Elimination of Redundant Polymorphism Queries in Object-Oriented Design Patterns

by

Rhodes Hart Fraser Brown
B.Sc., McGill University, 2000
M.Sc., McGill University, 2003

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy

in the Department of Computer Science
University of Victoria

Elimination of Redundant Polymorphism Queries in Object-Oriented Design Patterns

by

Rhodes Hart Fraser Brown
B.Sc., McGill University, 2000
M.Sc., McGill University, 2003

Supervisory Committee

Dr. R. Nigel Horspool, Supervisor
(Department of Computer Science)

Dr. Micaela Serra, Departmental Member
(Department of Computer Science)

Dr. Yvonne Coady, Departmental Member
(Department of Computer Science)

Dr. Kin F. Li, Outside Member
(Department of Electrical and Computer Engineering)

Supervisory Committee

Dr. R. Nigel Horspool, Supervisor
(Department of Computer Science)

Dr. Micaela Serra, Departmental Member
(Department of Computer Science)

Dr. Yvonne Coady, Departmental Member
(Department of Computer Science)

Dr. Kin F. Li, Outside Member
(Department of Electrical and Computer Engineering)

Abstract

This thesis presents an investigation of two new techniques for eliminating redundancy inherent in uses of dynamic polymorphism operations such as virtual dispatches and type tests. The novelty of both approaches derives from taking a subject-oriented perspective which considers multiple applications to the same run-time values, as opposed to previous site-oriented reductions which treat each operation independently. The first optimization (redundant polymorphism elimination – RPE) targets reuse over intraprocedural contexts, while the second (instance-specializing polymorphism elimination – ISPE) considers repeated uses of the same fields over the lifetime of individual object and class instances. In both cases, the specific formulations of the techniques are guided by a study of intentionally polymorphic constructions as seen in applications of common object-oriented design patterns. The techniques are implemented in Jikes RVM for the dynamic polymorphism operations supported by the Java programming language, namely virtual and interface dispatching, type tests, and type casts. In studying the complexities of Jikes RVM's adaptive optimization system and run-time environment, an improved evaluation methodology is derived for characterizing the performance of adaptive just-in-time compilation strategies. This methodology is applied to demonstrate that the proposed optimization techniques yield several significant improvements when applied to the DaCapo benchmarks. Moreover, dramatic improvements are observed for two programs designed to highlight the costs of redundant polymorphism. In the case of the intraprocedural RPE technique, a speed up of 14% is obtained for a program designed to focus on the costs of polymorphism in applications of the Iterator pattern. For the instance-specific technique, an improvement of 29% is obtained for a program designed to focus on the costs inherent in constructions similar to the Decorator pattern. Further analyses also point to several ways in which the results of this work may be used to complement and extend existing optimization techniques, and to provide clarification regarding the role of polymorphism in object-oriented design.

Contents

List of Tables

List of Figures

Acknowledgements

Many deserve recognition for efforts both large and seemingly inconsequential which have helped me to complete the work laid out in the following pages. Certainly, I must thank Nigel Horspool, my supervisor, for the sage advice and financial support he provided, and for tolerating my well-meaning, yet at times distracting and often petulant quest to become an educator. In addition, it was through this relationship that I became acquainted with Judith Bishop, and to her I owe the inspiration for the central theme of this thesis. In the latter part of 2006, she proffered a simple Visitor example which led me to a more expansive view of polymorphism and the role it plays design patterns and modern software development. Ultimately, much of the funding for this work was provided by grants from the Natural Sciences and Engineering Research Council of Canada. I also owe Micaela Serra for her kindhearted support throughout my studies, as well as the very practical guidance she offered me in organizing my research and structuring this document in particular.

As always, the value of the unwavering support of my parents has been tremendous. They have always given their backing to the merit of my intentions and expressed wholehearted confidence in my ability to achieve my dreams. They have been, and will always be, the foundation of my strength. I also wish to thank my office mates, Neil Burroughs and David Pereira, for being attentive and willing listeners to my many meandering rants. While they may not have realized it, these engaging explorations help to solidify my understanding of many technical and philosophical perspectives. Finally, to my wife Laura, I owe my will to persevere. She has stood by me patiently while I struggled to complete this work and has offered a comforting refuge during the most difficult times. Perhaps more than she knows, her earnest spirit has inspired me to clear this most daunting of academic hurdles and transformed me into a better person in the process.

For the open minds that will be shaped by the products of this labour.

Chapter 1
Introduction

Historically, efforts to improve the efficiency of software implementations have been dominated by a focus on the calculating aspects of program code. For the most part, the reoccurring questions have centered on how an understanding of the mathematical properties of operators can be combined with an understanding of their execution costs to identify situations where calculations can be simplified, replaced with previously computed results, or relocated to occur at more opportune times. However, a narrow emphasis on low-level calculations can only do so much. Recently, programming language implementors have taken to considering a wider view of optimization possibilities that incorporates both an understanding of source-level semantics, as realized in low-level code, and an appreciation for the statistical characteristics of execution behavior. These views have led to the development of significant advancements; for example tail call optimization, and speculative approaches to redundancy elimination. In many cases, these new approaches are predicated on insights regarding commonly applied programming idioms (patterns) related to a particular language feature, or programming style. And while such patterns may not always be precisely defined, they do provide a concrete empirical framework, allowing language implementors to identify the most relevant scenarios and devise new ways of optimizing their realization.

1.1 Eliminating Redundancy in Branching Operations

The study of redundancy in conditional branching operations provides a simple introductory example which points to a number of the ideas explored in this thesis.

It is commonly recognized that many of the conditional tests and branches performed by a program are patently redundant in the sense that their results are often directly correlated with, or determined by, the results of previous tests. To address these particular cases, several techniques for eliminating such "determined" branches based on data flow analyses and code duplication have been proposed [e.g., Mueller and Whalley, 1995; Bodík et al., 1997]. Although, so far, only a very limited set of predicating computations have been investigated.

Of course some branches—most, in fact—remain an essential part of program's logic and cannot be removed. Yet the cost of branching is not uniform: when executed on a modern pipelined architecture, a failure to anticipate the result of a conditional branch may incur a heavy penalty in instruction throughput. The problem for programmers and language implementors though, is that these costs are largely obscured in the code expression of a program. In other words, the actual

low-level actions performed to complete a branch operation are not apparent. There are, however, ways to mitigate these costs by combining an understanding of the mechanisms used to implement branching—in this case, how the processor anticipates the outcome of branches in the absence of previous information—with an understanding of how branches actually behave in common coding scenarios. The well-known study of Ball and Larus [1993] exemplifies this approach. In it, they identify seven simple code characteristics which can be used to better infer the outcome of branch operations. With this information in hand, programmers or compilers can arrange code so that the more likely path is actually the one anticipated by the processor during execution.

1.2 Redundancy in Polymorphism Queries

Polymorphism is a foundational programming abstraction which has become an essential feature in the design of object-oriented programs. Rather than decide explicitly how to proceed in manipulating data values, polymorphism allows the programmer to, in effect, let the program decide on the most appropriate action. In some cases (for example, expressions of ad hoc polymorphism), this choice may be resolved by the compiler, but it is also often performed through a *virtual dispatch*—a dynamic operation which may produce different behaviors based on the concrete type of the dispatch *receiver*. In this way, virtual dispatches, and other polymorphism operations such as type tests, represent a complex form of selection and branching logic. And similar to other uses of selection and branching, the common ways in which polymorphism is employed also lead to a significant amount of unnecessary overhead. A simple example of this redundancy is illustrated in figure 1.1.

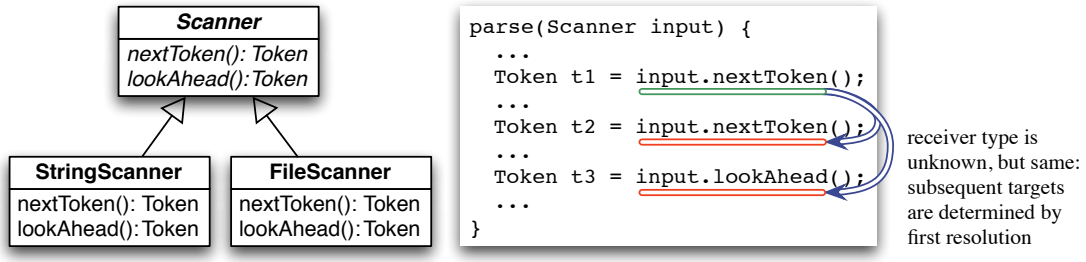
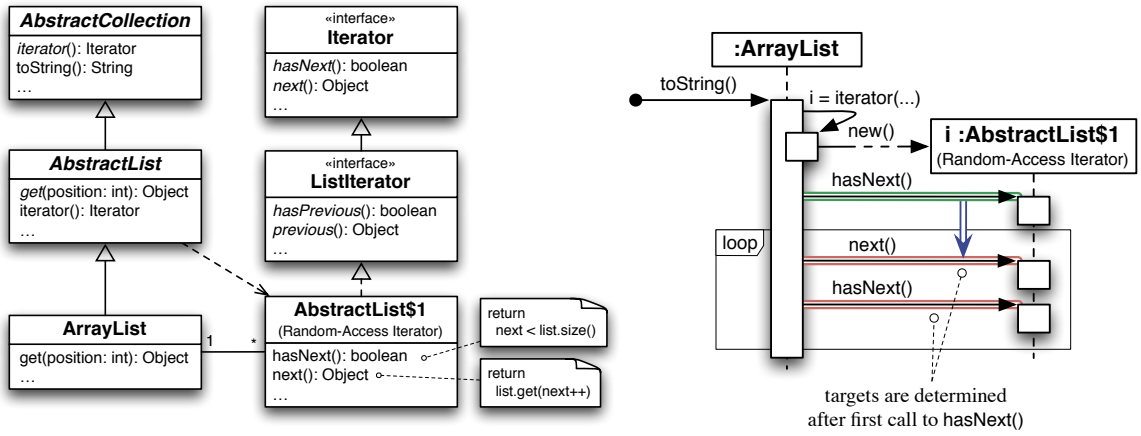


FIGURE 1.1: **Simple Dispatch Redundancy**

In this scenario, the specific actions performed by the `nextToken` and `lookAhead` methods are determined by the precise run-time type of the `input` object which, in this case, is obscured in the context of the `parse` method. Here, the parser has no knowledge of how the token values are actually obtained, just that the `input` object is somehow capable of producing them. This permits the parsing process to be generalized and reused over various input sources.

What is important to realize about this example, yet is often overlooked, is that each virtual dispatch applied to the `input` object entails more than just a call operation. The semantics of virtual dispatch imply that before the call is made, a query must be performed on the receiver to determine



(a) Structure (b) Interaction

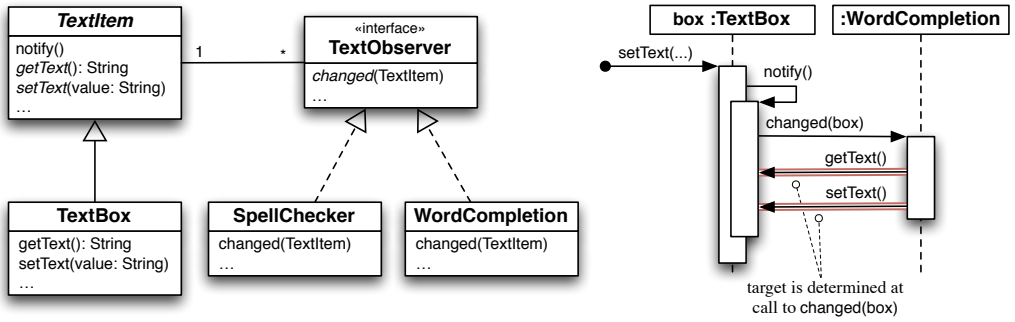
FIGURE 1.2: **Iterator - Intraprocedural Redundancy**

the target of the call—i.e., the address of the particular code implementation of the method. Note though that this process is completely hidden in the source representation. And even when compiled to certain intermediate forms, such as Java bytecode, the operations that comprise these queries are hidden, thus obscuring their ultimate costs. Yet in a typical implementation, these queries—which often entail several dependent load operations—are performed before every virtual call. However, a simple examination reveals that it is not always necessary to recompute the queries. In the given example, the targets of the second, and subsequent calls to `nextToken`, for any particular `input` value, are determined after the first query. Moreover, some partial results from this query may even be usable in simplifying the later resolution of the `lookAhead` target. The implications of this simple insight form the basis of the new polymorphism optimization techniques proposed in this thesis.

1.2.1 Contexts of Redundant Polymorphism

As demonstrated by the work of Ball and Larus, an important step in the process of identifying and eliminating redundancy is to develop a sense of the contexts in which the targeted redundant behavior may occur, and examine how recurrent idioms or coding patterns applied over these contexts may inform efforts to reduce the costs of this redundancy. In studying applications of polymorphism queries, four primary contexts of reuse emerge. Within each context, many common scenarios that exhibit eliminatable redundancy are exemplified by the uses of polymorphism in well-known object-oriented design patterns.

Similar to the example given in figure 1.1, applications of the ITERATOR pattern provide an example where the methods of a particular receiver, in this case the iterator object, are invoked repeatedly over the span of a single usage sequence. In figure 1.2, the first resolution of the `hasNext` method ultimately determines the target of all subsequent uses. Note, however, that the implemen-



(a) Structure (b) Interaction

FIGURE 1.3: **Observer - Interprocedural Redundancy**

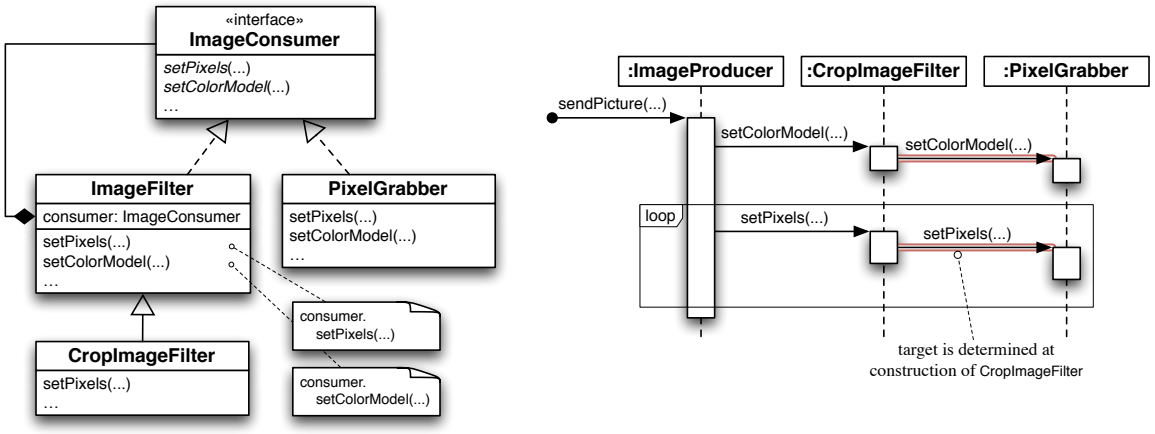
tation of the `toString` method is shared by all subclasses of `AbstractCollection`, each of which may produce its own concrete `Iterator` implementation. Thus, at the very least, the first virtual dispatch query is necessary in each separate application of `toString` since the `hasNext` and `next` targets may differ for each new use of an iterator value.

The interaction seen in applications of the OBSERVER pattern provides an example of polymorphism redundancy which crosses the boundary between method calls. The scenario in figure 1.3 illustrates how objects often pass a self-reference as a call argument in order to provide the callee with a handle that can be used to call back to the originator. Here, the exact targets of the `getText` and `setText` methods may not be known by the `WordCompletion` client, but they are certainly known to the `TextBox` when it initiates the sequence.

An example of polymorphism redundancy which is specific to the object-oriented paradigm is highlighted by the DECORATOR pattern (shown in figure 1.4), as well as other wrapper constructs. In this case, the inner field value of the `ImageFilter` is used repeatedly as a delegate. But since the delegation relationship is permanent for the life of the outer object, each resolution of the exact type and targets of the delegate are certainly redundant. However, this redundancy is specific to each `ImageFilter` instance since different filters may hold differently typed delegates. A similar form of redundancy occurs in applications of the SINGLETON pattern (figure 1.5), where the particular implementor of the singleton actions is fixed over the life of the module in which it is defined, in this case, a class. Here the static implementation of the class acts as a wrapper over the polymorphic delegate instance.

