarks tested, and column 5 is the average divided by this number, i.e., the typical time a single compilation takes. The results

[11] `http://openjdk.java.net/projects/code-tools/jmh/`

Table 5 Dynamo vs classic compiler performance.

benchmarks	average runtime (ms)	stdev (ms)	bench-mark count	average runtime single benchm. (ms)	confidence interval (ms) (99.9%)
classic	1923	65	49	39.24	[1857, 1988]
dynamo	2142	91	49	43.71	[2051, 2233]

Table 6 Method resolution performance.

benchmarks	average runtime (ms)	stdev (ms)	bench-mark count	average runtime single benchm. (ms)	confidence interval (ms) (99.9%)
all	0.180	0.001	14	0.013	[0.179, 0.181]

clearly show that the compile time overhead imposed by dynamo is very small.

6.2 Runtime Benchmarks

We have also created a benchmark for runtime resolution. As the low performance overhead of `invokedynamic` is known and has been reported in previous research [29, 3], the benchmark is based on unit tests that only compute the target methods using the resolution algorithm described in 5.1 from a set of candidate methods. Therefore, the design of the benchmarks is driven by the following considerations: (1) a benchmark has a (significant) number of candidate target methods scattered across several classes within the type hierarchy, (2) candidate methods are detected via conversions represented by non-trivial paths within the TCG including combinations of different edge types.

The individual benchmark scenarios are implemented as standard JUnit tests, and used for quality assurance as well as to assess performance. Each test case has a self-explanatory name to express its purpose. The packages and respective test cases are:

- `specstaticreturntype` – a simple scenario with a test to locate a static target method with a specialised return type
- `boxunbox` – a scenario with tests that require the following conversions to locate the target method: (1) boxing, (2) un-boxing, (3) boxing and widening of parameter types and (4) un-boxing and narrowing of the return type
- `methodoverloading` – a scenario with multiple d potential target methods, and tests checking different resolution strategies to select target methods
- `overloadinghierarchy` – a scenario with multiple overloaded potential target methods scattered across a class and its direct and indirect super classes
- `disambiguatebyreturntype` – a scenario to test situations where the target method is selected based on the return type, simulating a scenario similar to what will be further discussed in section 7.2

Table 6 summarises the performance results, using the same format as table 5 described above. This result confirms that the performance overhead of runtime resolution is negligible.

7 Case Studies

To demonstrate the applicability of our approach, we present two case studies sourced from real world scenarios.

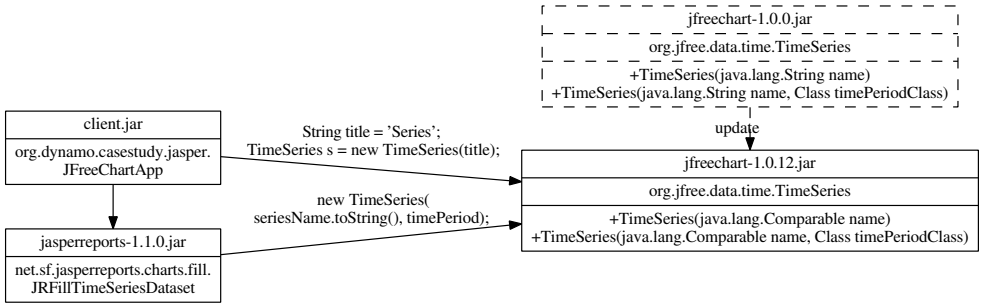

Figure 3 Case study 1 application design.

7.1 Solving Compatibility Issues between Jasperreports and Jfreechart

The first case study is sourced from a real world scenario using the popular Java reporting library *jasperreports*[12]. *Jasperreports* uses several other libraries including *jfreechart*[13], which evolve independently, and this can lead to incompatibilities. In particular, a problem occurs when *jfreechart* evolves from version 1.0.0 to 1.0.12. In this upgrade, the constructor `TimeSeries(String)` in the class `org.jfree.data.time.TimeSeries` is changed to `TimeSeries(Comparable)`. There are several overloaded variants of this constructor, but for all of them the type of the first parameter is changed from `String` to `Comparable`.

