Binary Refactoring: Improving Code Behind the Scenes

Eli Tilevich, Yannis Smaragdakis
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, USA
{tilevich, yannis}@cc.gatech.edu

ABSTRACT

We present Binary Refactoring: a software engineering technique for improving the implementation of programs without modifying their source code. While related to regular refactoring in preserving a program's functionality, binary refactoring aims to capture modifications that are often applied to source code, although they only improve the performance of the software application and not the code structure. We motivate binary refactoring, present a binary refactoring catalogue, describe the design and implementation of BARBER—our binary refactoring browser for Java, and demonstrate the usefulness of binary refactoring through a series of benchmarks.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—Restructuring, reverse engineering, and reengineering; D2.3 [**Software Engineering**]: Coding Tools and Techniques—Object-oriented programming; D.3.4 [**Programming Languages**]: Processors—Optimization

General Terms

Performance, Languages.

Keywords

Refactoring, adaptation, optimization, software evolution, maintenance, and bytecode engineering.

1. INTRODUCTION

The process of changing a software system that improves its internal structure without altering its external behavior is called *refactoring*. Traditionally, refactoring has entailed restructuring the source code of a program to make it easier to understand, maintain, and enhance, thereby improving the program's design. Refactoring has been actively explored by software engineering researchers and some of their ideas have successfully migrated to programming practice. For instance, several modern programming IDEs for mainstream languages such as C++, Java, and C# offer built-in refactoring support in the form of *refactoring browsers*.

In this paper we propose the concept of *binary refactoring* as a valuable software engineering technique. Binary refactoring is the application of refactoring transformations to a software application without affecting its source code. Thus, binary refactoring transformations are semantics-preserving (assuming correct application) and intended for performance optimization. Nevertheless, it is more appropriate to see binary refactoring as a technique that enhances maintainability without sacrificing performance, rather than as a technique for improving performance. The *raison d'être* of binary refactoring is not that current programs need more optimization but that programmers already apply code structure transformations for performance reasons, yet these transformations unnecessarily pollute the source code and affect its maintainability. For instance, classes are often split not because of design reasons but to exploit locality: one part of the object may need to be stored or transmitted independently of the rest. This source-code refactoring is common in server-side applications (e.g., with J2EE technology) in which objects need to be stored in databases and transmitted over the network. With binary refactoring, the class structure in the program can remain intact but a *split class* refactoring can produce the same performance benefit. As another example, source code modifications often are applied just to reduce indirection cost (e.g., by devirtualization, manual inlining, or the "remove middle man" source refactoring). In contrast, with binary refactoring all such transformations can be codified and applied through a refactoring browser without affecting the application source code.

Clearly, binary refactoring is closely related both to regular software refactoring and to annotation-guided compiler optimization. Nevertheless, we believe that our work makes novel contributions:

- We introduce binary refactoring as a general concept, worthy of careful study.
- We catalogue several useful refactorings, many of which (e.g., "split class", "glue classes") are much more complex and larger-scale than usual annotation-guided optimizations in standard compilers.
- We concretely demonstrate binary refactoring with a reference implementation. BARBER, our binary refactoring browser for Java, is a tool that we have found very appealing in our own software development work.
- We apply BARBER to several small and large benchmarks to showcase its effects. Although it is clear that binary refactoring can improve a program's

ICSE'05, May 15-21, 2005, St. Louis, Missouri, USA.

performance, it is occasionally surprising to quantify the improvement achieved with little effort.

The rest of this paper is organized as follows. Section 2 introduces the motivation for binary refactoring. Section 3 presents our refactoring catalogue. Section 4 discusses how binary refactoring fits into the software development process. Section 5 describes the design, implementation, and evaluation of our binary refactoring browser. Section 6 outlines related work, and Section 7 concludes.

2. MOTIVATION

"Premature optimization is the root of all evil"
C.A.R. Hoare

The motivation behind binary refactoring is offered by two common observations in software projects:

- programmers are trained to optimize their programs relative to facts that they statically know to be true (run-time invariants).
- *run-time invariants* are not necessarily *design invariants*: properties that always hold in the current version of the application are often not part of the long-term design throughout the application lifetime.

The consequence of these observations is that many program changes intended as performance optimizations actually conflict with an application's extensibility and maintainability. Such program changes are being performed all the time in actual projects, sometimes even without realizing their long-term implications.

As an extended example, consider an orders processing system implemented in Java. (We use Java as an example language throughout this paper.) One of the application's classes is class `Customer`. (For a concrete instance, see class `Customer` in the open source JFreeReport Library—`http://www.jfree.org/jfreereport/`. We later use this class in one of our experiments.) The implementation of this class may be quite mature and match the design intention. Nevertheless, some changes may become necessary due to the way `Customer` objects are used. Imagine that our application is deployed in a server-side environment: its functionality is accessed remotely over the internet. Good structuring of Java server-side applications (e.g., using the 3-tiered J2EE conventions) dictates separating the application presentation logic, business logic, and storage component. In this model, `Customer` objects often need to be copied over the network. Imagine a remote method that accepts a `Customer` parameter and uses it to calculate a shipping rate. Such a method likely uses only a few fields of `Customer`. If we could pass only the required fields to the remote method, then the cost of network communication would be reduced and the remote method would take less time to invoke. We could change the signature of the remote method to take several parameters instead of the single `Customer`, but that would violate the encapsulation principle: we do not want to use an object's fields independently of the object itself. Instead, a common way to address this problem is to split up the class into two classes so that one of the partitions ends up containing only the fields used by remote calls.