1.3 Simplified Thesis

This dissertation presents a study of two different program optimization techniques for eliminating redundant computations that arise from multiple applications of polymorphism queries to the same object. In general, the approaches are founded on the simple observation that all of the

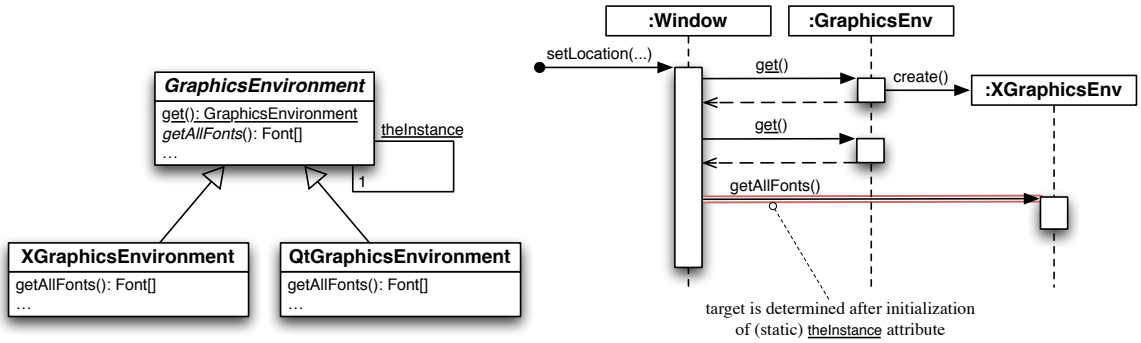


(a) Structure (b) Interaction

FIGURE 1.4: **Decorator - Instance-Lifetime Redundancy**

type and polymorphism data associated with a particular object are invariant for the lifetime of the object, at least for languages with invariant types such as C++, C#, and Java. Building on this insight, an adaptation of intraprocedural partial redundancy elimination (PRE) is developed for removing queries that are provably redundant over the span of a single method invocation, such as those highlighted in figure 1.1 and the ITERATOR example of figure 1.2. A second transformation is also developed to eliminate resolutions of polymorphism between instances with a fixed coupling, as in the DECORATOR (figure 1.4) and SINGLETON (figure 1.5) examples. The perspective that unifies these two approaches is a realization that some uses of polymorphism cannot be entirely eliminated from the code, even given strong whole-program type resolutions, since they are applied with the specific intention of creating polymorphic architectures—and yet, once a polymorphism query has been applied to a specific value (termed the "subject" in later discussions), repetitions of the same, or similar queries applied to the same subject are necessarily redundant and can be eliminated. This *subject-oriented* emphasis represents a departure from previous approaches to improving the efficiency of polymorphism in that it accepts that some queries cannot be removed, but their results can be retained to avoid the need to perform subsequent queries.

In considering the specific applicability of the subject-oriented approach, a theme which is revisited several times throughout the investigations of this thesis is how the intentionally polymorphic constructions exemplified by common object-oriented design patterns can be used to focus the proposed techniques on the characteristics specific to examples that are not likely to be reducible using other techniques.



(a) Structure (b) Interaction

FIGURE 1.5: **Singleton - Module-Lifetime Redundancy**

1.4 Document Organization

To capture the full scope of the research conducted in compiling this thesis, the presentation is divided into a sequence of two preparatory discussions followed by expositions of each of the two proposed optimization techniques—intraprocedural and instance-specific redundant polymorphism elimination (RPE). First, a history of the implementation and application of polymorphism in object-oriented programming is reviewed in chapter 2, and an analysis of the disconnect between modern design perspectives and implementation strategies is used to develop a more in-depth articulation of the primary thesis. Following this, chapter 3 provides an investigation of the research environment used to implement and evaluate the proposed transforms. This discussion examines a number of characteristics of the studied benchmarks (the DaCapo and SPECjvm98 suites), and reveals some challenges inherent in working with Jikes RVM—which is used as the compilation and execution platform. These insights are used to formulate a rigorous performance evaluation methodology appropriate for the later experiments, and to develop some predictions regarding the specific amenability of each the benchmarks to the proposed optimizations. With the motivational, theoretical, and practical foundations in mind, chapter 4 then describes the first of the two main optimization techniques, elaborating on aspects such as the need for a speculative code motion strategy, and the issues involved in performing such relocations safely. Performance evaluations of several different configurations of the transform are presented to show the utility of the approach and underscore the value of the particular implementation choices. Chapter 5 then expands the contextual scope to consider uses of a subject that span multiple invocations on the holder of the subject. In this case, rather than cache query results in local or instance variables, the implementation of the holder is actually *specialized* to use pre-computed polymorphism results. Again, the effects of the technique are evaluated, and some analyses of the results are offered to shape future refinements. In closing, chapter 6 provides an assessment of the main goals of the thesis, and revisits the subject-oriented and design pattern perspectives in order to give a sense of the broader implications of the work.

Chapter 2
Conceptual Foundations

The concept of polymorphism has deep ties to object-oriented programming. Early incarnations took root in foundational object-oriented languages such as SIMULA 67 [Nygaard and Dahl, 1978] and Smalltalk [Ingalls, 1978], well before the rise of popular languages such as C++ and Java, and well before the formalization of object-oriented design principles [Booch, 1982]. However, despite its prominent role in the co-evolution of programming language features and software engineering practices, a clear and general methodology articulating how to use polymorphism in object-oriented programming has yet to emerge. Instead, software design experts continue to advance strategies for using polymorphism based on anecdotal examples. Naturally, the diversity of these examples has led to confusion about how to consistently and effectively integrate polymorphism into new program designs.

One consequence of this lack of clarity is that research directed at improving the efficiency of polymorphism has mostly stagnated since the development of the virtual function table. Some recent efforts have sought to eliminate provably or probably superfluous polymorphism, but these techniques offer no improvement in situations where polymorphic behavior is actually intended. Yet the simple example in figure 1.1 illustrates that a broader focus on the context in which polymorphism appears, rather than a narrow focus on its implementation in isolation, reveals observable—and hence eliminatable—redundancies. Moreover, while no universal method for using polymorphism has emerged, programmers have adopted *de facto* strategies for using it in the form of reoccurring design patterns. This thesis posits that a study of how polymorphism is used in such patterns reveals several new approaches to eliminating some of the overhead incurred by virtual dispatching and other mechanisms related to object-oriented polymorphism.

2.1 Varieties of Polymorphism

Polymorphism (*poly*: many, *morphe*: form)—when apparently similar items exhibit distinct behavioral personalities—is a favored technique among software designers for adding both structure and abstraction to program designs. It delivers structure by unifying related items or actions under a single banner, and offers abstraction in the form of delegation—the "system" rather than the programmer is tasked with selecting the most appropriate form or action. Cardelli and Wegner [1985] identify two major classes of programming language polymorphism: *ad hoc polymorphism* and *universal polymorphism*.

Ad hoc polymorphism, which includes symbol *overloading* and type *coercion*, uses the same expression to represent different behaviors that may have no formal relationship. A common example is the use of the `+` (plus) symbol to represent both arithmetic addition and string concatenation. Programmers use ad hoc polymorphism to trade apparent ambiguity for concision: the compiler is relied upon to deduce the appropriate action based on the static (compile-time) types of items involved in the expression. This form of polymorphism is limited in the sense that the compiler must be aware of all the possible alternatives to make unambiguous decisions about the semantics of an expression.

Universal polymorphism, on the other hand, facilitates open-ended reuse because it permits substitution of any structure that conforms to certain type rules. There are two main varieties of value-oriented universal polymorphism: *parametric polymorphism* and *inclusion polymorphism*.[1] Parametric polymorphism essentially consists of reuse of the same packaged algorithm or data structure instantiated with different component types. Templates in C++ and generics in Java are both examples of parametric polymorphism. Inclusion polymorphism, conversely, unifies different algorithms or data structures under the same interface. All items are given the same general directives, but it is expected that individuals may satisfy the requests through distinctly different behaviors. The items and their behaviors are related through membership (inclusion) in a particular family of types. Virtual dispatch in object-oriented languages is the most common application of this kind of polymorphism. Functional programming languages such as Haskell and ML also support inclusion polymorphism in the form of pattern matching over *algebraic data types*. Type tests and downcasts are mechanisms related to inclusion polymorphism that allow programmers to make explicit selections of behavior based on a value's "type" attribute.

What distinguishes inclusion polymorphism from the other varieties of polymorphism is its dependence on dynamic (late) binding. The decision as to exactly which behavior to execute is intentionally delayed until the last possible moment to permit free substitution of different variants at run time. This valuable flexibility comes at a price though, in that the program, rather than the compiler or linker, must bear the cost of identifying the target behavior. This thesis investigates how such costs can be eliminated, or at least mitigated, through adaptive program transformations. Thus the focus will be on inclusion polymorphism, specifically as it is used and implemented in object-oriented programming languages. For the sake of concision, we will simply use the term "polymorphism" to mean inclusion polymorphism throughout the rest of the text.

2.2 The Role of Polymorphism in Object-Oriented Design

As both a language feature and a design tool, polymorphism has followed an interesting historical arc. Early incarnations were often developed in conjunction with efforts to include more abstraction mechanisms in programming languages. However, with the rise in popularity of class-based object-oriented languages—C++ in particular—the role of polymorphism seemed to shift. A view that began to dominate during the 1980s and early 1990s was that polymorphism was primarily a tool for

[1] Jay [2009] discusses several other, more esoteric forms of universal polymorphism over higher-order types.

enabling reuse of code via inheritance. This understanding of polymorphism fueled a vigorous debate regarding the utility of multiple inheritance in object-oriented languages. What followed in the mid to late 1990s was an interesting reversal. Software engineers began to realize that polymorphism was very useful in certain situations, enhancing the decoupling of program components while permitting flexible substitution. This was, in essence, a rediscovery of the original roots of polymorphism, laid out 20 years earlier, as a tool for building flexible abstractions. Despite this renaissance, however, what has yet to occur is a corresponding return to a similar emphasis in the implementation and optimization of polymorphism. To date, most of the research aimed at improving the efficiency of polymorphism (virtual dispatch, in particular) has grown from the legacy of C++, its descendants, and the view that polymorphism is primarily a tool for static code reuse rather than dynamic substitution.

2.2.1 The Origins of Inclusion Polymorphism

In the 1970s the diversity of programming languages increased dramatically. Along with this growth came a variety of arguments and proposals for languages and mechanisms to support both abstraction and substitution in programming. Parnas [1972], Wulf and Shaw [1973], and Liskov and Zilles [1974] all provided arguments for establishing opaque boundaries between program components. DeRemer and Kron [1975] argued that a program's architectural design—how the various components connect to, and interact with each other—should be programmed independently from the individual components. Meanwhile, Jones and Muchnick [1976] advocated for a language with very late binding to promote flexible and rapid development.

These ideas would serve as a foundation for two important paradigms related to polymorphism which had emerged by the end of the 1970s. First was the concept of programming relative to an opaque interface, thus permitting different implementations to be substituted without extensive code modifications. This approach would become a cornerstone of development in languages such as Modula-2 and Ada. The technique was limited, however, in that all of the actual implementation bindings still needed to be resolved at compile time, or at least link time. Selecting an implementation during execution was not yet possible. The second significant idea was the notion of objects as reactive entities, resolving requests on-demand in accordance with each receiver's particular capabilities. Smalltalk was the main progenitor of this approach, encouraging developers to view their programs in terms of objects sending messages to each other. This dynamic message passing would ultimately evolve into the mechanism known as virtual dispatch. More than a decade after their introduction, these two ideas (opaque interfaces and late-binding dispatch) would come together to form the modern conception of dynamic inclusion polymorphism.

2.2.2 The Rise of the Inheritance Paradigm

During the 1980s, C++ rose to dominate the world of object-oriented programming. Its sophisticated mechanisms for defining and relating classes encouraged developers to focus on the static hierarchical relationships among objects in their programs, in particular emphasizing the discovery

of behaviors that were similar and could potentially be housed in the same implementation. From this view sprung some of the classic polymorphism clichés: geometric shapes, animals, vehicles, employee roles within a company, etc.[2] These examples typically centered on the inheritance (reuse) of behavior, while overriding behavior—i.e., exhibiting actual polymorphism—was often treated as a feature to be used sparingly. Both Liskov [1987] and Madsen et al. [1990] observed that this emphasis on inheritance obscured an important distinction between subclassing and subtyping: extending the behavior of an existing (i.e., concrete) implementation is quite different, from a design point of view, than developing an implementation for an opaque (i.e., abstract) interface. This confusion ultimately erupted into a, sometimes heated, debate over the role and utility of multiple inheritance [Cargill, 1993; Waldo, 1993] which was known to have a problematic implementation in C++.

While the multiple inheritance debate was, more or less, put to rest by Java's adoption of distinct class vs. interface extension mechanisms [Gosling et al., 1996], the association of polymorphism with reuse (rather than substitution) continued to persist in early Java texts [e.g., Cooper, 1997].

2.2.3 Polymorphism in Modern Software Design

By the 1990s, object-oriented programming had become mainstream and more mature views of how to apply it finally began to take shape. The techniques of object-oriented analysis (OOA) and object-oriented design (OOD) emphasized decomposition, decoupling, and reuse via composition rather than inheritance [Wirfs-Brock et al., 1990; Booch et al., 2007; Sullo, 1994]. These views gained support as programmers made increasing use of frameworks (libraries) of code where little or no knowledge of the implementation details were available:

> "*Rather than using hierarchy as the basis for reuse, object-oriented frameworks employ the concept of 'separation of concerns' through modularity.*" [Budgen, 2003, p. 368]

What distinguished these new approaches from those advocated during the 1980s and 1970s was a focus on the use of polymorphism as a means to enable dynamic substitution of various components and behaviors.

2.2.4 The Role of Polymorphism in Design Patterns

In 1992, Coad published a seminal paper [Coad, 1992] in which he reflected on the concept of a *design pattern* and how such patterns could be applied to object-oriented program design. Briefly, design patterns—an idea popularized by the work of prominent architect Christopher Alexander—represent an attempt to codify reoccurring, effective arrangements of common artifacts. In the world of architecture, patterns often represent tangible arrangements of physical structures such as doors, windows, and walls. However, the elements of a design pattern may also take on more abstract roles and relationships—for example, a focal structure and its corresponding echoes. Coad (and others [e.g., Gamma et al., 1995; Pree, 1995]) recognized that this latter conception of a design

[2] Similar examples still pervade modern introductions to object-oriented programming. See, for example: Cohoon and Davidson [2006]; Johnson [2007]; Lewis and Loftus [2008]; Reges and Stepp [2008]; Savitch and Carrano [2009]; Lewis and Chase [2009].

pattern could also be used to describe and communicate about reoccurring object-oriented program designs.

What distinguished the design patterns approach from traditional object-oriented analysis was its emphasis on arranging objects in terms of archetypal roles, giving names to objects and their collaborations that are unencumbered by application-specific associations. In other words, patterns represented an approach to program organization which focused more on how objects are used—how they are shared, passed, used as delegates, etc.—as opposed to what they are used for—i.e., the application data and/or functionality that they embody. A classic example is the Model-View-Controller (MVC) approach to separating user interface logic from data representations and manipulations [Krasner and Pope, 1988]. This arrangement says nothing about the actual content of the data model, nor does it imply any particular viewing or controlling functionality, it simply posits that the overall adaptability of the program is enhanced if these aspects are partitioned into separate, independent components with explicit rather than implicit communication pathways.

The MVC design is now pervasive in user interface programming frameworks, and it is this wide acceptance that makes it a true design pattern. To qualify as a proper design pattern, rather than just a clever or elegant idea, a given arrangement must have a demonstrated history of value in application. In architecture, patterns often arise from geometric relationships that are known to work well, producing for example, structural rigidity, ease of movement, or a pleasing aesthetic effect. Software design patterns, on the other hand, typically focus on organizational relationships among program components. A key measure of value in such patterns is how effectively they support the inevitable evolution of software over time. A pattern has value if, as new demands and capabilities arise, its organizational structure permits additions and substitutions while at the same time keeping a reasonable bound on the impact that changes have on existing components.