This change is particularly dangerous for two reasons: (1) The change happens between two micro versions, such evolutions are widely regarded as being compatible in accordance with popular semantic versioning schemes [30, 31]. These schemes are widely used to represent compatibility contracts between collaborating components, and tools and frameworks like Maven and OSGi support them. (2) This change only affects binary compatibility, and its implications therefore depend on the mode of deployment. The issue does not occur when *jasperreports* is built, but only if a dynamic system upgrade mechanism like OSGi or Java WebStart is used. The key observation here is that `String` implements the `Comparable` interface, and that generalising a parameter type generally preserves source compatibility.[14]

The system used in the case study is depicted in Figure 3. The system is represented using a model resembling a UML class diagram. However, the arrows represent the invocation of a method or constructor in the target class, and the dashed box represents the older version of the library. Labels on edges show the Java code at the respective call site. The model contains an application `client.jar`, which is a simple client application producing a basic report containing a chart. The example illustrates both compile and runtime dependencies. While the client application is compiled and invoked against both libraries, *jasperreports* runs only against *jfreechart-1.0.0* and this dependency is not resolved at compile time.

This setting allows for at least three scenarios: (1) the client is compiled and executed with *jfreechart-1.0.0*, (2) the client is compiled against *jfreechart-1.0.0* and executed with *jfreechart-1.0.12*, (3) the client is compiled and executed with *jfreechart-1.0.12*. Only the first scenario succeeds with the standard Java compiler. In the second scenario, both dependencies *client* → *jfreechart* and *jasperreports* → *jfreechart* will fail due to the incompatible change in `TimeSeries(..)`. In the third scenario, the problem will be solved for the client as it can

[12] `http://community.jaspersoft.com/project/jasperreports-library`
[13] `http://www.jfree.org/jfreechart/`
[14] There are exceptions where generalising a parameter type can create ambiguities in the presence of overloading that lead to compilation errors.

```
ldc           #10  // String 'Series'
astore_2
new           #11  // class org/jfree/data/time/TimeSeries
dup
aload_2
invokespecial #12 // Method org/.../TimeSeries."<init>":(Ljava/lang/String;)V
```

Listing 10 Client code compiled with javac.

```
ldc           #10     // String 'Series'
astore_2
aload_2
invokedynamic #130, 0 // InvokeDynamic #3:C$D:(Ljava/../String;)Lorg/../TimeSeries;

BootstrapMethods:
 3: #126 invokestatic org/dynamo/rt/DynamoBootstrap.bootstrapSpecialNew:(
          Ljava/lang/invoke/MethodHandles$Lookup;
          Ljava/lang/String;
          Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;
```

Listing 11 Client code compiled with dynamo.

now be compiled, although invocations of `TimeSeries(..)` from *jasperreports* will still fail. However, the working client will remain prone to similar problems as *jfreechart* may evolve further.

To solve this problem, we compiled the client with the dynamo compiler. This produces a client program that is compatible with both versions of *jfreechart* and resilient to any possible future change of a similar nature. We also processed the *jasperreports* library with the dynamo enhancer, demonstrating the application to third party or legacy software where only bytecode is available. Finally, we invoked the program and verified by inspection that both versions produce the same report. Listings 10 and 11 show the bytecode of the original client compiled with the standard Java compiler followed by the enhanced bytecode. The output is provided in the format produced by the `javap` tool, with comments revealing values defined in the constant pool. The modifications follow the process described above: the instructions creating the object and invoking its constructor are replaced by `invokedynamic`, which refers to the respective bootstrap method.