This partitioning of the `Customer` class is an example of a source-level transformation that is dictated entirely by performance concerns. From a design perspective, the split is likely not desirable: the two partitions are always in one-to-one correspondence and they are conceptually the same object. Furthermore, the partitioning is brittle with respect to changes of the functionality of remote methods: if a remote method needs to access more fields in the future, the partitioning needs to change.

In contrast to the practice of splitting the `Customer` class manually, we propose that the *split class* binary refactoring offers a convenient solution to this problem. This refactoring enables flexible splitting of a single class into multiple partitions as required by a given application scenario. Furthermore, the directions specifying how the splitting should be done and to what classes it should apply are described externally from the application's source code, thus preserving a clean design. A binary refactoring browser tool processes the input for the split class refactoring and applies the refactoring to the entire application: both the class and its clients may change as a result of the split class refactoring.

We would be amiss not to mention up front that much research has gone into optimizing this special case of "split class" for distributed computing. Various approaches have been proposed for speeding up remote calls by splitting up their object parameters (i.e., sending to the server only the fields used by remote calls). As an example, the D [12] system uses a language framework approach, and the Doorastha [5] system uses special code-level annotations. Yet these approaches are special-purpose and attempt to solve only this particular instance of the problem. Consequently, special-purpose tools such as D and Doorastha are rarely used in practice: programmers avoid introducing extra dependencies just to address a single optimization task. Finally, since Java remote calls [18] use serialization [17] for passing complex object structures between different address spaces, the `transient` keyword can be used to indicate a field that is not part of an object's persistent state and should not be serialized. This solution works well to reduce network communication, but still introduces some overhead. (See also our experiments in Section 5.2.) As a result, programmers still largely perform this optimization manually by splitting their classes in the source code. Using binary refactoring would achieve the same performance benefits, but without making any changes to the source code.

3. REFACTORING CATALOGUE

In this section, we present a collection of common binary refactorings. To begin, we should ask what constitutes a binary refactoring. Not all regular software refactoring transformations are suitable to be binary refactorings. Most regular refactorings deal explicitly with source code entities and modify an application's design and structure. For instance, the majority of refinements in Fowler's catalogue [8] (e.g., "pull up field", "push down method", "hide method", "introduce assertion", etc.) concern source code modifications that by definition affect the subsequent maintenance of the application.

In contrast, a binary refactoring is not reflected in the source code. Despite the name, binary refactoring may not really occur at the binary code level, and in fact low-level binary representations are most likely unsuitable for binary refactoring transformations. Nevertheless, we use the term "binary" to signify that the refactoring is not applied as part of the programming process, but as part of the process of building/compiling the application. Binary refactoring could

even apply at the source code level, as long as it is invisible to the programmer—i.e., it does not affect the *maintained version* of the source code. Consequently, the refactoring is done to improve performance and not source code structure. Binary refactorings are semantics-preserving, but typically only under strict assumptions that are to be ensured by the user. These assumptions may be violated while the application source code evolves—the programmer is responsible for maintaining a *refactoring specification* together with the source code.

Our subsequent list of binary refactorings is meant to be representative, rather than exhaustive. We describe five refactorings in detail and outline a few more. We distinguish two main categories. These are *refactorings to improve locality* (such as "split class" and "glue classes") and refactorings to remove indirection (such as "inline method," "remove delegate," and "remove visitor"). Whenever a binary refactoring corresponds to a well-known source refactoring, we keep the established name.

3.1 Structural Refactorings for Locality

3.1.1 Split Class.

General Description.

Split a single class into multiple partition classes, preserving the functionality of the original class, possibly only under a specific application scenario. Make all the clients of the original class use the partition classes instead.

Purpose and Applicability.

The *split class* refactoring provides a systematic mechanism for modifying the internal representation of objects by partitioning them into distinct entities that exploit locality, thereby improving performance. This refactoring is applicable primarily in the application scenarios in which one part of the object needs to be stored or transmitted independently of the rest. Additionally, certain storage and network transmission mechanisms can provide better performance when handling an object in its entirety rather than some subset of its fields. Splitting an object provides a set of smaller objects that can be used by such mechanisms. Another application for splitting classes is improving cache locality. Consider a class with some frequently accessed fields and some infrequently accessed ones. By breaking up objects, we allow fields of different partitions to be allocated non-contiguously. As a result, a good allocation policy (e.g., one clustering together objects of the same size) will place related fields of different objects in the same caching unit (e.g., processor cache line or virtual memory page) thus allowing much tighter packing of the frequently accessed fields.

Specifics and Variations.

Conceptually, the process of splitting a class is quite straightforward and involves two steps. First, replace all the instances of creating an object of the original class with creating the corresponding objects of its partition classes. Second, make all the clients of the original class use the partition classes instead.

In general, because split class is applicable in a variety of application scenarios, it comes in many different flavors and variations, defined primarily by how the rest of the application accesses the partition classes and how they access each other. We find it useful to differentiate the splits based on direction, serializability, and security attributes.

The *direction* attribute defines how the partition classes access each other. A *unidirectional* split identifies "the primary partition" class, using it instead of the original class and accessing all other partitions through it. The primary partition class also contains all of the methods of the original class. By contrast, a *multidirectional* split treats all the partitions uniformly, enabling them to access each other and also to contain any subset of the methods of the original class. (The multidirectional split is generally not a semantics-preserving transformation: an object no longer has a unique identity as different parts of the code can be directly accessing different partitions. The user is responsible for ensuring that the refactoring can be applied.) The links between partitions can be implemented either intrusively, with partition classes containing references to each other as data members, or externally, using external mapping data structures. Additionally, intrusive partition references can be optionally marked as `transient`, allowing partitions to be serialized individually, thus implementing the *serializability* attribute.