In their, now classic, text the "Gang of Four" [Gamma et al., 1995] presented a catalogue of object-oriented design patterns, each with an established history of effectively facilitating change and adaptation. The book begins by outlining two principles that underlie the patterns approach to object-oriented design:

1. *Program to an interface, not an implementation.*
2. *Favor object composition over class inheritance.*

These principles represent an important departure from the subclassing heritage of C++ and its influence on object-oriented design because they implicitly assert that the boundaries between architectural roles should be polymorphic. Programming to an opaque interface, rather than an exposed implementation, anticipates the substitution of different component behaviors. And favoring architectures constructed through parameterized, dynamic composition, instead of static inheritance relationships, presumes that substitution of distinct variants will occur.

Now, it is worth noting that, while these principles are often touted as universally applicable to object-oriented design, the intent of the Gang of Four was to advocate for their use in the context of pattern-oriented design. Thus traditional subclassing—an example of polymorphic typing where

actual polymorphic behavior is rare—may be considered useful in the functional design of programs, but should be avoided as an architectural design tool. This last insight reveals our interest in design patterns: if the goal is to study situations where polymorphic behavior is expected, and discover contexts where the resolution of such behavior may contain redundancies, then patterns are likely to suggest examples which are both relevant and reoccurring, and hence worth investigating.

2.3 Inclusion Polymorphism Implementations

Object-oriented languages typically provide three operations for programming with inclusion polymorphism: type tests, type casts, and virtual dispatch. Each of these operations involve performing a query against a particular *subject* (or *receiver*) to identify either what it is—i.e., the type(s) it implements—or how it behaves. The queries, and their associated costs, are mostly concealed in high-level programming expressions to better support abstraction. For example, programs written in C++ or C# use the same syntax to represent both virtual and non-virtual instance method calls. In such programs, each virtual call entails a sequence of load operations to identify the actual target method, whereas non-virtual calls use a direct transfer to a predetermined target. Programs that use Java's generic typing system also contain numerous, hidden type casting operations which are required to enforce the source language semantics over the non-generic bytecode implementation. While the precise cost of polymorphism queries varies across different operations and implementations, all involve at least two (and often more) load operations. This thesis focuses, in particular, on the polymorphism operations provided by the Java programming language [Gosling et al., 2005] and its associated bytecode execution model [Lindholm and Yellin, 1999]. Driesen [2001] provides a broader historical perspective on the evolution of polymorphic dispatching, discusses the efficiency of different implementation alternatives, and surveys the technical challenges involved in implementing extended forms of polymorphism such as multiple inheritance and multiple subject dispatch.

Modern object-oriented languages use one of two approaches to implementing polymorphism queries. The first approach, seen primarily in the implementation of prototyping languages such as Self [Chambers et al., 1989], associates with each object only explicitly declared attributes and behaviors. Inherited characteristics are represented by implicit links to parent objects. Resolving a polymorphism query in such systems thus involves traversing a chain of inheritance links, requiring (in the worst case) a number of loads proportional to the inheritance depth.

The other common approach to implementing polymorphism is employed by most class-based object-oriented languages such as C++ and its progeny. Because of their static type hierarchy, such systems are able to reduce the number of load operations by associating a shared type information structure with each object. The type structure for each new derived class is built by first copying all information from parent types. Then, new features added to the subclass extend the type structure, whereas overriding (i.e., polymorphic) features replace entries copied from the parent. For its original implementation, C++ used a simple array of code pointers, called a *virtual function table*, to represent polymorphism data. Each object held a reference to a table shared by other instances of the same type. Executing a virtual method dispatch consisted of two load operations: one to

access the receiver's virtual table, and another to obtain the code address held at a precomputed offset within the table. However, while both simple and efficient, this implementation was not capable of supporting the broader range of polymorphism operations such as dynamic type casts or dispatching in the presence of multiple parent types. Modern implementations of C++ extend the virtual table model to support these operations, but require explicit programmer participation through the use of the `virtual` inheritance modifier and compiler options to include additional run-time type information (RTTI) support.

The Java execution model, on the other hand, explicitly defines four polymorphism operations which are always available: classic, single-inheritance virtual dispatching (`invokevirtual`), interface dispatching (`invokeinterface`), type testing (`instanceof`), and casting (`checkcast`). All of these operations entail a run-time query of an object's type information. The experiments in this thesis focus on the specific implementation of these features provided by the Jikes RVM (Research Virtual Machine).

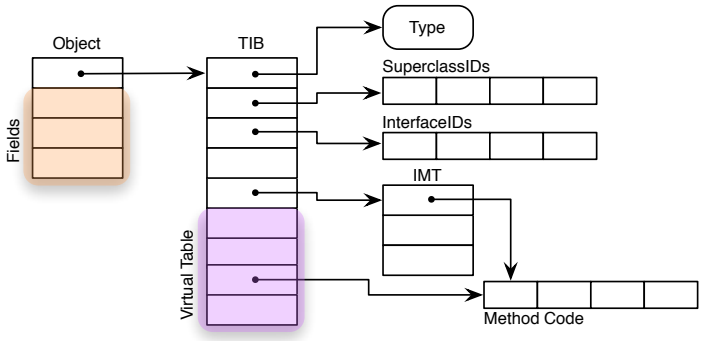


FIGURE 2.1: **Structure of Polymorphism Data in Jikes RVM**

In Jikes RVM, each object holds a reference to a shared *type information block* (TIB) [Bacon et al., 2002]. A TIB, depicted in figure 2.1, is similar to a C++ virtual table, but also includes other entries to support efficient type tests, casts, and interface dispatching. With this structure, a normal virtual dispatch (see figure 2.2) involves two load operations to access the target code. Through the use of a hashing scheme, most interface dispatches involve one additional load to access code via an *interface method table* (IMT) [Alpern, Cocchi, Fink, and Grove, 2001]. Some interface dispatches also entail an additional type test operation.[3] Type tests and casts are implemented by matching against bit patterns held in two arrays, one representing parent classes, and one representing parent interfaces [Alpern, Cocchi, and Grove, 2001]. Determining inclusion in a type family thus requires four loads: three to obtain the appropriate bit pattern, plus one to check that the computed bit pattern's index is within the array bounds.

[3] This requirement and its implications are discussed in §4.1.

2.3.1 The Run-Time Cost of Polymorphism

Although polymorphism is widely recognized as integral to object-oriented programming, confusion about its actual run-time cost has, on occasion, undermined its utility as a design tool. For example, consider how early, inefficient implementations of Java's interface mechanism fostered an aversion to the use of subtyping polymorphism:

> "*Some older, well-meaning Java performance literature suggests that casting to a class is faster than casting to an interface. Similar claims have been made that it is faster to invoke methods on a variable declared in terms of a class rather than an interface. However, benchmarking with modern JVMs and native compilers indicates no significant difference.*" [Larman and Guthrie, 2000, p. 96]

While such confusion is unfortunate, it is also understandable given that the mechanisms underlying polymorphism are obscured in high-level source code, thus concealing the extent to which they may impact performance.

Driesen and Hölzle [1996] examined the cost of virtual dispatch in particular by studying the behavior of C++ programs run on a number of actual and simulated architectures. They found that instructions related to virtual dispatching represented a significant portion of both the total instruction and cycle counts of their test programs. Moreover, when the programs were transformed to use virtual dispatching for all instance method invocations (as is the default in Java), the measured overhead more than doubled. Driesen and Hölzle account for the increase in cycle counts by noting that table-based virtual dispatch implementations introduce both data and control dependencies which interfere with the operation of pipelined, superscalar processors. Specifically, the back-to-back, dependent loads in a virtual table query increase load latency, inducing frequent pipeline stalls. These delays are exacerbated by the indirect branch at the end of a dispatch sequence which presents an opaque control hazard, further impeding pipelined execution. Driesen and Hölzle suggest that advanced hardware techniques such as out-of-order execution and speculative execution can, to some extent, be employed to reduce the delays incurred by these effects. However, these techniques may still be insufficient in the face of frequent, and truly polymorphic virtual dispatch sequences given the high cost of pipeline failures in modern architectures and the challenge of effectively predicting indirect branches.

In relating the results of Driesen and Hölzle to implications for Java, consider that Dufour et al. [2003] found that, over a wide range of Java benchmarks, virtual dispatches constitute approximately 5-10% of bytecodes executed, and that as much as 17% of such invokes exhibit some polymorphism. Also, Li et al. [2000] found that programs from the SPECjvm98 suite exhibited poor instruction-level parallelism (ILP) compared to other non-Java benchmarks—an effect due, in part, to the scheduling problems outlined by Driesen and Hölzle. Additionally, Zendra and Driesen [2002] confirm that the choice of virtual dispatch implementation for Java can have significant impact on efficiency.

A factor not addressed by Driesen and Hölzle is the reduction in efficiency of the memory hierarchy due to the displacement of regular program data by type information structures. Shuf et al. [2001] provide some insight through a study of programs run on an early version Jikes RVM. They observe

that Java benchmarks exhibit generally poor data cache performance relative to non-Java workloads,[4] however few L2 cache misses are due to loads of TIB entries. Surprisingly though, a significant fraction of translation look-aside buffer (TLB) misses are due to TIB queries. Shuf et al. do not consider the effect of TIB queries on L1 cache performance. They also do not differentiate between virtual dispatches, interface dispatches and other TIB queries such as type tests.

Another concern not addressed by current studies is the impact of polymorphism implementations on register use, and concomitant spills to the process stack to maintain live variables. By necessity, the semantics of polymorphism operations create extra demand for local variables (i.e., registers), often in the midst of other active computations. With Jikes RVM's TIB structure, an `invokevirtual` operation can usually be implemented by co-opting only one register. However, type tests—including those implied by `invokeinterface` operations—require several active registers to implement. On Intel x86 architectures (the focus of experiments in this thesis), registers are a very limited resource [Intel, 2009b]. Because of this, even the most trivial Java methods involving polymorphism operations are likely to induce costly spills to the process stack.

Building on many years of experience, the type information structures in Jikes RVM's have been carefully engineered to minimize the overhead of polymorphism queries and rival the best-performing alternatives seen in other class-based language implementations. However, each polymorphism operation still requires two or more dependent load operations and, in the case of dispatches, a costly indirect branch to the computed call target. For programs that involve millions of polymorphism operations,[5] the cumulative overhead of even the most streamlined implementation may still be quite significant. Fortunately, as the following sections reveal, much of this overhead can often be eliminated by considering the context in which polymorphism operations occur.

2.4 Current Approaches to Eliminating Redundant Polymorphism

Historically, strategies for improving the efficiency of polymorphism (virtual dispatch, in particular) fall into three categories: those that attempt to eliminate the receiver query, those that focus on the call overhead, and those that seek to eliminate both by transforming the code entirely. The first and third strategies generally rely on static analyses to identify situations where polymorphic behavior is degenerate—i.e., provably monomorphic. They transform programs by replacing all or part of a polymorphism computation with an exact, precomputed result. The second approach permits some polymorphic behavior, but uses heuristics and/or profile information to guess the most likely call targets. The execution path to likely targets is streamlined by trading the overhead of an indirect branching call for that of a guard—a simple conditional branch which compares the expected query result against the actual result. The body of the call target may then be inlined under the guard, exposing interprocedural optimization opportunities.

All three of the prevailing optimization strategies are based on the premise that, in practice, many polymorphism operations produce completely, or nearly monomorphic results. Dufour et al.

[4] For a simulated "PowerPC 604e-like" cache architecture.
[5] See tables 3.3 and 3.4.

[2003] confirm that this property holds for many, but not all, Java benchmark programs. Similar measurements presented in tables 3.3 and 3.4 also corroborate these observations.

The challenge in applying current techniques to Java lies in the fact that, unlike C++ and C#, the semantics of Java instance methods always imply a polymorphic invocation.[6] On the surface, this feature would seem to suggest the presence of more optimization opportunities. However, Java's dynamic class loading behavior and reflection capabilities complicate the situation since they permit new types to be introduced at any point during the execution of a program. As a consequence, all transformations of polymorphism in Java must be conservative and allow for the possibility that operations which appear monomorphic may, at a later point, resolve to a newly introduced variant.

2.4.1 Static Devirtualization

The most widely studied approach to eliminating redundant polymorphism is known as *devirtualization*. As its name suggests, the technique removes the need for a virtual table query at a particular call site by identifying a fixed target through some form of program analysis. Figure 2.2 depicts a simplified example transformation. The call remains after the transformation, but subsequent optimizations are permitted to eliminate the call overhead by inlining code from the unique target.

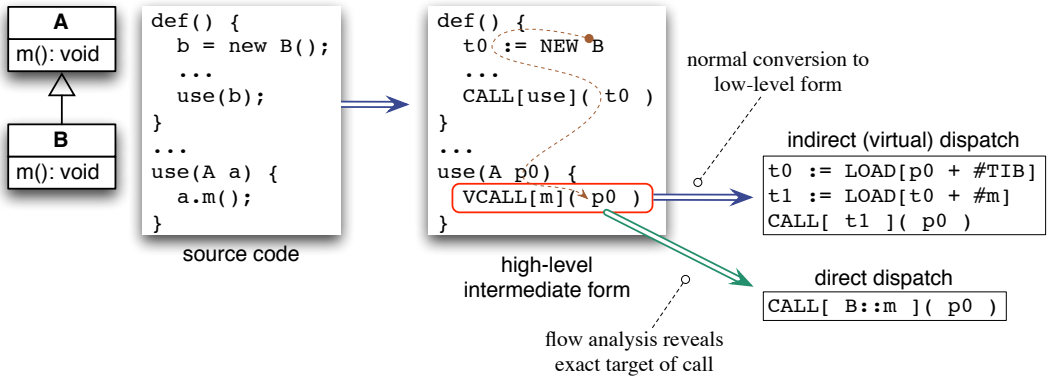


FIGURE 2.2: **Static Devirtualization Transform**

Although identifying all situations where devirtualization can be applied is *NP*-hard for C++ programs [Pande and Ryder, 1994] and effectively impossible for Java programs (due to dynamic class loading), a number of fruitful approximations have been developed. To locate opportunities in C++ programs, Calder and Grunwald [1994] search the list of compiled method signatures and substitute a direct binding in cases where only a single, matching implementation is found. Dean et al. [1995] use *class hierarchy analysis* (CHA), which combines the declared type of a dispatch receiver along with class hierarchy information to refine the set of possible targets, devirtualizing in

[6] Constructors and private methods are an exception, and may be directly accessed via an `invokespecial`.

situations where only a single target is possible. Bacon and Sweeney [1996] further refine the results of CHA, through a modification called *rapid type analysis* (RTA), by only considering classes that are actually instantiated. An important strength of both CHA and RTA is the fact that they do not rely on control flow analyses or interprocedural context modeling. While such detailed analyses (for example, k-CFA [Shivers, 1991]) do yield more precise results they are known to be impractical for large, object-oriented programs [Grove et al., 1997].

To provide stronger results for Java, Sundaresan et al. [2000] use a flow-insensitive, whole-program analysis, called *variable type analysis* (VTA), to estimate the set of types each (static) variable may contain throughout its lifetime. Tip and Palsberg [2000] explore several similar approaches—primarily one dubbed XTA—which collect the set of possible types in coarser granularities such as those used throughout the implementation of a method or class. These approaches represent intermediate points between precise, but costly analyses like k-CFA and the simplicity of RTA: VTA reduces the cost of analysis by sacrificing flow-sensitivity, XTA sacrifices both flow-sensitivity and variable-sensitivity, and RTA is completely oblivious to control flow and uses only one, global set of possible types as opposed to variable-specific, method-local, or class-local sets. The success of a whole program type analysis does, however, rest on an odd circularity: in order to establish a call graph over which to perform the analysis, it is first necessary to have a reasonably precise estimate of the receiver types at each call site. To cope with this paradox, the original version of VTA used estimates provided by CHA or RTA. Subsequent work by Lhoták and Hendren [2003, 2006] examined variations on pointer analyses for Java which provided more precise estimates, but at a higher cost.