We have also used this case study as a macro benchmark in order to measure the overhead of using dynamo on a real-world program. Again, JMH was used to conduct these experiments, with 15 warm-up and 30 trial runs. Table 7 summarises the compiler results, and confirms that the performance overhead of using dynamo is small.

We have also measured the runtime overhead. In this benchmark, a simple report is generated first by running a simple demo program that uses *jasperreports* compiled with the classic compiler, and then by running it again with *jasperreports* compiled by dynamo. We do use the same version of *jfreechart* (1.0.0) in both cases in order to avoid a bias that would be caused by the different performance characteristics of different versions of this library. Therefore, the same classes are used in both cases, but the generated bytecode and the method used for linking differs. The respective results are reported in table 8. This again confirms that the overhead of dynamo runtime resolution is small.

Table 7 Dynamo vs classic compiler performance for *jasperreports*.

benchmark	average runtime (ms)	stdev (ms)	confidence interval (ms) (99.9%)
classic	1278	52	[1255,1302]
dynamo	1330	56	[1305,1355]

Table 8 Runtime performance of simple report generation with *jasperreports*.

benchmark	average runtime (ms)	stdev (ms)	confidence interval (ms) (99.9%)
classic	168	9	[165,172]
dynamo	175	12	[154,218]

7.2 Avoiding the Hazards of Covariant Return Types and Bridge Methods

In a second case study, we demonstrate how compilation with dynamo can avoid an error that was encountered when the JDK was refactored to use more covariant return types when overriding `clone()` methods[15]. We use the formulation of this problem presented in [34]. The scenario consists of three simple classes, `Wrapper`, `WrapperChild` and `WrapperGrandchild`, the source code is shown in listing 12. Both `WrapperChild` and `WrapperGrandchild` override `wrap`, but `WrapperGrandchild` does so using a covariant return type. When a client calls `wrap` on an object that is an instance of `WrapperGrandchild` but declared as `Wrapper`, an `invokevirtual` instruction pointing to a `wrap` method with the descriptor `(LObject;)LCollection;` (package names omitted) is used. Therefore, a method with such a descriptor must exist in `WrapperGrandchild`. The compiler solves this problem by generating a synthetic bridge method with the descriptor `(LObject;)LCollection;` that then just delegates to the actual method with the descriptor `(LObject;)LList;`. This is therefore a case of return type overloading. The respective bytecode is shown in listing 13[16].

The problem occurs when `Wrapper` and `WrapperChild` on the one hand and `WrapperGrandchild` on the other hand are deployed in different libraries. When `WrapperChild` is refactored to also use a covariant return type, it too gets two `wrap` methods - the actual method with the descriptor `(LObject;)LList;` and the synthetic methods with the descriptor `(LObject;)LCollection;`. Now assume we execute the code in listing 14.

Executing this line invokes `WrapperGrandchild.wrap:(LObject;)LList;`. This method calls `WrapperChild.wrap:(LObject;)LCollection;` via `super` as this was the only method in `WrapperChild` that was available when `WrapperGrandchild` was compiled against the old version of `WrapperChild`. This method is now a bridge method, and therefore calls `WrapperChild.wrap:(LObject;)LList` using an `invokevirtual` instruction. Since the actual type of the receiver object is `WrapperGrandchild`, this call is dispatched to `WrapperGrandchild.wrap:(LObject;)LList;`. This causes the program to loop infinitely and terminate with a `StackOverflowError`. The respective call graph is shown in Figure 4.