The *security* attribute defines whether the encapsulation properties of the original class are preserved in the implementation of the partition classes. Because only the class itself can access its `private` fields, this invariant changes to include all other partition classes as well, after the split takes place. In a safe language like Java that checks access rights at run-time, implementing the desired security semantics requires a creative solution. For instance, in our implementation we use a solution similar to that proposed by Bhowmik and Pugh [4] for the Java inner classes rewrite. At load time, one partition class obtains a secret key and passes it to the other partition class. When objects of one partition class need to access fields from the other, they call a public method that also receives and checks the secret key.

3.1.2 Glue Classes

General Description.

Merge two classes `FrontEnd` and `BackEnd` whose objects are always in one-to-one correspondence into a single class. The refactoring is applicable when each object of class `BackEnd` is accessed only by a single object of class `FrontEnd`.

Purpose and Applicability.

By merging two closely related classes that are always used together, we get the advantage of removing access indirection and, more importantly, merging member data for better locality. For instance, the refactoring is typically applicable when the source code design employs the *Bridge* design pattern. The Bridge design pattern "decouples an abstraction from its implementation so that the two can vary independently" [9]. Nevertheless, often the abstraction has a single implementation, known statically. The implementation may be fixed for the current version (yet the design pattern is used for future extensibility) or the implementation may be fixed for a certain hardware platform or build configuration. The canonical example [9] demonstrating the Bridge pattern is one of a window abstraction with two possible flavors (`IconWindow`/`TransientWindow`) and two possible

implementations for different windowing toolkits. Yet, the windowing toolkits do not change during application runtime, although the flexibility to switch toolkits is desirable in the application source code. The *glue classes* refactoring can be used to merge the current window implementation with each abstraction flavor, in order to obtain performance benefits without changing the source code.

Specifics and Variations.

The resulting class has the union of the data members and methods of both glued classes. The client interface remains that of class `FrontEnd`, which is accessed by clients as before. The construction interface of `FrontEnd` expands to accept any `BackEnd` construction parameters. Clients should construct objects of the two classes using the common constructor chaining idiom:

```
new FrontEnd(..., new BackEnd([params]), ...)
```

This gets replaced with a single call to the expanded `FrontEnd` constructor. The `BackEnd` constructor is inlined (see the *inline method* refactoring) for correct initialization. Methods of class `BackEnd` can also be inlined/devirtualized at their call sites inside class `FrontEnd`—the assumptions of the *glue classes* refactoring ensure that the target of the method call is statically known.

3.2 Refactorings to Remove Indirection

3.2.1 Inline/Devirtualize Method

General Description.

a. Replace an indirect (virtual) method call with a direct (static) one.

or

b. Replace a method call with the adapted body of the callee (inlining).

Purpose and Applicability.

An indirect (dynamic) dispatch is costlier than a direct (static) one. The true purpose of an indirect dispatch is to enable polymorphic behavior. However, when no polymorphism is present, an indirect dispatch can be replaced with a direct one or the call target can be inlined. Although standard compiler optimizations (e.g., inline caching [6]) are very effective in eliminating the double indirection cost, they still do not usually attain the performance of inlined code (see our later experiments) and they can benefit from user-supplied information on what methods to devirtualize/inline.

Clearly the refactoring is low-level and should be applied very sparingly. Nevertheless, several common coding patterns can benefit from the inlining/devirtualization refactoring. Object-oriented software development promotes a coding style that results in the proliferation of a large number of very small methods. Take, for example, the practice of declaring all member fields in a class to be private and providing a pair of accessor/mutator methods as unique access points for each field. These methods simply return or modify the value of their respective field but result in an indirect dispatch. When the dynamic type of an object is known statically, it is beneficial to inline accessor and mutator method calls. (Note that sophisticated optimizing compilers do perform this optimization but only under clear enabling conditions—e.g., when the method is never overridden.)

Specifics and Variations.

Inlining has the standard trade-off of expanding the code size and reducing locality. Thus, too aggressive inlining can actually hurt performance.

In Java, static methods are applied to classes, not objects. Thus, the static method resulting from devirtualization needs to accept an extra argument: the object the method code will act on.

If the refactoring is applied above the level of a secure runtime, such as the Java VM, inlining code will require weakening the encapsulation properties of the target class. For instance, private fields will need to be made public to be accessed directly by external code.

3.2.2 Remove Delegate

General Description.

Change client code so that instead of calling a delegate object, it calls the target object directly. The refactoring is applicable when there are multiple clients per delegate and the clients statically know which delegate is getting called, but not necessarily what is the target object.

Purpose and Applicability.

This refactoring is similar to *glue classes*. It differs, however, in its purpose and mechanism. The purpose of *remove delegate* is to remove indirection rather than to enhance locality. The delegate class is not glued with the client class because there are multiple client classes, all accessing the same delegate. Instead the delegate object still exists but its fields are accessed directly inside client code (the *inline/devirtualize method* refactoring is applied to all calls of delegate methods inside all refactored clients in turn).

A common reason to apply *remove delegate* is when the code is structured using design patterns such as Proxy, Adapter, Bridge, Composite, etc. These design patterns introduce an extra object between the client and the implementation of a concept. Typically, the extra level of indirection is necessary because there are two degrees of run-time variability. Clients do not statically know which delegate they get (all delegates support the same abstract interface) and delegates do not statically know which target object they call. Nevertheless, occasionally the dynamic type of a delegate is statically known in the current version of the application or in the current build configuration or by some of the clients but not all. In all these cases, it is beneficial to eliminate the overhead of the delegate using binary refactoring.