Even though VTA and XTA have been shown to produce fairly precise call graphs (exposing devirtualization opportunities when only a single path emanates from a call site), it is important to note that they are ahead-of-time analyses which presume that all relevant class definitions are available and contribute to the results. As noted previously, this requirement is not entirely reasonable for Java programs since it may be impossible to infer the complete type hierarchy of a program until it has finished executing. In practice, this issue can be dealt with by allowing developers to provide an explicit list of classes which may be loaded dynamically and then making conservative assumptions regarding areas of the type hierarchy which remain open or incomplete. To support interprocedural analyses in a just-in-time environment (i.e., Jikes RVM), Qian and Hendren [2004] developed a version of XTA which accumulates results incrementally as fields and methods are resolved during execution.

Given this plethora of type analyses,[7] with applications well beyond the optimization of polymorphism, what are the practical implications for the devirtualization of Java programs? In a follow-up study, Qian and Hendren [2005] provide a succinct answer:

> "*Surprisingly, the simple dynamic CHA is nearly as good as an ideal type analysis for inlining virtual method calls. ... On the other hand, only reachability-based interprocedural analysis (VTA) is able to capture the majority of monomorphic interface calls.*"

[7] Grove et al. [1997] enumerate many more techniques with varying precision in their study of call graph construction for Cecil and Java programs.

Interestingly, these results reflect the subclassing vs. subtyping design dichotomy discussed earlier in §2.2: the opaque nature of interfaces creates polymorphism which is difficult to pierce, whereas class-oriented polymorphism—presumably a byproduct of hierarchy-oriented reuse—is reasonably transparent and easy to resolve.

2.4.2 Speculative Binding and Inlining

In general, the most costly element in the implementation of virtual dispatching is the overhead of the actual call operation—specifically, the indirect transfer of control and the stack manipulations in the prologue and epilogue of the call. In an effort to eliminate all or part of this overhead, several alternatives to C++-style virtual dispatching have been developed.

Inline Caching

To convert indirect calls into direct calls, systems such as Smalltalk-80 used a mechanism known as an *inline cache* which performs a simple test against a receiver's type, branching directly to the target implementation on a match, or performing a slower lookup and indirect branch on a mismatch [Deutsch and Schiffman, 1984]. When a mismatch occurs, the dispatch site is transformed so that subsequent queries match against the most recently identified target, hence relying on the observation that most call sites exhibit little polymorphism to achieve amortized improvement. Hölzle et al. [1991] present an extension called a *polymorphic inline cache* which uses a sequence of tests for truly polymorphic sites where regularly updating a single, last result becomes counter-productive. In short, the inline cache approach trades an indirect call (i.e., a branch) for a direct call preceded by a simple test. At the hardware level, this approach improves scheduling for superscalar architectures, utilizing a processor's branch history table (BHT) more than its branch target buffer (BTB)—the capacity of the former often being significantly greater than the latter. Leveraging results from a type inference analysis, the implementation of inline caching employed by the SmallEiffel compiler also eliminates a load operation at each dispatch by disposing of an object's virtual table in favor of a simple type identifier [Zendra et al., 1997]. Driesen [2001] provides a more complete overview of inline caching, including a discussion of the architectural implications for performance.

Guarded Inlining

To eliminate the overhead of call operations, many optimizing compilers perform a transformation known as *inlining* (not to be confused with inline caching) which replaces call instructions with a complete copy of the target routine's code. Inlining not only removes the cost of the branch, prologue, epilogue, and return, but also exposes further opportunities for optimization by unifying the caller and callee contexts. For example, constant values passed from the caller can be propagated into the duplicate code, allowing simplifications which would not be possible in the original callee's code.

The difficulty in applying inlining to languages like Java lies in the fact that virtual calls may have more than one target, and thus more than one candidate implementation which could be copied into the caller. Furthermore, as noted in §2.4.1, identifying a unique target for virtual dispatches

can be challenging, and is often impractical for just-in-time compilation systems.

Detlefs and Agesen [1999] provide a solution to this problem by inlining a likely target and preceding the inlined section with a guard: a quick test against the receiver's type (a *class test*) or the expected target address (a *method test*) which falls back to an indirect call when the test fails. The method test variation is illustrated in figure 2.3. Although the virtual table query remains, the overhead of the call is eliminated in the case where the actual target matches the expected target. Choosing which target to inline can either be achieved through heuristics (Detlefs and Agesen use a combination of run-time hierarchy information together with the receiver's declared type) or by discovering the most frequent target as indicated by on-line or off-line profiles.

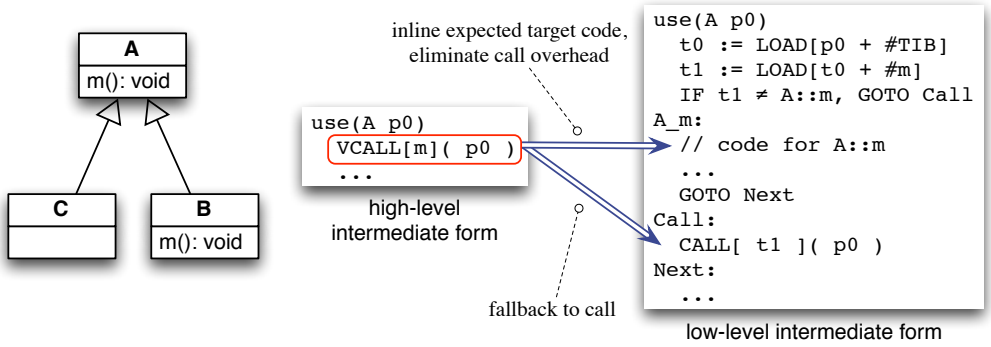


FIGURE 2.3: **Guarded Inlining Transform**

For small inlined targets, Joao et al. [2008] show how to extend the approach by eliminating the cost of the branching guard through the use of predicated instructions tied to the result of the guarding test. Detlefs and Agesen [1999] also use a property they call *preexistence* to speculatively eliminate both the query and guard in certain cases by determining whether the receiver was instantiated at time when only one target implementation was possible. The approach does require recompilation when another implementation target becomes available (via dynamic loading), but it removes the need for an immediate update using *on-stack replacement* (OSR), as is done in SELF [Hölzle and Ungar, 1994]. Ishizaki et al. [2000] take the idea further and always omit the query and guard when only a single target is known at the time of compilation. If another target implementation is loaded later which may violate the direct inlining assumption, rather than recompile the entire invalidated method, the compiler simply overwrites the first inlined instruction with an unconditional branch to a classic virtual dispatch sequence or an OSR recompilation trigger. This lazy approach defers recompilation, only updating the code in cases when a particular execution actually visits the unexpected case for the site in question.

To optimize virtual dispatching, Jikes RVM uses guarded inlining with a combination of the *patching* scheme of Ishizaki et al., preexistence, and class tests or method tests for sites where more than one target is known.

2.4.3 Specialization

A similar technique to inlining is *specialization*. Rather than create an inline duplicate of the target code, specialization creates a separate routine with certain input parameters bound to constant values. These bindings permit the partial evaluation of some of the specialized code at compile time, thus creating a faster implementation alternative. Code analyses and/or run-time statistics gathered through profiling can be used to suggest when to create specialized routines, and where they ought to be invoked instead of the default routine. The goal of specialization is to avoid the significant increase in code size which is a common byproduct of inlining: although specialization does duplicate code, the assumption is that some of these duplications are reusable. In practice however, unrestrained specialization can still generate a significant amount of mostly redundant code, and may be overly wasteful of compilation effort in a just-in-time environment.

Chambers et al. [1989] developed a restricted version of specialization for Self called *customization* which only specializes on the receiver of an object-oriented dispatch. Customization is implemented by creating a separate version of all inherited methods, specialized on the current receiver type. The primary optimization applied within the specialized methods is direct binding of any sub-calls invoked on the `self` reference (`this` in Java) since these calls must also have a unique specialized implementation.[8] Observing that customization is often unfocused for programs with deep hierarchies and many methods, Dean et al. [1995] revised the approach by returning to general specialization over all method parameters, but used an analysis and a cost/benefit model to selectively apply the transformation. Again, their aim was to identify opportunities for direct binding. Suganuma et al. [2001] developed a similar approach, but also considered situations where specialization can be used to eliminate type tests and array-bounds checks. The object-inlining technique of Dolby and Chien [2000] is also similar and represents a form of specialization achieved through a combination of data restructuring and devirtualization.

While analysis-based specialization and its extensions have proven effective, they still overlook some opportunities due to the limitations of type analysis and dynamic loading. Some have sought to expose further opportunities through the use of explicit, declarative specifications for specialization [Volanschi et al., 1997; Tourwé and De Meute, 1999; Schultz et al., 2000; Liu et al., 2005; Rogers et al., 2008]. These approaches generally take the form of constraint languages or annotations that supplement the original program code. Schultz et al. [2003] extended one approach to use profiling information to automatically generate specialization directives as aspect-oriented specifications for Java. Other automated approaches to specialization include those that gather *traces*—instructions or calls observed executing in sequence—and produce code optimized for specific execution paths. Versions that target polymorphism operations include the JVM implementations of Suganuma et al. [2006], and the JavaScript optimizer of Gal et al. [2009].

An alternate view, which is particularly apt for object-oriented languages, is to consider specializations that are applicable to specific object instances, rather than procedural contexts. Tip and Sweeney [1997] created structural specializations, eliminating unused fields for instances which

[8] This optimization is only possible if all inherited implementations are specialized.

do not use them. Su and Lipasti [2006] inspected the values of instance fields to create specialized method implementations for individual objects, effectively replacing the virtual table of certain objects. Chapter 5 presents a similar technique that focuses on identifying the type, rather than the value of fields, to create instance-specific specializations since knowing the exact type of a field is sufficient to reduce polymorphism operations applied to it.

2.5 Opportunities

In general, the previous approaches to optimizing polymorphism reviewed in §2.4 all suffer from the same bias—they seek a unique, "best" optimization at each query site. This focus discounts the possibility that a more contextualized approach may yield a better net result. Specifically, such approaches overlook a simple fact: if polymorphism bindings are fixed over the lifetime of a subject (a property of all statically typed languages) then subsequent repetitions of the same query applied to the same subject must, necessarily, produce the same result. In other words, once a query has been performed within a given context, it does not need to be recomputed if applied again to the same subject. Thus, if one is willing to accept the cost of the initial query, rather than apply heavy-handed analyses to deduce the result, then the first result computed can be stored and used as a replacement for any subsequent, similar queries.

2.5.1 Four Contexts of Redundancy

If the goal is to reuse the results of queries already applied to a given subject, then two important questions present themselves. First, what constitutes a redundant computation? More precisely, what is the scope or context over which an operation should be deemed to be repeating an earlier, equivalent computation? The semantics of object-oriented languages with dynamic loading suggest five potential contexts:

Intraprocedural –
: Reuse within the context of a single function, procedure or method.

Interprocedural –
: Thread-local reuse that crosses invocation boundaries from caller to callee, or sub-callee.

Instance-Lifetime –
: Reuse that occurs relative to a specific object instance, over its entire lifetime.

Module-Lifetime –
: Reuse relative to a specific metaobject or dynamically loaded module—a class in Java.

Global –
: Any recomputation of a semantically equivalent result throughout the execution of a program.

Strictly speaking, the global context is not important for polymorphism queries since no subject, and hence no query result, is reusable beyond the (module-lifetime) context of its type definition.

With the first four relevant contexts in mind, the next obvious question for language implementors is then: How should redundant query results be stored such that they can be retrieved with little effort for a given context? In the case of the intraprocedural context, the obvious choice is to store results in a new, local variable, as is done in all traditional approaches to intraprocedural redundancy elimination. On the other hand, assuming multithreaded execution, the only directly accessible memory shared across interprocedural contexts is the process stack. However, storing query results in the stack is problematic for several reasons. First, one load operation from the stack is not much of an improvement over the typical two loads needed to access a virtual table entry. Second, locating shared results within the stack would require some form of communication between caller and callee—in other words, an additional parameter for each shared result. And third, the boundary between caller and callee may itself be polymorphic, making it unclear which query results ought to be passed. For these reasons, there seems to be little opportunity in attempting to eliminate interprocedural polymorphism redundancies. Accessing results that span instance or module lifetimes would also appear to require a load from some extension of the instance or module's memory space. However, an option in these two cases is to store results implicitly in code specialized for the values held in a particular object or module instantiation. Thus rather than load instance-lifetime and module-lifetime query results as needed, the results are computed by a just-in-time compiler when a specialization is deemed profitable and then embedded in the specialized code as directly bound calls or pre-evaluated type tests.

This understanding of the opportunities for efficiently accessing previously computed query results, in combination with insights about the evolving role of polymorphism in object-oriented design, suggests an entirely new approach to optimizing polymorphism operations.

2.5.2 Thesis

Over the past two decades, the role of polymorphism in software design has matured. The influence of hierarchy-oriented reuse still lingers, but leaders in the field now promote a view of polymorphism as a tool for enabling the dynamic (i.e., run-time, or computed) construction of software from interchangeable components. However, this shift in the understanding polymorphism has not been met with similar progress in the field of program optimization. Traditional approaches to improving the performance of polymorphism all share the common aim of identifying static code locations where all or part of a polymorphism operation can be bypassed due to provable or probable monomorphic behavior. As a consequence, these approaches offer no improvement for sites where polymorphic behavior is actually intended as part of a program's design.

The insight needed to adapt code optimization techniques to the modern conception of polymorphism is a subject-oriented, rather than site-oriented, approach to redundancy elimination. Specifically, an understanding that, while the site of a polymorphism operation may resolve to different results over the span of a program execution, repeated polymorphism operations applied to the same subject must produce redundant results even if the exact type of the subject cannot be revealed through analysis. Leveraging this insight requires the ability to discern when an operation

is, in fact, applied to the same subject—a challenge which is also, in general, a difficult analysis problem.[9] However, reuse of the same subject can be detected with little effort in two common situations: when the subject is a local variable in an intraprocedural context, and when the subject is a fixed (i.e., `final`) field of an object or module. Together, these observations lead to the three central claims of this thesis:

1. It is possible to improve the performance of intentional inclusion polymorphism by transforming programs to reuse rather than recompute polymorphism query results applied to the same subject.

2. Effective reuse of polymorphism query results over intraprocedural contexts can be achieved by applying a speculative partial redundancy elimination (PRE) transformation to query expressions.

3. Effective reuse of polymorphism query results for subjects that are fixed over the lifetime of specific instances or modules can be achieved by applying an instance- or module-specific specialization transformation.

Sketch of Transformations

Chapter 4 presents an application of partial redundancy elimination to the low-level implementation of polymorphism operations in the Jikes RVM. Specifically, the approach hoists load operations used in virtual table queries and type tests to earlier control flow locations, allowing subsequent queries to be replaced with the initially loaded results. What makes the approach different from general load elimination is that the semantics of (statically typed) polymorphism imply that memory values in the type structures for a particular object remain unchanged. This obviates the need for sophisticated analyses that model aliasing and memory updates, needed to guarantee the safety of arbitrary load eliminations. Relocating polymorphism queries does, however, require some additional consideration to preserve the semantics of dispatches and type tests applied to potentially `null` references. To address this issue, a novel, on-demand guarding scheme is used to supplement the code motion transform such that queries against `null` references are avoided and precise exception semantics are preserved. Furthermore, to ensure that the transformation is efficient both in terms of compilation and execution savings, a novel approach to redundancy elimination which is both selective and speculative is developed and applied. Overall, the technique is inspired by an understanding of the repetitive use of polymorphism in applications of the ITERATOR pattern.