Using the dynamo compiler prevents this from happening not because of the signature adaptation, but due to the disambiguation strategy used during runtime resolution. Resolution only considers non-synthetic methods. Therefore, we have two ad-

[15] `http://mail.openjdk.java.net/pipermail/core-libs-dev/2012-January/009119.html` [Accessed: October 20, 2015]

[16] The repeated `checkcast` instruction is a Java compiler bug reported in `http://bugs.java.com/bugdatabase/view_bug.do?bug_id=6246854`[Accessed: May 3, 2016]

```
public class Wrapper {
    public Collection wrap(Object o) {
          Collection c = new ArrayList();
          c.add(o);
          return c;
    }
}

public class WrapperChild extends Wrapper {
    @Override public Collection wrap(Object o) {
          return super.wrap(o);
    }
}

public class WrapperGrandchild extends WrapperChild {
    @Override public List wrap(Object o) {
          return (List) (super.wrap(o));
    }
}
```

Listing 12 Case study 2 source code (imports omitted).

```
// access flags 0x1
public wrap(Ljava/lang/Object;)Ljava/util/List;
ALOAD 0
ALOAD 1
INVOKESPECIAL WrapperChild.wrap (Ljava/lang/Object;)Ljava/util/Collection;
CHECKCAST java/util/List
CHECKCAST java/util/List
ARETURN

// access flags 0x1041
public synthetic bridge wrap(Ljava/lang/Object;)Ljava/util/Collection;
ALOAD 0
ALOAD 1
INVOKEVIRTUAL WrapperGrandchild.wrap (Ljava/lang/Object;)Ljava/util/List;
ARETURN
```

Listing 13 Case study 2 bytecode of `WrapperGrandchild`.

```
new WrapperGrandchild().wrap("x");
```

Listing 14 Case study 2 client code.

```
1  public wrap(Ljava/lang/Object;)Ljava/util/List;
2   aload_0
3   aload_1
4   invokedynamic #31 // wrap:(LWrapperChild;Ljava/lang/Object;)Ljava/util/Collection;
5   checkcast java/util/List
6   checkcast java/util/List
7   areturn
8
9  public synthetic bridge wrap(Ljava/lang/Object;)Ljava/util/Collection;
10  aload_0
11  aload_1
12  invokevirtual #4 // Method wrap:(Ljava/lang/Object;)Ljava/util/List;
13  areturn
14
15 Constant pool:
16  #4 = Methodref // WrapperGrandchild.wrap:(Ljava/lang/Object;)Ljava/util/List;
17 #31 = InvokeDynamic// wrap:(LWrapperChild;Ljava/lang/Object;)Ljava/util/Collection;
```

Listing 15 Bytecode of `WrapperGrandchild` generated by dynamo.

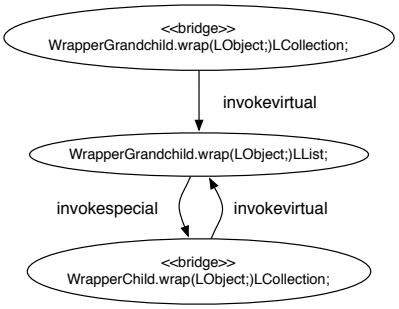

Figure 4 Call graph after refactoring, compiled with javac.

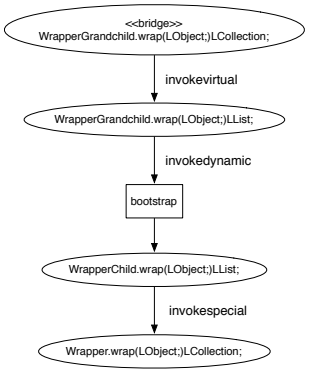

Figure 5 Call graph after refactoring, compiled with dynamo.

aptable candidate methods with the extended descriptors `Wrapper:(Object)Collection` and `WrapperChild:(Object)List`. Both are minimal (most specific), but because they both have the same parameter types, the method with the more specific return type `WrapperChild:(Object)List` is selected. The replacement pattern used is `invokespecial` (`super`) as the call via `super` is the cross-component call. The bytecode produced by *dynamo* is shown in Listing 15, the respective call graph is shown in Figure 5.