3.2.3 Remove Visitor

General Description

Eliminate the overheads resulting from the use of the Visitor design pattern [9]. The refactoring is applicable when the types of visited objects and visitors are known statically. This is an example of a composite refactoring that the programmer can

express concisely because of the underlying design pattern and effect its implementation by automatically determining and applying a series of simpler (e.g., *devirtualize and inline method)* refactorings.

Purpose and Applicability

A common object-oriented solution to the problem of providing an extensible machinery for expressing operations on the elements of an object structure without modifying their source code is the Visitor design pattern [9]. That is, Visitor enables the programmer to express a new operation without changing the code of the classes on which it operates. The key to extensibility is the use of polymorphism in the implementation of the pattern, which references both visited objects and visitors through their respective superclasses. This results in a complex double dispatching structure that usually hinders performance. In the cases when the types of the visited classes and the visitors are known statically, the runtime overhead of double dispatching can be eliminated by replacing indirect calls with direct ones or by inlining the visitors' code in the visited classes.

Specifics and Variations

Traditional implementations of the Visitor design pattern come in several different flavors, with some of them being more amenable to binary refactoring than others. For example, programmers can choose multiple approaches to traversing the visited objects. One of the options is to have the calling program keep all the visited objects in a simple structure such as an array, passing the visitor to each of them in a loop. A more common approach though, particularly for Composite [9] structures, is for the visited objects to provide their own traversal strategy by calling the `accept` methods on the objects that they contain.

How much of the overhead of double dispatching can be eliminated depends on whether polymorphism is used in dispatching the `accept` methods of the visited objects. If the exact type of all the visited objects is known statically, the code in the `visit` methods can be inlined in the visited objects, thereby completely eliminating all the overhead of the double dispatching in the pattern's implementation.

More commonly in practice, the type of the visitor is known statically, yet the type of the visited objects truly varies polymorphically. In this case, only half of the indirection overhead can be removed. For instance, in a compiler implementation, we may know exactly when a `TypeCheck` visitor is applied to the root of a syntax tree. Once a `TypeCheck` visitor is used on the root element type, it will only pass the `TypeCheck` visitor to components of the syntax tree. In this case, binary refactoring can replace the generic `accept` methods with specialized versions for each visitor and devirtualize the `visit` methods in the visitor classes.

3.3 Other Binary Refactorings

Many more optimizations can be recast as binary refactorings. The ideal candidate optimization is one that cannot be performed well without user-supplied knowledge and affects the application structurally—i.e., it is reflected at the level of the interface between classes, rather than being entirely internal to a class.

Thus, for instance, refactorings can be introduced for removing the indirection overhead of many design patterns, when the target of an indirect call is known statically. For example, in addition to the Visitor design pattern, the Abstract Factory is usually one in which indirection can be removed. (Gamma et al. [9] observe: "Normally a single instance of a ConcreteFactory class is created at run-time.") Note that in all these cases, keeping the design pattern is advantageous for the long-term maintainability and extensibility of the application (e.g., the application may need to support multiple windowing systems) yet in any particular build configuration the actual type of objects involved in the pattern is statically known.

Many low-level optimizations can also be presented as binary refactorings. (Nevertheless, the lower the level of the optimization, the more sparingly it should be applied.) For instance, a possible refinement would be "replace polymorphism with conditional", replacing polymorphic calls with a dynamic check of the type of the dispatch object and an inline execution of the potential call targets. This parallels the polymorphic inline caching optimization [10] in traditional compilers, but with user-supplied information that guide the transformation.

4. DISCUSSION

"Rules of Optimization: Rule 1: Don't do it. Rule 2 (for experts only): Don't do it yet."
M.A. Jackson

"More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason — including blind stupidity."
W.A. Wulf

We are all aware of the stern warnings against being too eager to optimize software. Optimization is often responsible for obfuscating programs, limiting their maintainability, and hindering their generality. Nevertheless, binary refactoring is orthogonal to the question of *when* to optimize. An optimization approach using binary refactoring does not advocate more optimization. Instead, binary refactoring dictates that, when optimization is necessary, it is often best done without polluting the application source code. In this sense, binary refactoring has a certain aspect-oriented flavor: it advocates expressing aspects of an application that are traditionally encoded in source code separately and then composing them back with the application behind the scenes. At first, this optimization approach may seem dangerous: assumptions that affect the correctness of the resulting program are expressed separately from the source code. Thus, subtle errors can be introduced and inspection of the source code is not sufficient to explain application behavior when an erroneous refactoring is applied. Debugging may be similarly hindered—a common pitfall of automatic program transformations.

Nevertheless, we believe that binary refactoring is strictly superior to the alternatives, *as long as it is not viewed as an excuse to apply optimization more eagerly or less carefully*. Without binary refactoring, the same assumptions would be reflected in the application source code structure. Furthermore, the assumptions would be neither explicit nor localized. Instead, binary refactoring allows the concise expression of static knowledge of the application structure as a separately

maintainable entity: the refactoring specification. When assumptions change, the refactoring specification should be updated to reflect them. This change is significantly easier than editing the source code. For instance, undoing a *glue classes* refactoring by editing the refactoring specification is much easier than separating two classes and changing their clients manually.

The explicit, programmer-guided nature of binary refactoring also alleviates issues relating to debugging. Debugging should occur at the level of the maintained application source code. If an error is introduced only due to the use of a binary refactoring transformation, then the programmer needs to inspect the refactoring specification and re-examine whether the correctness assumptions are satisfied. Even without explicit debugging support for binary refactoring, the problem is manageable. The refactoring specification should be short and relatively easy to inspect. Furthermore, the programmer explicitly introduces the new behavior and will not be surprised by it.