Chapter 5 presents a novel transformation that generates specialized code for polymorphism operations applied to `final`, reference-type fields of objects and classes. The specialized code is created by inspecting the actual run-time type of such fields and then using the information to reduce virtual dispatches to directly bound calls, type tests to direct branches, and type casts to empty operations (`nop`s) or immediate exceptions. The exact type information also permits the

[9] Effectively this is the problem of determining when two reference are global *aliases*. The problem is intimately related to the type analysis and call graph construction problems discussed in §2.4.1.

reduction of class test and method test inlining guards, as well as explicit and implicit tests against `null` and certain other reference-equality tests. The specializations over class fields are universally applicable since, once initialized, such field values are unique throughout a program's execution. On the other hand, specializations over instance fields require additional treatment since they may differ from one instance to another. To ensure that the correct specialization is invoked for a given instance, the virtual table (the TIB in Jikes RVM) of candidates is replaced with a custom version that dispatches to code specialized for the specific field type bindings of the instance. The advantage of specializing over field types, rather than values, means that the specialized code and custom dispatch structures can be reused for instances with different field values but similar field type bindings. This approach to specialization is inspired by an understanding of patterns like Decorator and Singleton which "wrap" one or more implementation fields in a manner that defers the choice of specific field values until run time, making their exact types imperceptible to ahead-of-time analyses. In cases where the fields of such wrappers are fixed throughout the lifetime of the wrapper, repeated polymorphism operations applied to the fields are clearly redundant with respect to the wrapper. Moreover, constructions like Decorator, Bridge, Proxy, and others may exhibit a variety of inner/outer value combinations, but are likely to exhibit a limited set of reoccurring inner/outer type combinations.

Relevance

Applications of devirtualization and guarded inlining have shown significant improvements in the past, but the true utility of these techniques remains debatable. Profile data, such as that gathered by Dufour et al. [2003], indicate that most virtual dispatches are monomorphic, or near monomorphic for many Java benchmark programs. But what proportion of these operations are intentional uses of polymorphism that are degenerate in practice (i.e., the true targets of semantic optimization), and what proportion are simply examples of object-orientation that do not require polymorphism (i.e., byproducts of Java's use of virtual dispatching for all instance invocations)? Ultimately, this distinction is subjective and not easily quantified. However, to address the need for some objective basis by which to evaluate the success of the optimizations proposed in this thesis, Chapter 3 presents a detailed study of the potential for reuse of polymorphism query results across both intraprocedural and instance-lifetime contexts. The results are contrasted against the utility of guarding with respect to the most common query result—in other words, the potential of an ideal guarded binding strategy. This study provides guidance for understanding the extent to which commonly used benchmarks are amenable to redundant polymorphism elimination, and also provides a basis for evaluating the degree to which the proposed transformations are able to take advantage of inherent redundancy.

Relationship to Design Patterns

If authentic uses of polymorphism are overshadowed by, and effectively indistinguishable from unnecessary uses in the accepted "standard" benchmarks, then what options remain to guide an

empirical study of redundancy across intentional uses of polymorphism? One alternative is to seek other example programs that contain a higher proportion of polymorphic behavior, but this option is problematic since it is open to the obvious criticism that the examples may be chosen to suit the strengths of the solution. Another, more legitimate option, is to consider the general role that polymorphism plays in object-oriented design, and how this role translates into common usage scenarios. As outlined in §2.2.4, design patterns use polymorphism in ways that are particularly characteristic of its modern conception as a tool for dynamic substitution. Moreover, the uses are both intentional and likely to be actively polymorphic in practice. Most importantly, patterns are, by definition, pervasive: they represent constructions that are known to be commonly occurring, even to the extent that they may be present in a program's design without being recognized. For these reasons, patterns stand to form an appropriate basis for the study of redundancy across uses of polymorphism.

However, not all patterns contain the kinds of redundancy considered in this thesis. To achieve a noticeable improvement in performance, the redundancy must occur frequently, be easy to retain rather than recompute, and comprise a significant portion of the overall execution cost. Patterns that are not relevant include Memento and Command, which are more concerned with representing polymorphic state and do not involve significant, recurrent use of polymorphic behaviors. Others such as Observer, Visitor and Interpreter rely primarily on interprocedural polymorphic relationships that are difficult to optimize effectively. And, while uses of creational patterns such as Prototype, Flyweight and factories may entail repeated use of polymorphism, the overhead is often minimal in comparison to the cost of allocation or value lookup associated with each call.

On the other hand, the techniques developed in this thesis follow from a study of patterns that involve repeatedly crossing a polymorphic boundary. The intraprocedural transformation of chapter 4 is based, primarily, on an understanding of the uses of Iterator and Builder. The approach is also relevant for patterns such as Mediator, Strategy, State, and Template Method when their specific applications involve repeated local invocations or type queries. The specialization transform of chapter 5 targets the common idiom of objects that employ a standing delegation relationship where the relation is polymorphic from a static perspective, but fixed from a dynamic perspective. Examples of one-to-one delegation relationships include Decorator, Bridge, Proxy and Chain of Responsibility. Examples with multiple delegates include Facade, Adapter, and Composite; although, in the case of the latter, the delegates are often obscured by an intermediate collection object. The Singleton pattern also represents a form of delegation mediated by a unique metaobject—a class.

It is important to note, however, that the transformations presented in this thesis do not attempt to identify occurrences of particular patterns. Such a strategy is avoided for two reasons. First, current research suggests that automated pattern detection is difficult, and previous work in the area has met with limited success. For example, approaches that focus on static characteristics such as stereotypical relations of hierarchy and/or composition are able to detect some patterns [e.g., Kaczor et al., 2006; Tsantalis et al., 2006], but fail to differentiate patterns that are defined by their

dynamics—notably, the subtle difference between a wrapper and an arbitrary aggregate. On the other hand, approaches such as that of Heuzeroth et al. [2003] which incorporate dynamic information tend to generate numerous false-positive matches, identifying some constructions as patterns when they are not. While disappointing, these deficiencies are understandable given that patterns represent abstract relationships which may not be readily deduced from simple static or dynamic characteristics. Fortunately, although perhaps important for software engineering purposes, it is not necessary to deduce such relationships in order to identify and eliminate redundant polymorphism.

Another reason for choosing not to base a code optimization on pattern detection is that all of the relevant patterns may not be accounted for. Most discussions regarding software patterns focus on the catalogue of the Gang of Four, however this list is by no means exhaustive. Many of the *micro patterns* identified by Gil and Maman [2005] also show potential for redundant intraprocedural and instance-lifetime uses of polymorphism. So does the Dynamic Dispatcher proposed by Büttner et al. [2004]. More importantly, it is reasonable to assume that new, relevant patterns will continue to emerge as the maturity and diversity of object-oriented applications increases.

Thus patterns are best treated as a guide for the work of this thesis rather than the subject of any specific claims. Their structure and dynamics are examined in chapters 4 and 5 to identify common scenarios which are likely to prove relevant for the design of the intraprocedural and instance-specific transformations. To highlight the significance of the insights gained from this study, two synthetic benchmarks are distilled from reoccurring pattern idioms and used to show striking improvements not obtainable using previous techniques.

2.6 Summary

As a programming language feature, polymorphism can trace its lineage back through a variety of approaches to support abstraction and modularization. In its modern form, dynamic inclusion polymorphism is now most commonly associated with the virtual dispatch mechanism of C++. However, for many years the dominance of C++ and its emphasis on inheritance of behavior led to a wide-spread misunderstanding of polymorphism as simply a tool for permitting the partial replacement of existing functionality. This misconception then spread to many users of derivative languages such as Java and C#. In the study of software design, this perception was slowly corrected by the development of object-oriented design methodologies which used subtyping rather than subclassing polymorphism to create decoupled architectures with opaque interface boundaries. The design patterns approach, in particular, was instrumental in establishing that polymorphism is most useful when seen as a tool for enabling the dynamic substitution of new variants, rather than the extension of old forms and behaviors. However, this view of polymorphism remained implicit in object-oriented design methodologies, and was not articulated through any set of clear guidelines for its application. As a consequence of this lack of clarity, the need for a similar reinterpretation of polymorphism was not perceived by the language implementation and optimization communities.

Following the development of the virtual function table, subsequent efforts to improve the efficiency of polymorphism focused almost exclusively on bypassing its underlying mechanisms. Four

main techniques emerged: static devirtualization, inline caching, guarded inlining, and specialization. When applied to languages like Java and SELF, which employ polymorphic dispatch by default, these approaches were all reasonably successful at reducing the cost of superfluous polymorphism. However, their efficacy with respect to intentional uses of polymorphism remains questionable.

A new opportunity for improving the performance of actively polymorphic code lies in accepting that some queries need to be executed, but never more than once for a given subject. Thus shifting the perspective from a site-oriented view, which looks for independent optimizations applicable to all execution contexts, to a subject-oriented view which seeks to eliminate redundancy with respect to the properties of a specific object. In particular, adaptations of partial redundancy elimination and specialization can be employed to eliminate subject-specific redundant polymorphism queries across intraprocedural and instance-lifetime contexts. Furthermore, a study of the use of polymorphism in object-oriented design patterns can serve to enhance these techniques by suggesting specific scenarios which are likely to be relevant and recurrent.

2.6.1 Contributions

This chapter provides a number of research contributions in its own right. It builds on the survey of Ryder et al. [2005], presenting a historical review of the development and adoption of inclusion polymorphism with a particular emphasis on the influence that design methodology (§2.2) has had on implementation (§2.3) and optimization strategies (§2.4). Studies examining the execution cost of polymorphism are synthesized (§2.3.1) to reveal issues relevant to Java programs, and specifically their execution on the Jikes RVM. In particular, the discussion notes that additional register pressure incurred by polymorphism queries is an important issue for Intel x86 architectures—a factor not addressed by previous studies.

The motivation for this thesis (§2.5) presents a new perspective on the optimization of query computations. The idea is to ask not what can be eliminated, but what can be reused. The approach suggests viewing query subjects as defining the context of redundancy, rather than a subject-agnostic code location. This view is particularly apt for the patterns approach to object-oriented design which focuses on the repetitive use of polymorphic objects in certain archetypal roles. Moreover, the proposed optimization strategies derived from this view are notably the first to target the general characteristics of design patterns, whereas previous attempts have either required programmer-supplied annotations or focused on the peculiarities of specific patterns.

The contexts outlined in §2.5.1 and their relation to various patterns (§2.5.2) also suggest a new approach to pattern classification. In their catalogue, the Gang of Four described patterns as creational, structural, or behavioral. Gil and Lorenz [1998], and Agerbo and Cornils [1998] both take a different view, arranging patterns by their proximity to language constructs. Zimmer [1995], on the other hand, classifies patterns based on how they are composed or derived from each other. This view organizes patterns based on static relationships. Redundancy contexts can be used to similarly organize patterns by the nature of their dynamic relationships. Patterns that exhibit intraprocedural redundancy, such as ITERATOR and BUILDER, act in a purely *subordinate* role, taking

direction from some higher-level authority. Alternately, patterns with interprocedural redundancy, such as OBSERVER and VISITOR, establish a more *collaborative* relationship where no one object can be viewed as dominant. Finally, patterns such as DECORATOR and SINGLETON show redundancy that persists over the lifetime of an instance or module, typically exerting a purely *dominant* role over their inner value.

2.6.2 Directions

Before moving on to consider the details of redundant polymorphism elimination (RPE) covered in chapters 4 and 5, it is worth noting that the analysis in this chapter already suggests several new directions for future work.

Object-Specific Optimizations

The thesis outlined in §2.5.2 focuses on the immutable properties of objects that are used to implement inclusion polymorphism. As suggested in earlier work [Brown and Horspool, 2006], other properties fixed at allocation time, such as an array's `length` and an object's *identity hash code*,[10] also stand to benefit from a similar subject-oriented approach to redundancy elimination. This idea could be carried further to consider all loads of `final` instance fields, and possibly even executions of instance methods proven to be *pure*—that is, without side effects or mutable state dependencies. However, there is an important caveat to optimizing over arbitrary object properties: the semantics of Java's `final` modifier permit fields to be set any time before the normal completion of the object's constructor. Thus there remains a window over which the value of `final` fields may actually change. Any sound approach to eliminating redundant uses of such fields (or calls to methods that use them) would require a guarantee that the subject of the load or call be fully constructed. Providing such a guarantee is no simple feat though, since a subject reference may escape to other contexts during the execution of its constructor.

Trace-Oriented Polymorphism Optimization

The discussion of §2.5.1 identifies four contexts relevant to polymorphism redundancy. Of these, the interprocedural context is discounted since communicating across method boundaries introduces additional costs that may outweigh any savings due to redundancy elimination. There is, however, a potential fifth context which could alternately be used to eliminate interprocedural redundancies.

The notion of specializing with respect to a trace—a specific path of execution—rather than a method is briefly mentioned in §2.4.3. In this approach, method boundaries are irrelevant: a trace includes any sequence of instructions likely to execute to completion. Because of its lack of control flow, optimized trace code typically includes a number of guarding tests to ensure that intermediate computations produce the same results seen during trace construction. At present though, approaches to trace-oriented optimization do not consider guards to prevent redundant

[10] As associated with reference equality, not computed equality.

load operations. However, polymorphism queries represent a unique class of load operations where the results are guaranteed to be invariant, assuming a static typing system. Indeed, a simple test against an object's exact run-time type could be used to guard a direct binding of all polymorphism operations over a given trace.

Pattern Discovery

In developing an organization of design patterns based on their static relationships, Zimmer [1995] discovered and defined a new pattern, OBJECTIFIER, as a generalization of uses seen in BRIDGE, BUILDER, STRATEGY, and others where functionality is promoted to an object abstraction. This suggests that a further study of the dynamic object relationships—subordinate, collaborative, and dominant—implied by the various contexts of redundancy (§2.6.1) could also lead to the discovery of new patterns, and/or the generalization of existing patterns. For example a recognition of the subordinate nature of ITERATOR, BUILDER, and others could form the basis of a more general "state machine" pattern. Similarly, a recognition of the dominating character of DECORATOR, ADAPTER, and others could provide a clearer articulation of the "wrapper" concept.

Chapter 3
Practical Foundations

It is no simple matter to quantify the improvements offered by the code optimizations outlined in the previous chapter. Each transformation will certainly affect different programs to different degrees, but what basis should be used to evaluate whether the change is positive, significant, and reproducible? Also, what factors have the potential to overshadow or interfere with the transformations, masking their effects? This chapter examines these questions, and presents a methodology used to characterize performance improvement in subsequent experiments. The discussion includes an overview of the virtual machine system within which the code transformations are implemented, a description of the benchmark programs used to evaluate the transformations, and several analyses of the system and benchmarks which provide context for the discussions to follow.

The primary purpose of this chapter is to underscore the complexity of quantifying the performance of compiler transformations—in particular, those that focus on dispatching—which are implemented in a modern, adaptive virtual machine and run on a modern hardware platform. It is worth noting that, when compared to similar contemporary research, such an elaborate preparatory analysis is rare. However, the need for an in-depth discussion is supported by several recent studies [Georges et al., 2007; Mytkowicz et al., 2009] which suggest that many published results (using similar experimental platforms) lack statistical rigor, may be misleading, and understate the potential for instability.

The results and analyses in this chapter were developed to provide context, some hypotheses regarding impact, and a rigorous measurement framework for the main experiments of this thesis presented in chapters 4 and 5. Fortunately, many aspects of the work are also novel in their own right and applicable beyond the scope of this thesis.

3.1 Experimental Environment

The experiments in this thesis were performed on an Intel Pentium D ("Smithfield") 3.2GHz processor running Ubuntu Linux 8.04 Server Edition (kernel version 2.6.24, with SMP support), with a total of 3GB of 800MHz DDR RAM. At present, this platform is roughly four years old, however it does represent the first in Intel's line of dual-core processors, and includes a number of the key advances developed for the Pentium 4 ("Prescott") architecture [Intel, 2009a] which is used to implement each of the two cores. As alluded to in §2.3.1, features of the Prescott such as its very deep (31-stage) pipeline, enhanced branch-prediction techniques, and limited set of registers are all

likely to have a significant influence on the efficiency of virtual dispatching and type tests, as well as on techniques for relocating or removing these structures.

3.1.1 Compilation and Execution: The Jikes RVM

All of the experiments in this thesis are based on the *Jikes RVM* (Research Virtual Machine), version 3.0.1.[1] Jikes RVM is an implementation of the Java Virtual Machine Specification [Lindholm and Yellin, 1999] which is designed to support research and experimentation relating to the compilation and execution of Java programs.