8 Conclusion

In this paper, we have proposed the dynamo framework consisting of a bytecode enhancer, a compiler, a filter DSL and a runtime component. We have demonstrated the benefits of using dynamo using a set of benchmarks and two case studies sourced from real-world scenarios. In both case studies, unintuitive runtime errors can be avoided by using the dynamo compiler. The benchmarks show that the overhead of using dynamo is low.

While we have discussed dynamo in the context of the Java language and compilation targeting the JVM, the same idea can be potentially applied to other languages and platforms.

The enhancer can be used as a tool kit to improve other compilers targeting the JVM, such as `scalac`. Dynamo could also be ported to other language / platform combinations facing similar problems related to binary compatibility, such as `C#/CLR/.NET`.

In the context of Java, the same idea could also be applied to support compatible type changes for fields exposed by libraries. We did not include this in our work as direct field access in Java programs is discouraged. Other possible extensions include to restrict constant inlining across components.

Acknowledgements. The authors would like to thank Alex Buckley, Nicholas Hollingum, Stepanka Jezkova and Alex Potanin for their valuable feedback.

References

1. JSR-000199 Java™ Compiler API. `https://jcp.org/aboutJava/communityprocess/final/jsr199/index.html`. [Accessed: October 20, 2015], 2006.
2. Miles Barr and Susan Eisenbach. Safe upgrading without restarting. In *Proceedings ICSM'03*, pages 129–137, 2003.
3. Patricia Conde and Francisco Ortin. JINDY: A java library to support invokedynamic. *Computer Science and Information Systems*, 11(1):47–68, 2014.
4. Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings STOC'87*. ACM, 1987.
5. John Corwin, David F. Bacon, David Grove, and Chet Murthy. Mj: a rational module system for java and its applications. In *Proceedings OOPSLA'03*. ACM, 2003.
6. Bradley E Cossette and Robert J Walker. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *Proceedings FSE'12*. ACM, 2012.
7. Ilie Şavga and Michael Rudolf. Refactoring-based support for binary compatibility in evolving frameworks. In *Proceedings GPCE '07*. ACM, 2007.
8. Joseph D. Darcy. Kinds of compatibility: Source, binary, and behavioral. `https://blogs.oracle.com/darcy/entry/kinds_of_compatibility`. [Accessed: October 20, 2015], 2008.
9. Joseph D. Darcy. JDK release types and compatibility regions. `https://blogs.oracle.com/darcy/entry/release_types_compatibility_regions`. [Accessed: October 20, 2015], 2009.
10. Jim des Rivières. Evolving Java-based APIs. `http://wiki.eclipse.org/Evolving_Java-based_APIs`. [Accessed: Nov. 20, 2015], 2007.
11. Jens Dietrich, Kamil Jezek, and Premek Brada. Broken promises - an empirical study into evolution problems in java programs caused by library upgrades. In *Proceedings CSMR-WCRE'14*. IEEE, 2014.
12. Jens Dietrich, Kamil Jezek, and Premek Brada. What java developers know about compatibility, and why this matters. *Empirical Software Engineering*, pages 1–26, 2015.
13. Danny Dig and Ralph Johnson. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, 2006.
14. Danny Dig, Stas Negara, Vibhu Mohindra, and Ralph Johnson. ReBA: refactoring-aware binary adaptation of evolving libraries. In *Proceedings ICSE'08*. ACM, 2008.
15. Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.
16. Mikhail Dmitriev. Language-specific make technology for the Java programming language. In *Proceedings OOPSLA '02*, pages 373–385, New York, NY, USA, 2002. ACM.
17. Sophia Drossopoulou, David Wragg, and Susan Eisenbach. What is Java binary compatibility? In *Proceedings OOPSLA'98*. ACM, 1998.

18 David Flanagan and Yukihiro Matsumoto. *The ruby programming language*. O'Reilly Media, Inc., 2008.

19 Ira R. Forman, Michael H. Conner, Scott H. Danforth, and Larry K. Raper. Release-to-release binary compatibility in SOM. In *Proceedings OOPSLA '95*. ACM, 1995.