5. BINARY REFACTORING BROWSER (BARBER)

We demonstrate binary refactoring concretely with BARBER (the Binary Application Refactoring BrowsER), our reference implementation of a binary refactoring browser for Java. BARBER "grooms" existing programs for optimized execution. We first describe the structure and some implementation specifics of BARBER, and then showcase the effects of binary refactoring through a series of small and large benchmarks. BARBER can be downloaded from `http://j-orchestra.org/barber` .)

5.1 Structure

Our intention was to make the structure of BARBER straightforward in order to maximize usability. BARBER is an extensible framework in which binary refactorings can be added and share useful support functionality. Currently the system implements several variants of the *split class*, *remove delegate*, *remove visitor*, and *inline/devirtualize method* refactorings. Figure 1 shows the user view of BARBER schematically. Because binary refactoring can be thought of as a new activity in the software development chain, we have provided support for using BARBER as a task in the popular ANT build tool [2] for Java. Specifically, the BARBER task is a separate build step that runs after all the standard build steps have been completed. In addition, depending on the target environment for a given build configuration, different sets of binary refactorings can be applied to the same source code base.

As input, BARBER accepts a configuration file that specifies a collection of binary refactorings in XML format. This input file contains all the necessary parameters for the specified refactorings such as the file system location of the original application classes. The type of a given binary refactoring determines the exact set of input parameters that a configuration file must provide. For example, Figure 2 shows a fragment of a BARBER XML file that specifies the parameters of a *split class* binary refactoring. The fragment starts by describing the refactoring type along with its direction, security, and transiency attributes, which indicate that this particular *split class* is unidirectional, secure, and serializable, respectively. (For a detailed explanation of each of these

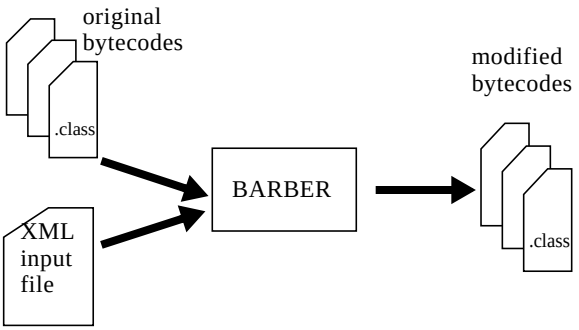

Figure 1. Using BARBER

attributes see Section 3.1.1.) After that, the `CLASS_TO_SPLIT` tag provides a fully-qualified name of the class to be split as `acc.Customer`. The member fields' names that follow next are placed by convention into the primary partition of the split. Since member fields of the same class have unique names in Java, providing only the fields' names is sufficient. The compilation unit definition, containing all the classes that participate in this refactoring, completes the configuration fragment. Since listing all the classes in a large compilation unit individually can be a tedious task, we are currently working on providing support for defining groups of classes by the means of wildcard patterns, similar to the ones of directory-based tasks in ANT [2].

After BARBER finishes parsing the input XML file, all subsequent operations are performed at the bytecode level. Before effecting the refactorings specified by a given configuration, BARBER performs some syntactic checking of the input application classes to ensure that the refactorings would make sense. (Currently such checks are limited to checking whether all entities exist and have an appropriate type. An interesting future work direction would be to try to identify complex enabling conditions, such as uniqueness of reference, or patterns of instantiation and use.) If the check is successful, BARBER then applies the refactorings, producing a modified set of binary Java classes.

Currently we are exploring the possibilities of offering BARBER as an Eclipse [7] plug-in. Being able to specify binary refactorings through a GUI-based tool would not only completely eliminate the need for editing XML files by hand, but would also provide greater flexibility in integrating the technique into the development process. Furthermore, operating within the context of an IDE would make it easier to provide additional support for the programmer, such as automatically discovering beneficial binary refactorings and verifying preconditions for their successful application via static and dynamic analyses. Finally, having integrated BARBER into a popular IDE such as Eclipse would facilitate the task of exploring the benefits of the technique in medium to large software projects via case studies.

We have found BARBER very appealing in our own software development work. It enables an interesting level of flexibility in fine-tuning applications. We employed BARBER in the context of developing J-Orchestra [19], an automatic partitioning system for Java programs. By using BARBER, we have successfully improved the performance both of some of the J-Orchestra runtime components and of the binary Java

```
...
<REFACTORING>
 <SPLIT_CLASS DIRECTION_TYPE="UNIDIRECTIONAL"
              SECURITY_TYPE="SECURE"
              TRANSIENCY_TYPE="SERIALIZABLE">
  <CLASS_TO_SPLIT>acc.Customer</CLASS_TO_SPLIT>
  <FIELD_TO_SPLIT>_id</FIELD_TO_SPLIT>
  <FIELD_TO_SPLIT>_zipCode</FIELD_TO_SPLIT>
  <COMPILATION_UNIT>
   <CLASS_NAME>acc.Customer</CLASS_NAME>
   <CLASS_NAME>acc.Invoice</CLASS_NAME>
   <CLASS_NAME>acc.InvoiceInfo</CLASS_NAME>
   <CLASS_NAME>acc.Main</CLASS_NAME>
  </COMPILATION_UNIT>
 </SPLIT_CLASS>
</REFACTORING>
```

Figure 2. BARBER XML Input File for *Split Class*

classes that J-Orchestra generates. We quantify some of these improvements in one of the benchmarks in Section 5.2.