Unlike many JVMs, Jikes RVM does not contain a bytecode interpreter. Instead, all input bytecodes are compiled to native code using one of several just-in-time (JIT) strategies. More precisely, Jikes RVM uses an *adaptive optimization* scheme (outlined in figure 3.1) to compile and then recompile code using increasingly aggressive optimization techniques.

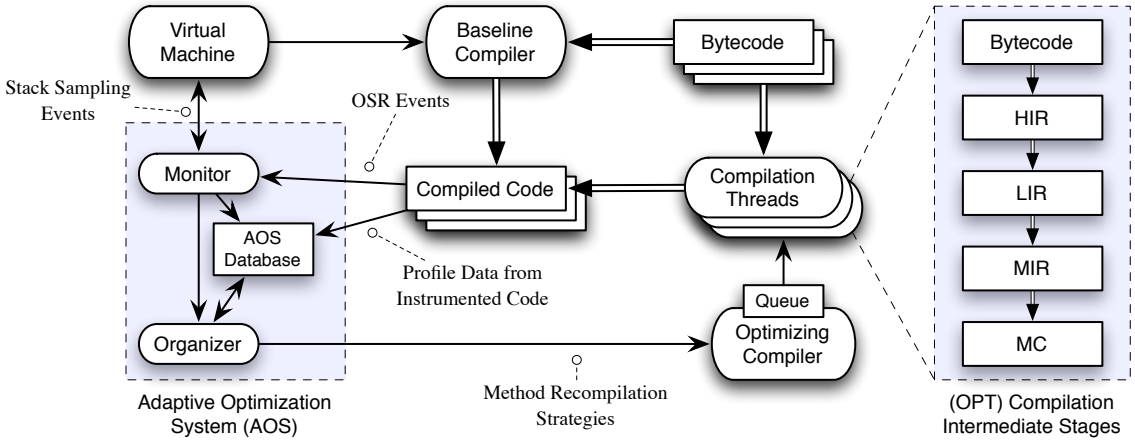


FIGURE 3.1: **Adaptive Optimization in Jikes RVM**

Initially, the *baseline compiler* generates native code from templates for each bytecode instruction. This compiler is fast, but generates relatively inefficient code. While a program executes, the *adaptive optimization system* (AOS) periodically samples the execution stack to determine which methods are frequently active. When a method is identified as "hot" (i.e., frequently active), it is queued for recompilation using the *optimizing compiler*—an approach known as *selective optimization* [Arnold et al., 2000]. Some methods are also identified through on-stack replacement (OSR) events which indicate that recompilation is needed to satisfy new requirements revealed by dynamic class loading.

The optimizing compiler translates bytecode into a high-level intermediate representation (HIR). The HIR contains instructions very similar to Java bytecode but represents local data using typed,

[1] Jikes RVM was originally known as *Jalapeño* [Alpern et al., 2000]. The new name, adopted in 2001, was conceived as an extension of the *Jikes* open-source Java compiler, although the two projects developed independently and remain unrelated.

symbolic variables rather than stack manipulations. Control flow is represented more explicitly than in bytecode by grouping instructions into *basic blocks* which, in turn, are linked to form an explicit *control flow graph* (CFG). The intermediate representation is amenable to many standard code optimizations such as copy propagation, constant folding and control flow restructuring (loop unrolling, inversion, etc.). After optimizations appropriate to the HIR are applied, the code is translated to a low-level form (LIR) which exposes the instructions that comprise complex operations such as type tests and virtual dispatching. Standard optimizations are then applied again to capture effects relating to the newly revealed LIR values. Next, the LIR—a machine-neutral representation—is reduced to a machine-specific form (MIR) which parallels the target instruction set. The final stage of translation to MIR assigns symbolic variables to actual registers. Once in this form, the code can be directly assembled to binary machine code (MC). Jikes RVM has the capability to generate code for both the PowerPC and Intel x86 instruction sets, however experiments in this thesis focus exclusively on the latter.

Unlike the baseline compiler which operates on-demand,[2] the optimizing compiler operates concurrently with the running program, installing improved method implementations as they become available. In this way, a program's effective implementation changes over time, with performance (ideally) improving incrementally the longer a program continues to run. In fact, the same method may be recompiled several times during the execution of a program. Normally, recompilation proceeds through three distinct levels of optimization (O0, O1, and O2), each employing more time-consuming analyses and aggressive transformations than the previous. Jikes RVM's adaptive optimization system uses a heuristic function to estimate when compilation at a particular optimization level is likely to yield a net benefit. Methods are promoted to the next level when the predicted savings in future execution time exceed the compilation effort required [Arnold et al., 2004].

Jikes RVM's compilation system has several features which make it an appropriate choice for the work of this thesis. First, it exposes the instructions and data structures used to implement polymorphism operations. This is important since, unlike many other optimizations for Java, the techniques described in chapters 4 and 5 cannot be achieved through manipulation of bytecode alone, they require access to hidden values such as each object's TIB reference. The second feature, is Jikes RVM's ability to capture on-line branching statistics and use them to infer execution-specific instruction frequencies, permitting focused approaches to speculative optimization. Finally, the fact that the compilation environment is integrated with the run-time environment permits actual run-time values to be inspected and incorporated into optimized code in ways that are not possible with ahead-of-time techniques. Access to these features does, however, come at a price. Jikes RVM's compilation system is complex and its effects on performance are challenging to quantify accurately. The following sections provide some insight into the source and nature of these complexities, and provide a framework for understanding their role in later investigations.

[2] Since many classes include code that is never used for certain executions, the baseline compiler employs a *lazy compilation* strategy [Krintz et al., 2001], generating code one method at a time, only as needed.

SPECjvm98 *Suite*

201_compress	Lempel-Ziv compression; a port of 129.compress from SPECcpu95
202_jess	Java Expert Shell System: compute logical inferences from a set of "facts"
209_db	Database operations (find, update, etc..) applied to memory-resident records
213_javac	Java compiler from Sun's JDK 1.0.2 applied to JavaLex source code
222_mpegaudio	Decodes 4MB of MPEG Layer-3 audio data
227_mtrt	Multithreaded ray-tracing application; two concurrent threads
228_jack	Scanner and parser generator run on its own source code

DaCapo *Suite*

antlr	Scanner and parser generator; processes 44 grammar files
bloat	Java bytecode optimization tool; analyzes one class and all its dependencies
chart	Plots 9 line graphs, rendering them as PDF
eclipse	non-interactive Java IDE tasks: compilation, code analysis
fop	Parse and render a XSL-FO file as PDF
hsqldb	20 client threads execute transactions against a simulated banking database
jython	Python interpreter; executes pybench benchmark
luindex	Use lucene to index terms from Shakespeare and the King James Bible
lusearch	Use lucene to search Shakespeare and the King James Bible; 32 search threads
pmd	Java source code diagnostic tool
xalan	XSLT processor; 8 threads transform XML to HTML

Polymorphism Benchmarks

L1	Simulate Iterator and Builder with a variety of receivers to produce intraprocedurally redundant polymorphism that is not amenable to inlining or devirtualization.
L3	Simulate Decorator with a variety of object couplings to produce instance-specific redundant polymorphism that is not amenable to inlining or devirtualization.

Table 3.1: **Benchmark Program Descriptions**

3.1.2 Evaluation Benchmarks

To provide a basis for comparing the effects of redundant polymorphism elimination against other optimizations, this thesis presents results gathered from executions of two widely used Java benchmark collections: the SPECjvm98 suite and the DaCapo suite, version 2006-10 [Blackburn et al., 2006]. The programs contained in each suite are summarized in table 3.1. Two other synthetic benchmarks are also included that focus on the polymorphism redundancy scenarios in design patterns reviewed in §4.1 and §5.1. Other benchmarks, such as SPEC's jbb2000, jbb2005, and jvm2008 suites, are not considered since they do not produce timing results compatible with the measurement methodology outlined in §3.2.1.

The SPECjvm98 suite is included since it is the most widely recognized set of Java benchmarks. However, there remains an ongoing debate as to whether these programs are truly representative of modern Java applications. For example, the compress and mpegaudio programs—which are mostly derived from earlier C language benchmarks—work mostly with arrays and allocate few objects. On the other hand, the db program uses many objects, but it contains just three application classes and

primarily operates on only one kind of object. As a response to many of the perceived problems with SPECjvm98, the DaCapo suite was developed to provide a more realistic set of benchmarks. Its programs show a wider diversity of memory allocation behaviors which are taken to be more representative of typical Java workloads. The DaCapo programs also execute over a much larger code base for longer periods. Additionally, the programs are mostly based on open-source code with a longer development history than the SPECjvm98 programs. This aspect is important for the current examination since Riehle [2009] has argued that the *design pattern density* of an object-oriented system typically correlates with its maturity. By extension, this suggests that examples of redundant polymorphism associated with design patterns are more likely to be found in programs with a longer development history. For these reasons, the outcomes associated with DaCapo programs are treated as more relevant for the evaluations in this thesis.

The L1 and L3 benchmarks are included to highlight the degree to which the techniques of chapters 4 and 5 can improve the efficiency of polymorphism where other techniques cannot. These programs have simple workloads which make highly repetitive use of virtual dispatching and type tests. They are designed to use several polymorphic subjects in a manner that subverts typical site-oriented optimization strategies. Their names are meant to be associated with the contextual "level" of redundancy they stress: intraprocedural for L1, and instance-lifetime for L3. A more complete description of these programs is provided in appendix C.

Benchmark Characteristics

Table 3.2 presents several properties of the studied benchmarks as measured by Jikes RVM. The first two columns provide information about all the code that passes through the baseline compiler. The next four columns show information for only the "hot" code that passes through the optimizing compiler. Transformations such as guarded inlining and redundant polymorphism elimination are only applied to the hot code. Also, some methods pass through the optimizing compiler several times, and thus may be represented several times. Since the selection of hot methods is non-deterministic, the values shown represent a mean over 10 *benchmark executions*. In most cases the given values are derived from the arithmetic mean of measured quantities, except for the suite aggregates of density, expansion, and effort ratios which are computed using the harmonic mean.

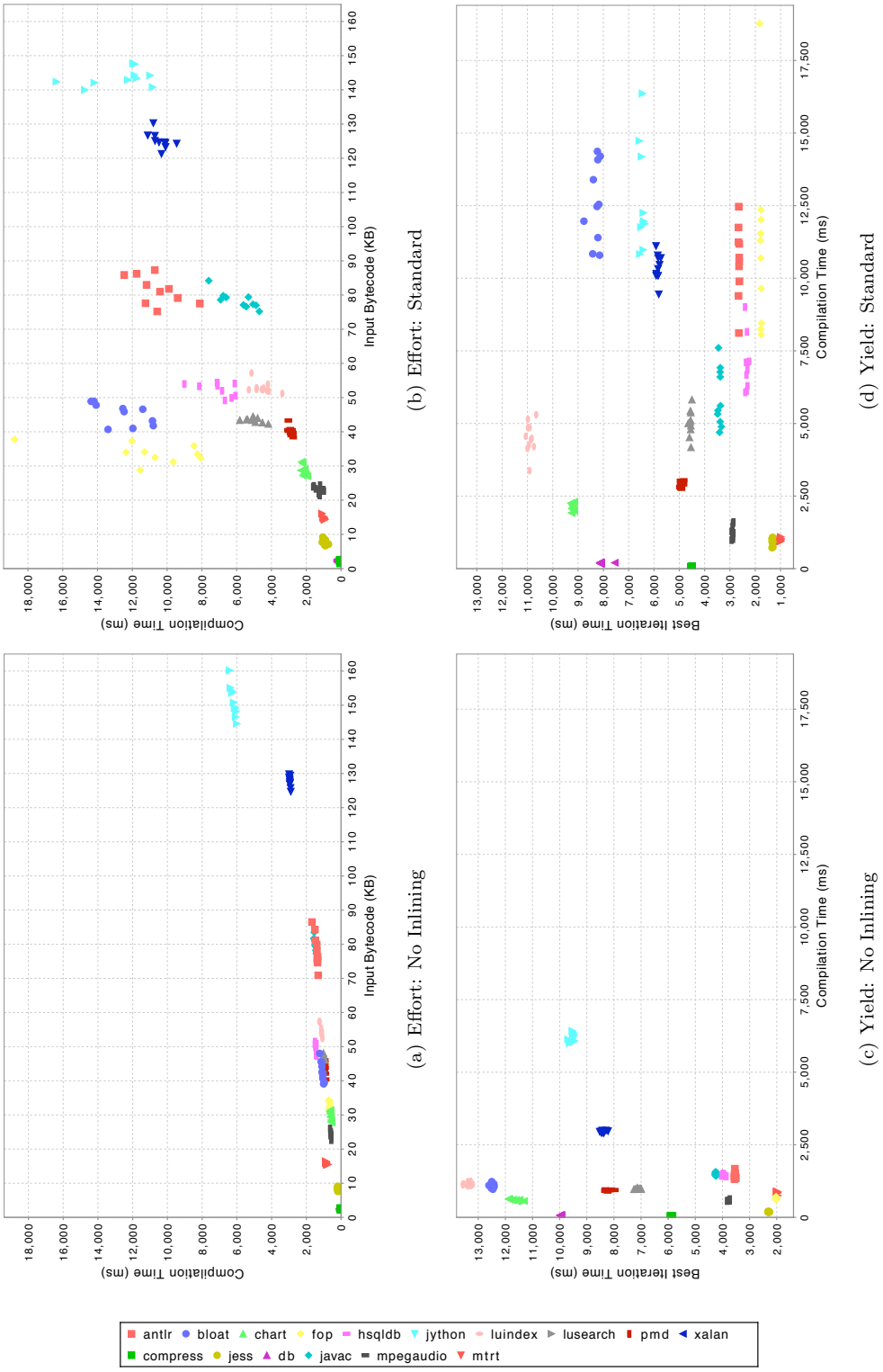
The metrics in table 3.2 reveal a number of characteristics that are worth bearing in mind when reviewing the performance evaluations given in later sections. The quantity (size) of optimized code provides a rough measure of the diversity of optimization opportunities. For example, compress, jess, and db all have rather narrowly focused main processing loops, whereas the hot execution paths of eclipse cover a code base that is two orders of magnitude larger. In considering the relevance of compilation strategies, improved results for the latter—and other large programs—are much more indicative of a broadly applicable technique. Another dimension of relevance beyond simple coverage is how well a technique works in the face of complex code. Method *density* (instructions per method), provides a simplistic measure of complexity based on the assumption that larger methods are likely to contain more elaborate control flow. However, in comparison to the other results, the

	All Code		*Optimized Code*				*Best Iteration*	
Benchmark	*Size*	*Density*	*Size*	*Density*	*Expansion*	*Effort*	*Mean*	*(dev)*
antlr	216.2	142.5	81.5	211.8	13.4	129.8	27.3	(3.8)
bloat	178.3	90.5	45.2	142.2	23.1	279.2	20.8	(6.3)
chart	216.5	95.3	28.5	162.5	8.6	74.5	16.9	(7.5)
eclipse	819.6	83.1	322.0	176.5	7.3	57.1	19.1	(7.9)
fop	240.4	83.6	33.7	145.1	23.4	329.3	26.6	(4.7)
hsqldb	140.1	96.3	52.5	189.9	9.7	134.2	22.1	(6.3)
jython	819.7	124.3	143.5	243.6	7.8	88.4	27.6	(2.9)
luindex	83.8	91.0	52.9	193.6	9.8	85.4	18.7	(8.2)
lusearch	69.3	86.6	43.4	189.6	11.6	115.7	23.4	(7.0)
pmd	176.0	69.4	40.2	113.3	8.9	72.0	25.7	(4.4)
xalan	184.2	73.5	125.2	139.8	9.7	83.0	28.4	(3.0)
compress	24.6	87.6	2.0	140.7	4.5	30.0	15.6	(7.5)
jess	43.2	63.5	7.7	99.4	12.7	120.7	20.8	(5.8)
db	24.5	83.9	2.2	112.6	10.1	88.2	15.5	(7.6)
javac	89.7	89.8	78.5	193.8	7.9	75.1	29.3	(1.2)
mpegaudio	69.2	151.0	23.1	185.1	6.3	54.8	20.8	(6.5)
mtrt	33.2	80.7	15.0	180.4	8.1	66.9	21.2	(6.0)
jack	54.4	105.2	25.0	226.7	9.4	98.4	21.7	(8.2)
L1	14.3	52.1	1.1	22.0	5.1	51.1	17.2	(5.8)
L3	14.2	55.7	0.8	29.8	5.9	53.2	11.3	(9.2)
Mean								
DaCapo	285.8	90.5	88.1	166.0	10.5	99.7		
SPECjvm98	48.4	88.9	21.9	150.3	7.6	63.9		

Size – *Input Bytecode (KB)*
Density – *Input Bytecodes/Method*
Expansion – *Machine Code Bytes/Input Bytecode*
Effort – *Compilation Time/Input Bytecode (ms/KB)*
Best Iteration – *Mean Iteration Number with Fastest Execution Time*

TABLE 3.2: **Benchmark Code Characteristics**

density of hot code from the two main suites is actually quite uniform. To provide more depth, two other complexity measures are given. The code expansion factor indicates how much machine code is used to represent the input bytecode instructions. Some bytecodes, such as array stores and type tests, are implemented with many machine code instructions, whereas branches and arithmetic operations typically have a one-to-one mapping onto machine code. For some programs, very high expansion values are indicative of frequent inlining where dispatch instructions are replaced with all the code from the body of the dispatch target. While inlining does create more complex control flow, a broader measure is the *effort* (time per instruction) expended by the compiler to analyze and translate the code. Naturally, more effort is required for more complex code. An important observation is that, overall, the DaCapo programs present a larger and more complex code base to the optimizing compiler. These results serve to reinforce the earlier discussion regarding the



(a) Effort: No Inlining (b) Effort: Standard

(c) Yield: No Inlining (d) Yield: Standard

FIGURE 3.2: **Variation in Compilation Results**

relevance of the DaCapo programs in comparison to those from the SPECjvm98 suite.