20 Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings OOPSLA'07*. ACM, 2007.

21 Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

22 James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java™ Language Specification (Java SE 8 Edition)*. Oracle America, Inc., February 2015.

23 Chanwit Kaewkasi. Towards performance measurements for the java virtual machine's invokedynamic. In *Proceedings VIML'10*. ACM, 2010.

24 Ralph Keller and Urs Hölzle. Binary Component Adaptation. In *Proceedings ECOOP '98*. Springer, 1998.

25 M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

26 Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java™ Virtual Machine Specification (Java SE 8 Edition)*. Oracle America, Inc., February 2015.

27 Tom Mens, Juan Fernández-Ramil, and Sylvain Degrandsart. The Evolution of Eclipse. In *Proceedings ICSM'08*. IEEE, 2008.

28 Charles Nutter. A first taste of invokedynamic. `http://blog.headius.com/2008/09/first-taste-of-invokedynamic.html`. [Accessed: October 20, 2015], 2008.

29 Francisco Ortin, Patricia Conde, Daniel Fernandez-Lanvin, and Raul Izquierdo. The runtime performance of invokedynamic: An evaluation with a java library. *IEEE software*, (4):82–90, 2014.

30 OSGi Alliance. Semantic versioning – technical whitepaper. Technical report, OSGi Alliance, 2010.

31 Tom Preston-Werner. Semantic Versioning 2.0.0. `http://semver.org/`. [Accessed: October 20, 2015], 2010.

32 Steven Raemaekers, Arie van Deursen, and Joost Visser. Measuring software library stability through historical version analysis. In *Proceedings ICSM'12*. IEEE, 2012.

33 Steven Raemaekers, Arie Van Deursen, and Joost Visser. Semantic versioning versus breaking changes: a study of the maven repository. Technical report, Delft University of Technology, Software Engineering Research Group, 2014.

34 Ian Robertson. A hazard of covariant return types and bridge methods. `http://www.artima.com/weblogs/viewpost.jsp?thread=354443`. [Accessed: October 20, 2015], 2013.

35 John R Rose. Bytecodes meet combinators: invokedynamic on the jvm. In *Proceedings VMIL'09*. ACM, 2009.

36 Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

37 Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. An analysis of language-level support for self-adaptive software. *ACM Trans. Auton. Adapt. Syst.*, 8(2):7:1–7:29, July 2013.

38 Attila Szegedi. JEP 276: Dynamic Linking of Language-Defined Object Models. `http://openjdk.java.net/jeps/276`. [Accessed: November 10, 2015], 2015.

39 Ewan Tempero, James Noble, and Hayden Melton. How do java programs use inheritance? an empirical study of inheritance in java software. In *Proceedings ECOOP'08*. Springer, 2008.

40 The OSGi Alliance. OSGi core release 5. `http://www.osgi.org/Specifications`, 2012.

Appendices

A Dynamo DSL Grammar

```
kind     ::= '+' | '-'
role     ::= 'callsite' | 'target'
filters  ::= filter ( ',' filter )*
className ::= NAME
methodName ::= NAME
descriptor ::= '(' NAMES ')' NAME
filter   ::= kind role className ( '#' methodName descriptor? )?
NAMES    ::= ( NAME ( ',' NAME )* )+
```

`NAME` is used as defined in [22, sect 6], but with added support for the `*` and `?` wildcard characters.

B Dynamo Compiler Options

Table 9 Dynamo Compiler Options.

parameter	required	description
`-cp`	yes	Specify where to find user class files and annotation processors
`-d`	yes	Specify where to place generated class files
`-sourcepath`	yes	Specify where to find input source files
`-classic`	no	Classic compilation without enhancement - same as javac
`-encoding`	no	Specify character encoding used by source files
`-filter`	no	Specify which methods to enhance as defined in section 4.3
`-help`	no	Print instructions