5.2 Experiments

We have used BARBER on a series of micro and macro benchmarks. The measurements are on a 2.4GHz Pentium 4 machine, running Sun JDK 1.4. Note that the experiments described aim at demonstrating, rather than validating, binary refactoring. Since our main argument is one of usability, it cannot be validated or refuted with program performance measurements. Nevertheless, these measurements are interesting for quantifying the effect of some binary refactoring transformations in specific settings.

Our first microbenchmark demonstrates the benefits of the *split class* refactoring in the example discussed in Section 2: reducing the amount of data transferred across different address spaces in a remote method call. The scenario consists of an instance of class `Customer` passed to a remote method call that uses only some of the fields of the class. Specifically, the remote method invocation uses the Customer's id and postal code to determine a shipping rate. To make the scenario more realistic, we used an actual `Customer` class from the open source JFreeReport Library (`jfree.org/jfreereport`). Among the trivial changes that we made to the class prior to using it in our benchmark was adding a `long` id field to the existing seven `String` fields, and marking the class as `Serializable`.

The *split class* binary refactoring for this experiment was effected by instructing BARBER to split `Customer` into two partitions in such a way that one of them would contain only the two fields used by the remote method call with the other one containing the rest of the fields. In this way, the first partition could be used in the remote method call, significantly reducing the amount of transferred data. To isolate the improvements due to reduced serialization from those due to reduced network communication time, we placed the client and the server on the same machine. Table 1 shows the results of running the original and optimized versions of the benchmark. (Each remote call is repeated 10^5 times. In all experiments—except for the SPEC JVM 98 raytracer benchmark, which has its own test settings—we "warm" the JVM with the tested code but different inputs before measuring, to ensure that the code is compiled.)

Table 1. Speeding up remote method calls.

Baseline (ms)	Split class (ms)	Speedup
27062	22984	1.18

A reader familiar with the specifics of Java Serialization might wonder if using the `transient` keyword would not be a better way to avoid transferring unused data in a remote call. Since Java remote calls [18] use serialization [17] for passing complex object structures between different address spaces, the `transient` keyword can be used to indicate a field that is not part of an object's persistent state and should not be serialized. While marking fields of a class as transient is one of the binary refactorings supported by BARBER, using "split class" has the advantage that the fields are not even examined during serialization. This is important in configurations where the bottleneck is not the transfer time of the data but the computational overhead of serialization. (This is actually the case in a common server-side execution scenario: when the serialization takes place between two different address spaces that exist on the same machine.) As a simple example, we have used a class `Record` consisting of a hundred integer fields, out of which only every tenth field needs to be serialized. Using a class with that many fields might seem a little contrived. However, if we consider that Java Serialization involves an exhaustive traversal of an entire object graph, serializing even a simple class that has object fields and/or has superclasses could easily involve going through as many and more primitive fields. We created 50K objects of class `Record` and measured the total time it takes to write them to a disk file and read them back into memory. Table 2 shows that transient fields still impose a significant computation overhead.

Table 2. Speeding up serialization.

Using "transient" fields (ms)	Using "split class" (ms)	Speedup
1218	1078	1.13

Another area in which "split class" can be beneficial is in exploiting locality. Consider a "number-crunching" application that operates on a collection of records, each of which containing ten integer fields, with only one of the fields used by a computation. A simple benchmark program is the following:

```
class NumericRecord {
 int _field0;
 int _field1;
 ...
 int _field10;
}
...
NumericRecord[] records; //declared somewhere
for (int i = 1; i < numRecords; ++i) {
 records[i-1]._field0 += records[i]._field0;
}
```

If the `for` loop presented above is repeated several times, its performance would be greatly affected by the effectiveness of

the processor's cache. Splitting the record in such a way so that it would contain only the fields used in the repeated computation could allow more records to fit in the cache (allocators tend to cluster objects of the same type or size) thereby improving the overall performance. In this microbenchmark, we created a 100K records and repeated the `for` loop five times. The split created a partition containing a single filed `_field0`. Table 3 shows the results. Even though this optimization is quite low-level, it can have a significant pay off for the sophisticated programmer.

Table 3. Improving caching performance.

Baseline (ms)	"Split class" (ms)	Speedup
47	16	2.93

One of the most-widely used and well-understood optimizations in the arsenal of optimizing compilers and, recently, JIT compilers has been method inlining and "devirtualization." The first optimization refers to replacing a method call in the body of the caller with the actual code of the callee, and the second one replaces virtual and interface method calls that are dispatched indirectly with direct calls. Despite the common wisdom stating that these optimizations should not be applied by hand, in some well-known cases, hand-annotation can achieve performance not reached automatically. We have applied a combination of the aforementioned refactorings to the multithreaded raytracer benchmark from the SpecJVM'98 suite. We ran this benchmark under both Sun's JDK 1.3 and JDK 1.4, to see the effect that a more sophisticated JIT technology has on low-level optimizations. Table 4 and Table 5 show the results.