However, the metrics of table 3.2 only represent a first step toward understanding the dynamics of these programs. What is not revealed by the table is the degree to which some measurements can vary from one benchmark execution to another because of the non-deterministic action of the adaptive optimization system. For example, even though the input code size and density values remain quite stable from run to run—indicating that similar code is being selected for optimization—the expansion and effort values can vary significantly when inlining is enabled. These effects are illustrated in figure 3.2 which plots the individual results for each execution (table 3.2 is based on the "Standard" configuration). With no inlining, compilation time has a clear linear relationship to the input code size, and the total time expended is similar over multiple runs. However, inlining is a significant optimization in Jikes RVM with an important relationship to polymorphism, so disabling it to improve stability would simply be inappropriate. Enabling it, on the other hand, introduces some curious behavior. Many of the programs spend significantly more time compiling, but with no obvious relation to the amount of input code. And perhaps more importantly, although more effort is expended on compilation this, on its own, does not produce faster code.

One possible explanation for this unusual effect is that, in addition to inlining frequently used methods, Jikes RVM's strategy can be overly aggressive and inline many calls unnecessarily. Such unwarranted inlining occurs when the optimizing compiler prematurely perceives a skew in the calling patterns inferred from the on-line statistics—for example, basing a decision on the fact that a sub-call is made in 3 out of 5 observed occurrences rather than 600 out of 1000 occurrences.[3] This extra inlining requires significantly more work to analyze and transform the resulting code, but offers little performance gain since much of the inlined code occurs on infrequently executed paths. While an interesting curiosity, this behavior also has significant implications for the primary goal of measuring performance. When the optimizing compiler is retarded by dubious inlining decisions, this incurs a delay in the queue of methods to be optimized (see figure 3.1). Over time, such delays—along with other system load factors—reshape the decisions made by the AOS since the main program continues to execute with certain hot methods not working at their full, optimized potential. Given enough time, the AOS does eventually identify and optimize the most relevant methods, however the key point is that the time at which the best code is available is highly variable from one execution to another. As the next section reveals, this behavior can significantly complicate efforts to quantify performance improvements due to changes in compilation strategy.

3.2 Performance Measurement

Previous studies of Jikes RVM's sampling methodology have confirmed that, on average, the adaptive optimization system is successful at identifying active methods [Arnold et al., 2000], and inferring the frequency of calls between them [Arnold and Grove, 2005]—the latter being necessary to make effective inlining decisions. What this means is that, even though adaptive updates occur non-deterministically, over long program runs the most active execution paths receive a similar opti-

[3] This pathology is prominent in the execution behavior of the L1 and L3 programs, and is discussed in appendix C.

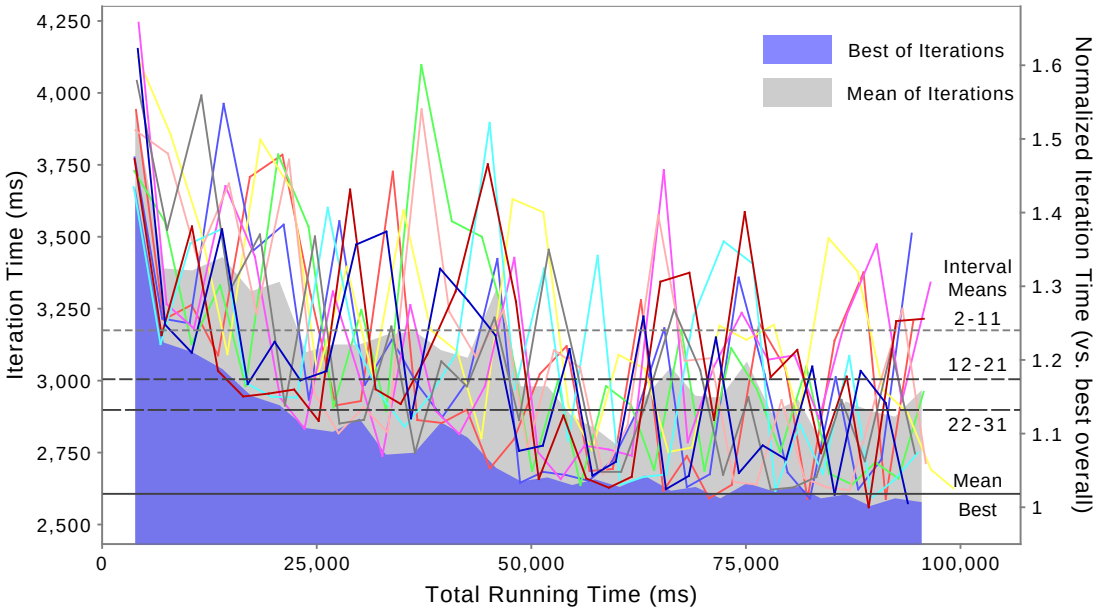


FIGURE 3.3: **Performance Variation of** antlr

mization treatment. As a result, the overall throughput of long executions remains fairly consistent. This behavior is certainly beneficial for those interested in measuring total system throughput, but it belies the actual amount of short-term variation in performance that can occur over the course of a program run. Moreover, these summative results regarding the long-term, average behavior of Jikes RVM have instilled a false understanding among many researchers that, given enough time, a program run will eventually "converge" to a stable, maximal performance level.

Figure 3.3 provides a reasonably stark refutation of this view by overlaying intermediate results gathered from 10 different executions of the antlr benchmark program. Each of the coloured lines represent the time to complete an iteration of the benchmark's main loop plotted against the total running time. It is plainly evident from the graph that there is no consistent pattern in the progress of iteration scores over the course of an execution. Even the average of scores across executions at each iteration (shown as the grey, shaded area) do not exhibit a stable evolution.[4]

Many factors contribute to this variability in iteration scores. Garbage collection is often touted as a primary contributor of instability, although the SPECJVM98 and DACAPO programs do invoke a full heap collection between iterations, and incur a similar allocation load over each iteration. External system load certainly has some influence too, however steps are taken to minimize the number and effect of other processes during benchmark execution. The introduction of new, optimized code by the AOS also has an effect on performance. However, one would expect that installing improved

[4] Most of the other benchmarks show a similar, non-monotonic progression in iteration scores, with variability in the range of 5-20% relative to the best result. Some (e.g., fop and hsqldb) actually exhibit extreme variations of more than 100%. Other researchers have noted similar, corroborating observations in postings to the Jikes RVM mailing list at http://lists.sourceforge.net/lists/listinfo/jikesrvm-researchers (cf. 7-May-2009 & 4-Dec-2009).

method implementations should lead to progressively better results, not the erratic behavior seen.

What is not evident is that the AOS does, in fact, make steady improvement. To reveal this progress it is necessary to synthesize results from numerous runs. The blue, shaded area plots the best (fastest) score observed at each iteration over all executions. Combined with the knowledge that the code selected for optimization is mostly the same for each run, this projection provides an effective way to filter out the perturbations caused by other factors, and show the maximal potential of the optimized code after a certain amount of monitoring and adaptive updates. In particular, the plot clearly reveals the progression towards a limit on the effect of adaptive compilation. Since the specific focus of this thesis is on analyzing the effects of alternate compilation strategies, this limit is a key quantity of interest, as opposed to other, more general measures of throughput.

3.2.1 Measurement Methodology

Accurately and consistently measuring the performance of Jikes RVM is no simple matter. As the previous discussion highlights, simple reductions such as the best overall result, or the median of a particular iteration measured over a few executions do not provide an adequate summary of "typical" execution behavior. Indeed, Georges et al. [2007] demonstrate that a number of published methodologies similar to these are not statistically robust, and may convey misleading results.

For characterizing long-run performance, Georges et al. recommend averaging scores over a window of 10 iterations per execution, and then taking the mean of these results over 10 executions to obtain a statistically meaningful result. These particular values are demonstrated to yield confidence intervals (as determined by Student's t-distribution) for the sample means which are typically precise enough to distinguish performance changes of 1–2%. Crucial to this approach is determining which 10 iterations comprise the measurement interval. As figure 3.3 indicates, choosing a fixed interval (as some do) can be problematic. The dashed lines illustrate the results of sampling from the 2^{nd}–11^{th} iterations, the 12^{th}–21^{st}, and the 22^{nd}–31^{st}. (The first iteration is discarded since it includes a significant initialization overhead.) Clearly each choice leads to a distinctly different mean result, and none seem to provide a preferable measure. To address this problem, Georges et al. advocate using a dynamic method for identifying the measurement interval wherein the first 10 scores with a *coefficient of variance* (CoV) less than 0.02 are used to form the sample.[5] However, on the test system, using the standard "production" configuration for Jikes RVM, such an interval was not consistently obtainable over 31 iterations of antlr, eclipse, fop, hsqldb, xalan, or mtrt.[6] This is, of course, understandable given the variability of antlr seen in figure 3.3. Plots for the other programs exhibit similar eccentricities. What is also clear is that extending the executions to perform more iterations is not necessarily a viable solution. In fact, several of the benchmarks—notably, eclipse, fop, jython, and mtrt—actually exhibit more variation at later iterations. More importantly though, the sampling-window approach, when it works, is best for capturing the overall system performance, including the effects of garbage collection, caching disruptions, anomalous page faults, etc. It does

[5] The coefficient of variance for a sample is defined as the sample deviation divided by the sample mean.
[6] Georges et al. recommend 30 iterations.

not provide an effective technique for isolating the outcome of changes in compilation strategy.

To provide an alternative measurement technique that is more focused on the changes in code potential, this thesis uses the following performance evaluation methodology adapted from Georges et al.:

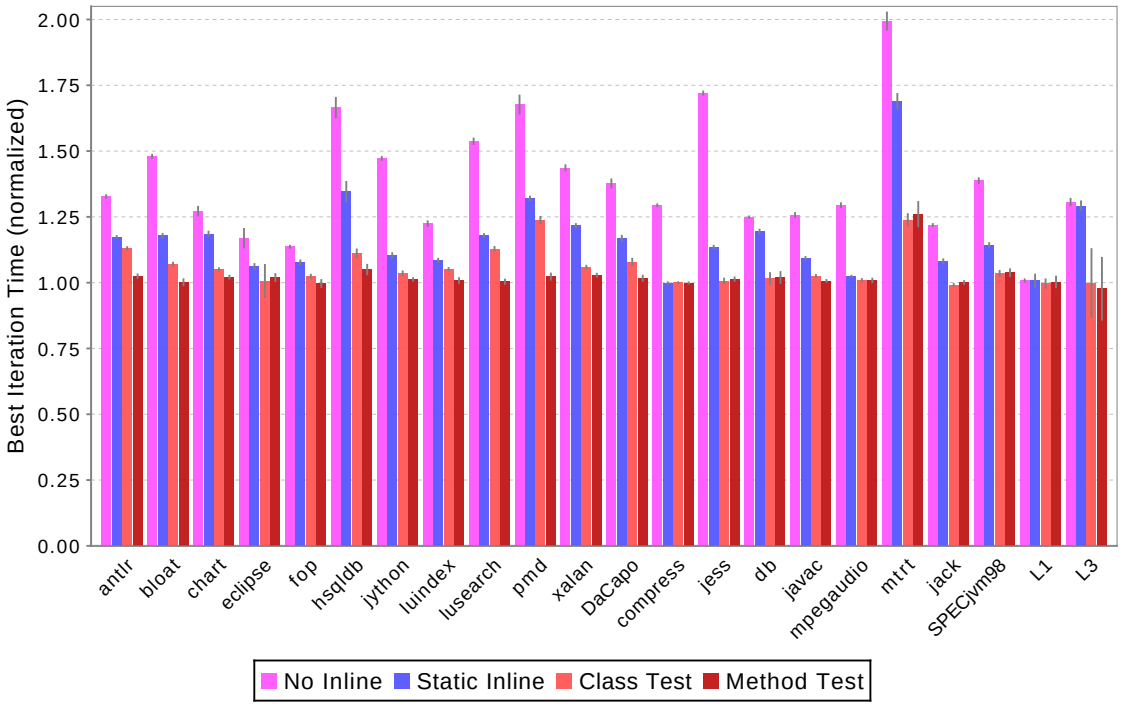
- The heap size is set at 5 times the minimum required for each program.
- Each benchmark is executed at least 11 times, discarding the first as a warm-up run.
- The best score (lowest time) over 31 iterations, regardless of when it occurs, is extracted from each execution and the values averaged using the arithmetic mean.

The result of this strategy is plotted as the "Mean Best" in figure 3.3. This measure clearly provides an effective approximation of the lower limit of the code potential—in other words, the typically achievable maximum performance. And, although the best score may be extracted at any point during an execution, the far right column of table 3.2 confirms that, on average, the best result is observed well into the execution, but before the 30^{th} iteration.

Georges et al. also suggest that measurements be taken over several different heap sizes, but this is primarily in the context of experimental changes to the garbage collection strategy, and so is not necessary when focusing solely on changes in compilation strategy. The one notable weakness of the proposed measurement scheme is that trials are only performed on a single hardware platform. Mytkowicz et al. [2009] argue that this kind of limited focus can introduce a systemic bias in results. However, since only one platform was available, the trials are limited in that regard. Certainly an area for future work would be to provide comparative results for other platforms, in particular the PowerPC which is Jikes RVM's other major target architecture, and notably quite different from the Intel x86 platform.

3.2.2 Baseline Results

The upcoming evaluations in §4.3 and §5.3 present performance comparisons against two different baseline Jikes RVM configurations: one that allows inlining of fixed ("static") targets—such as system calls, constructors, and `static` methods—but does not alter the usual polymorphism operations, and one that allows guarded inlining using method tests and class tests (see §2.4.2). To provide a sense of the relative speed of these configurations, without any subject-oriented optimization, figure 3.4 contrasts the performance effects of the four major inlining strategies. Since the durations of the benchmarks vary significantly, the performance scores—measured as the mean-best result—are given as values normalized against the performance of runs on Jikes RVM's standard ("production") configuration. Thus, in this and subsequent similar figures, results below 1.0 indicate improvement over the stated baseline which is given as the last configuration alternative. The thin line overlaid on top of each bar represents the span of the 95% confidence interval for each best-result sample. Aggregate results for the two benchmark suites are given to the right of the programs they contain, and are calculated as the *harmonic mean* of normalized performance scores. It is worth noting that the prevailing tendency is to report the *geometric mean* of performance improvements



No Inline	–	*Do not inline any methods*
Static Inline	–	*Only inline methods with fixed targets (e.g., static methods and constructors)*
Class Test	–	*Speculatively inline using class test guards only*
Method Test	–	*Speculatively inline using method test and class test guards*
Baseline	–	*Jikes RVM default: Speculatively inline using preexistence, patching, and guards*

FIGURE 3.4: **Effect of Inlining Optimizations**

[e.g., Suganuma et al., 2001; Bacon et al., 2002; Blackburn et al., 2006], however John [2005] makes a cogent argument for using the harmonic mean to average such ratios. The raw score values are listed in §A.2.