Table 4. Speeding up SpecJVM Raytracer in JDK 1.3

Base line (ms)	Inline method (ms)	Inline method speed-up	Inline method + Devirt. method (ms)	Inline method + Devirt. method speedup
3,782	3,141	1.20	3,031	1.25

Table 5. Speeding up SpecJVM Raytracer in JDK 1.4

Base line (ms)	Inline method (ms)	Inline method speedup	Inline method + Devirt. method (ms)	Inline method + Devirt. method speedup
2,781	2,703	1.03	no further benefit	n/a

There are two elements worth pointing out in this benchmark. First, the optimal points were reached for slightly different configurations in the two JDK platforms. Furthermore, the best performance for JDK 1.3 was achieved with a combination of inlining and devirtualization. 14 methods in 4 different classes were inlined but 3 other methods were just converted to static methods without inlining, to avoid code blowup. The second interesting (although hardly surprising) point is that under JDK 1.4 the improvement is small: a ~3% speedup for inlining methods. Trying to improve the performance with a configuration mixing inlining and devirtualization did not yield any fruit. An improvement of 3% is significant for automatic optimizations, but it is a low payoff for an optimization activity that requires programmer intervention, and as a result is brittle with respect to changes in the program and the runtime system. This seems to be a clear argument in favor of letting low-level optimizations be determined entirely by sophisticated compilers, rather than through programmer annotations. Nevertheless, note that less sophisticated JVMs, not featuring any advanced JIT compilation capabilities, could still greatly benefit from low-level refactorings. Such JVMs are often used in environments with limited processing power (e.g., embedded systems). This is exactly a domain in which many "optimization sins" (of the form described in Section 4) are routinely committed in software development!

To measure the benefits of the *remove visitor* refactoring, we created a composite structure, consisting of a class `Node` that implements `Visitable` interface and contains a `List` of `Visitable` objects, each of which could be either a `Leaf` or another `Node`. The `accept` method in `Node` implements the object traversal strategy in this composite structure.

```
class Visitable {
 abstract void accept (Visitor visitor);
}

class Leaf implements Visitable {...}

class Node implements Visitable {
 //in JDK 1.5 one could use generics
 List /*Visitable*/ _elems;

 void accept (Visitor visitor) {
  visitor.visit (this);
  for(Iterator it = _elems.iterator();
                      _elems.hasNext();) {
   Visitable visitable = (Visitable)it.next();
   visitable.accept (visitor);
  }
 }...
}
```

We have also created a factory class that generates a pseudo-random composite structure after being parameterized with a random seed, the composite's depth and breadth, and the percentage of leaves. We measure the total time it takes for two different visitors to visit the generated structure. (The numbers in Table 6 are for a depth of 26, breadth of 4 and 72% leaves but are representative of other configurations as well.)

Table 6. Improving caching performance.

Baseline (ms)	"Remove Visitor" (ms)	Speedup
7016	6593	1.06

This benchmark reflects the common case, in which the type of visitor is known statically, but the type of visitable objects is not: exact types of objects in the `List` of `Visitables` in class `Node` vary at run-time. Therefore, the refactoring could safely remove only one level of indirection. The benefits would be even more pronounced if all the types of the visited objects were known statically.

In our final benchmark, we measure the benefits of using binary refactoring in the context of J-Orchestra [19], our automatic partitioning system for Java programs. J-Orchestra rewrites the bytecodes of a Java program running on a single machine, to convert it into a distributed program, running across multiple machines. To enable remote execution, J-Orchestra heavily utilizes the Proxy pattern [9], generating proxy classes in source code form and then compiling them into bytecode. Proxies hide the actual location of objects, enabling both local and remote execution. In some cases (especially when dealing with system classes) the complications of working with standard distribution middleware requires applying the Proxy pattern multiple times. That is, a client holds an instance of a proxy to a proxy and each method call results in a double delegation. This incurs significant performance overhead when a proxy's holder and its destination are co-located in the same address space. Then the proxy indirection can be removed using the "remove delegate" binary refactoring. In principle, J-Orchestra could output the specialized code in the first place. Nevertheless, the proxies are generated in source code form while the target classes are in bytecode form. Hence, although the optimization could be integrated in J-Orchestra instead of using BARBER, the implementation would need to be a bytecode transform, essentially identical to the BARBER refactoring. In our microbenchmarks on the overhead of the J-Orchestra indirection, the effect of this optimization ranged a lot (depending on the amount of work performed per method and the opportunities for JIT optimization) but the refactoring speedup ranged from none to 1.20 for different methods.

6. RELATED WORK

Binary refactoring is related both to regular software refactoring and to annotation-guided compiler optimization. These are large and diverse research areas, so our presentation is limited to closely related or representative work.

Binary refactoring is not the first attempt of providing higher-level optimizations for object-oriented programs. In an approach closely related to ours, Tourwe and De Meuter [22][23] use an open compiler for removing higher-level design abstraction by performing architectural transformations based on programmer-supplied annotations. In their work, they concentrate on removing the overheads resulting from dynamic dispatch in common design patterns, such as Visitor [9]. The approach uses logic-based transformation and declaration annotation languages to parameterize a code generator that transforms higher-level design patterns into more efficient representations. Our work differs in the presentation (we cast the problem in a refactoring framework); in the application (we use a refactoring browser tool and XML specifications instead of a logic-based transformation language); and in the concrete transformations (we identify and evaluate quite different and more complex refactorings—Tourwe and De Meuter concentrate on the Visitor pattern). In terms of methodology, our approach is black-box, while that of Tourwe and De Meuter is more white-box: we do not assume that the programmer will write transformations but that these will be supplied ready-made in a binary refactoring browser.

A similarly closely related line of work is that of automatic program specialization [15]. Automatic specialization is a form of partial evaluation that accepts programmer-supplied invariants and propagates them throughout the program in an attempt to specialize code with respect to known types and values. The exact arguments used in favor of automatic program specialization over standard compiler optimizations can also be used in favor of binary refactoring. Nevertheless, compared to automatic specialization, our binary refactoring work also identifies higher-level transformations that affect the class hierarchy directly, such as *split class* and *glue classes*. Furthermore, by drawing the analogy to source-level refactoring, we establish a clear and familiar methodology regarding the use of binary refactoring in the software development process.

Much work has been done in the domain of just-in-time compilation to improve the performance of dynamic dispatch. Modern JVMs employ a sophisticated arsenal of compiler optimizations, and especially stress fast dynamic dispatch using traditional inline caching techniques [6][10] and variations suited to Java's dynamic loading model [1]. A common criticism of all speculative devirtualization optimizations, however, is that they never achieve the performance of static calls because they do not easily enable inlining and the subsequent intra-procedural optimizations that this entails. (Inline caching techniques need to guard the execution of the code with a conditional, in case the target object is not of the expected type.) Thus, devirtualize/inline binary refactorings can still improve performance relative to automatic techniques, and can also do so regardless of the JIT compilation capabilities of the JVM.

An overview of software refactoring can be found in a recent survey by Mens and Tourwe [13]. Both source-level and binary refactoring are not automatic but rather programmer-initiated and driven activities. That is, despite the success of several research projects aimed at providing program analysis tools for discovering and suggesting possible refactoring opportunities [16][17][21], it is always the programmer who decides which changes would improve the program and ensures that they would not alter its external behavior.

Binary refactoring can be loosely qualified as an aspect-oriented technique (i.e., treating the runtime performance of a program as a separate aspect). However, our refactorings are not supported by traditional AOP tools [11]. In fact, such tools do not allow the programmer any control over the implementation specifics of weaving aspects into a program. Furthermore, AOP is concerned with changing the behavior of a program by linking it with a cross-cutting concern. In contrast, when modifying code, binary refactoring aims at maintaining the original execution semantics of a program even though possibly only under specific assumptions.

7. CONCLUSIONS

We presented binary refactoring: a technique for introducing optimizing transformations in object-oriented programs without affecting the maintained version of the program source code. The argument for binary refactoring is one of usability: we believe that performing program optimizations using binary refactoring results in safer and more maintainable programs than manually changing the source code. Certainly, further work is needed to assess the benefits of binary refactoring in the software development process. Therefore, we hope that the idea of binary refactoring and our reference implementation will stimulate interest that will prompt us and others to continue this research further, hopefully evolving binary refactoring into a valuable addition to the working programmer's toolset.

8. ACKNOWLEDGMENTS

This research was supported by the NSF through grants CCR-0238289 and CCR-0220248, and by the Georgia Electronic Design Center.

9. REFERENCES

[1] B. Alpern, A. Cocchi, S. Fink, D. Grove, and D. Lieber, "Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless", in Proc. *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[2] The Apache ANT Project. On-line at `http://ant.apache.org/`.

[3] E. Casais. "Automatic reorganization of object-oriented hierarchies: A case study", *Object-Oriented Systems*, 1(2):95–115, 1994.

[4] A. Bhowmik and W. Pugh, "A Secure Implementation of Java Inner Classes", PLDI 99 poster session.

[5] M. Dahm, "Doorastha—a step towards distribution transparency", *JIT* 2000. See `http://www.inf.fu-berlin.de/~dahm/doorastha/` .

[6] L. P. Deutsch and A. M. Schiffman, "Efficient Implementation of the Smalltalk-80 System", *ACM Symposium on Principles of Programming Languages (POPL)*, 1984.

[7] The Eclipse Foundation. On-line at `http://www.eclipse.org`.

[8] M. Fowler, "Refactoring: Improving the Design of Existing Programs", Addison-Wesley, 1999.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[10] U. Hölzle, C. Chambers, and D. Ungar, "Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches", *European Conference on Object-Oriented Programming (ECOOP)*, 1991.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold, "An Overview of AspectJ", *European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[12] C. V. Lopes and G. Kiczales, "D: A Language Framework for Distributed Programming", PARC Technical report, February 97, SPL97-010 P9710047.

[13] T. Mens and T. Tourwe, "A Survey of Software Refactoring", IEEE Trans. on Software Engineering 30(2): 126-139 (2004).

[14] W.F. Opdyke, "Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks," PhD thesis, Univ. of Illinois at Urbana-Champaign, 1992.

[15] U. P. Schultz, J. L. Lawall, C. Consel, "Automatic program specialization for Java", ACM Trans. Program. Lang. Syst. 25(4): 452-499 (2003).

[16] M. Streckenbach, G. Snelting, "Refactoring Class Hierarchies with KABA", OOPSLA 2004.

[17] Sun Microsystems, Java Object Serialization Specification.

[18] Sun Microsystems, Remote Method Invocation Specification.

[19] E. Tilevich and Y. Smaragdakis, "J-Orchestra: Automatic Java Application Partitioning", *European Conference on Object-Oriented Programming (ECOOP)* 2002.

[20] F. Tip, A. Kiezun, and D. Baeumer. "Refactoring for generalization using type constraints", In *Proc. 18th* Conference on Object-Oriented Programming Systems, *Languages, and Applications (OOPSLA'03)*, pages 13–26, 2003.

[21] F. Tip, G. Snelting, and R. Johnson, "Program analysis for object-oriented evolution", Technical report, Dagstuhl Seminar Report 03091, 2003.

[22] T. Tourwe and W. De Meuter, "Optimizing Object-Oriented Languages Through Architectural Transformations", In *Proceedings of the 8th International Conference on Compiler Construction, pp 150-164, Springer-Verlag, 1999.*

[23] T. Tourwe and W. De Meuter, "An Open Compiler Using Meta-Level Information for Improving the Efficiency of Object-Oriented Programs", *OOPSLA 1998 Workshop on Reflective Programming in C++ and Java, Vancouver, Canada, 1998.*