The main conclusion made apparent by figure 3.4 is that inlining leads to significant gains for all of the standard benchmarks. Overall, the DACAPO programs are 38% slower with no inlining, and 17% slower with only static inlining. Similarly, the SPECJVM98 programs are, respectively, 39% and 14% slower. These benefits are understandable given that inlining removes not just the call and return instructions, but also many reads and writes involved in stack manipulations, as well as threading yieldpoints which are contained in the prologue and epilogue of calls. This, of course, is in addition to the opportunities that inlining creates for interprocedural optimizations by allowing specialized parameter values to be propagated through the inlined code from the calling context.

Another important observation, relevant to the evaluations given in chapters 4 and 5, is that the "Method Test" configuration, which is often used as a baseline, is nearly equivalent to Jikes RVM's standard configuration, with the exception of mtrt which receives an additional boost from specu-

lative, unguarded inlining (i.e., "patching" – cf. §2.4.2). Thus, in general, improvements relative to the Method Test configuration represent realistic advancements over the state of the art.

3.3 Polymorphism Characteristics

As suggested in §2.5.2, there are good reasons to suspect that redundancy in the use of polymorphism does occur in code that is based on mature object-oriented design principles, especially code that employs common design patterns in its architecture. But how much of this kind of redundancy actually occurs in the studied benchmarks? If a benchmark shows no improvement when the proposed optimizations are applied, should this be considered a failure of the techniques? Or is it simply an indication that the program makes no significant use of polymorphism? Figure 3.4 shows consistent improvements when guarded inlining is applied, but how much of this is attributable to the technique's ability to pierce degenerate examples of polymorphism, and how much is truly an adaptation to actual polymorphic behavior? Perhaps the author of a particular code segment intended to support polymorphic behavior, but none actually occurs when using the standard benchmark inputs. To address these issues and provide context for interpreting the results given in later chapters, the following sections review some previous studies of polymorphism and devirtualization, and provide a comprehensive analysis of the redundancy observed in the studied programs.

3.3.1 Devirtualization Potential

As outlined in §2.4.1, many approaches to type analysis and call graph refinement have been proposed for Java. These techniques are needed to support devirtualization, but also serve as a foundation for many other whole-program analyses and optimizations. Thus, their utility is typically assessed by measuring the reduction in number of potential call edges or potential subject types. Often, in the discussion of these results, there is a implication that the observed reductions will naturally lead to substantial performance gains when used as a basis for devirtualization. For example, Lhoták and Hendren [2006, p. 11] infer: "*an analysis that proves that call sites are not polymorphic can be used to significantly improve run-time performance.*" Surprisingly though, there is little evidence to corroborate this widely held view. Of the numerous treatments for Java, only Sundaresan et al. [2000] make any mention of actual performance measurements where devirtualization is applied, and their results are quite limited and provide no methodological details. Using their VTA technique to support devirtualization, they report a 3% improvement for the soot program, and a 2% improvement for javac. However, their approach also performs static inlining after devirtualizing, so it is likely that most of the benefits come from this later step, not from simplifications of the dispatching mechanism. Similarly, Ishizaki et al. [2000] claim significant improvements from their "devirtualization" techniques. Although, again, they confound the removal of polymorphism mechanisms with the benefits of inlining, obtaining results very similar to those listed in figure 3.4.

A more substantive perspective is provided by Dufour et al. [2003] who examine several metrics for quantifying static and dynamic occurrences of polymorphic calls. Their study covers the SPECjvm98 programs, and several other benchmarks that, like the DaCapo programs, have a

more well-developed object-oriented code base. Their measurements indicate that `invokevirtual` and `invokeinterface` instructions constitute approximately 7-13% of the executed (i.e., dynamic) bytecode stream for the more object-oriented programs studied.[7] However, few of these call sites actually dispatch to more than one target. Among the SPECjvm98 programs, most see at least 99% of all executed calls resolve to one target. The only exceptions are javac and jack, in which 8% and 10% of calls, respectively, are dispatched from a site where more than one target is observed. The DaCapo programs are not considered, but among other large object-oriented programs (e.g., sablecc, soot, and volano) the diversity is also better, with 5-15% of calls occurring at sites with more than one target.

3.3.2 Polymorphism Query Profiles

To provide an alternative and more detailed analysis than that of Dufour et al., a series of profiles were gathered to identify how many polymorphism sites and subjects are actually visited in executions of the target benchmarks. These results reflect a single iteration of each benchmark, with only static inlining permitted—thus all polymorphism operations are left intact. Upon the execution of each operation the following information is recorded:

site – the static location of the operation;

subject – an unique identifier for the invoke receiver, or test subject instance;

method context – an unique identifier for each dynamic invocation (intraprocedural scope);

instance context – an unique identifier for the `this` reference of the method context; and

result – the call target, or test subject's precise run-time type resolved by the query.

From this data, two main reuse models are derived. The first considers how often the same result is computed at a given site with respect to either the global, method invocation, or instance context. This provides a sense of the maximum degree to which a site-oriented caching scheme could be used to eliminate redundant queries by suppressing the recomputation of a previously obtained result. In reality, at least some work must be done to determine if a result needs to be recomputed. The second reuse model provides a different perspective by tracking how many times a particular subject is the basis of a polymorphism query over the span of a certain context. This alternate view provides an indication of the maximum degree to which a subject-oriented redundancy elimination scheme could be used to reduce all queries that follow the first resolution of a subject's type.

In combination, these results provide an indication of the potential impact of the transformations proposed in chapters 4 and 5. Site-oriented redundancy that is more likely to compute the last result seen, rather than the most common result, is more amenable to subject-specific caching approaches, as opposed to static code reductions. At the same time though, the utility of such caching is clearly proportional to the number of times a subject is actually used over the span of a given context.

[7] Some of the non-object-oriented programs are well outside of this range. For example, compress only performs 16.5 invokes per thousand bytecodes executed.

Benchmark	*Sites*	*Calls*	*Predictor Success*			
			Best	*Last*	*Invocation*	*Instance*
antlr	5492	55800017	0.9969	0.9985	0.2701	0.9840
bloat	5925	111021027	0.9846	0.9919	0.0643	0.9086
chart	3464	190823388	1.0000	1.0000	0.0409	0.9760
fop	3551	10716712	0.9271	0.9488	0.1886	0.8737
hsqldb	2122	135578493	0.9999	1.0000	0.2096	0.9616
jython	21050	181166404	0.9552	0.9934	0.3787	0.9665
luindex	1429	99429452	0.9993	0.9997	0.2204	0.9929
lusearch	735	34513022	0.9999	0.9998	0.2446	0.9974
pmd	2634	247491526	0.9947	0.9968	0.3559	0.7402
xalan	3309	2439469	0.9886	0.9870	0.2093	0.9258
compress	80	3837	0.8723	0.9229	0.3500	0.9315
jess	454	28171554	0.9903	0.9860	0.7107	0.9859
db	115	56851440	0.9999	1.0000	0.8205	1.0000
javac	1502	42816218	0.9515	0.9520	0.3495	0.7826
mpegaudio	161	9849215	0.9733	0.9959	0.4017	0.9967
mtrt	645	85477072	0.9998	0.9998	0.0683	0.9593
jack	765	19310007	0.9988	0.9999	0.4831	0.9974
Mean, weighted by Calls:			0.9795	0.9871	0.4478	0.9459
L1	64	326460440	0.1668	0.9916	0.9916	0.9916
L3	64	380070587	0.2226	0.7720	0.3303	0.9967

TABLE 3.3: **Call Target Redundancy**

Site-Oriented Query Redundancy

Tables 3.3 and 3.4 present a summary of the overall site-oriented redundancy of dispatch queries (calls) and type tests. The values listed as *predictor success* represent the total cache-hit ratio of a collection of simulated site-specific result caches that use one of four strategies. The *best* predictor assumes that the most common result seen at a site is available. The *last* predictor assumes that the last result computed at a site is available, even across calls. When the *last* result exceeds the *best* result, this is an indication that polymorphic behavior is more strongly associated with repetitive uses than with a simple statistical distribution of subject variations. The *invocation* predictor checks against the last result, but only for a particular method invocation—it is always stale upon first use within a particular invocation. The *instance* predictor uses the current method's receiver (the `this` reference) as its context, thus predicting how uses of one object correlate with dispatches and tests applied to another object. The prediction rates of the standard benchmarks are summarized using the arithmetic mean weighted by number of events (calls or tests). According to John [2005], this measure is the most appropriate for selection rates.

The results for the L1 and L3 programs are notably quite different from the mean rates, indicating that their behavior does, in fact, consist of redundancy that is most pronounced over the contexts that they are designed to emphasize—invocation for L1, and instance for L3. The distinctly lower

Benchmark	*Sites*	*Tests*	*Predictor Success*			
			Best	*Last*	*Invocation*	*Instance*
antlr	546	2140240	0.8960	0.9567	0.0578	0.8890
bloat	1030	80757448	0.9682	0.9770	0.0215	0.9655
chart	306	29830758	1.0000	0.9999	0.0398	0.9808
fop	342	5163749	0.6386	0.9239	0.0045	0.9179
hsqldb	156	25849098	0.9999	0.9999	0.0255	0.9989
jython	568	40192183	0.9501	0.9881	0.0057	0.9417
luindex	145	14441388	0.9726	0.9731	0.1254	0.9346
lusearch	74	113706	0.9999	0.9993	0.0638	0.9453
pmd	363	21917397	0.5661	0.5165	0.1999	0.4253
xalan	339	231671	0.9161	0.9066	0.0611	0.8435
compress	14	56	1.0000	0.7500	0.7500	0.7500
jess	73	16766573	0.9801	0.9686	0.2120	0.9305
db	25	54662598	1.0000	0.9999	0.9202	0.9999
javac	316	8333615	0.7523	0.7497	0.1195	0.5748
mpegaudio	18	49067	0.9787	0.9655	0.0004	0.9993
mtrt	17	582438	0.8119	0.7709	0.0004	0.9829
jack	162	4856561	0.9722	0.9675	0.4873	0.9637
Mean, weighted by Tests:			0.9248	0.9251	0.5737	0.9122
L1	23	12060241	0.1667	0.9900	0.9900	0.9900
L3	N/A[8]					

TABLE 3.4: **Type Test Redundancy**

best predictor hit rates for L1 and L3 also reveal the degree to which these programs work against the assumptions of guarded inlining, which attempts to match the most common result(s) seen. The eclipse program is not included since it was not possible to obtain a successful profile run due to difficulties with its use of reflection. However, the remaining DACAPO programs clearly demonstrate a greater variety of call and test sites, as well as a greater number of run-time queries. Also notable is the fact that the global prediction rates for chart, hsqldb, lusearch, and db are all approximately equal to 1.0 despite performing a relatively large number of calls and tests. This is an indication that these programs show very little sign of polymorphic behavior. The results also reveal the distinct lack of virtual dispatch and type test operations in compress, thus suggesting it is of little interest for the work of this thesis.

Subject-Oriented Query Redundancy

Table 3.5 presents a summary of the overall subject-oriented redundancy—in other words, how often a particular object is reused in any form of polymorphism query. The *global* reuse rate represents the likelihood that an object will be re-queried throughout the entire benchmark iteration. The *invocation* rate is the likelihood that an object will be re-queried over the course of a particular

[8] The L3 program focuses on dispatching and performs no significant type testing. See the code in §C.2.

Benchmark	*Subjects*	*Uses*	*Object Reuse*		
			Global	*Invocation*	*Instance*
antlr	703693	57890133	0.9878	0.3693	0.9761
bloat	4335993	191235237	0.9773	0.0947	0.5664
chart	2849209	217847599	0.9869	0.3725	0.9306
fop	149293	13315904	0.9888	0.3139	0.9530
hsqldb	2699265	161361442	0.9833	0.3616	0.9616
jython	5595831	220389104	0.9746	0.4123	0.9556
luindex	1486426	113472202	0.9869	0.4099	0.9809
lusearch	28933	34626684	0.9992	0.2569	0.9984
pmd	10859453	268316997	0.9595	0.4276	0.8647
xalan	26603	2647250	0.9900	0.4179	0.9598
compress	174	3888	0.9552	0.3691	0.9336
jess	734948	42259036	0.9826	0.5665	0.9340
db	3028331	111511981	0.9728	0.8426	0.9337
javac	2188870	50536828	0.9567	0.3746	0.8520
mpegaudio	278	9898278	1.0000	0.5261	0.9999
mtrt	1590384	85949970	0.9815	0.6167	0.9212
jack	1512195	22840434	0.9338	0.5742	0.9201
Mean, weighted by Uses:			0.9778	0.4885	0.9302
L1	150	338520680	1.0000	0.9982	1.0000
L3	147	380070859	1.0000	0.3332	1.0000

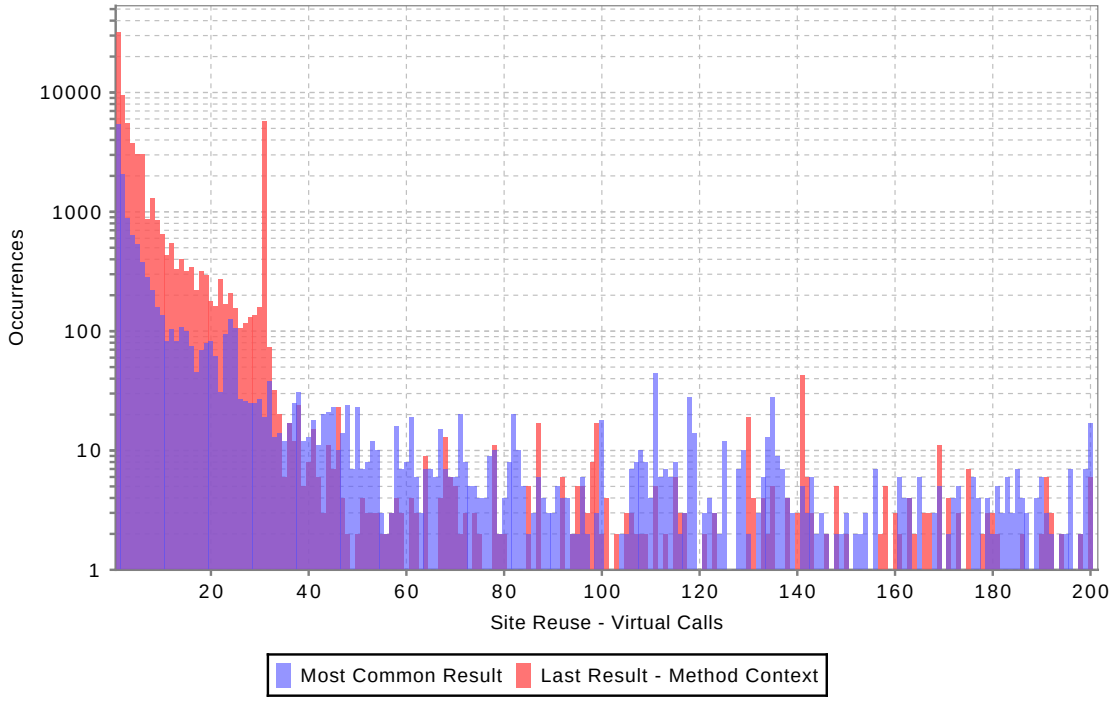
TABLE 3.5: **Multiple Object Queries**

method invocation (intraprocedural scope). Finally, the *instance* rate is the likelihood that an object will be re-queried within the scope of any instance method belonging to a particular object. This last measure also includes self-dispatches of the sort targeted by customization techniques. Again, the reuse rate is summarized over all the standard benchmarks using the arithmetic mean weighted by number of uses.

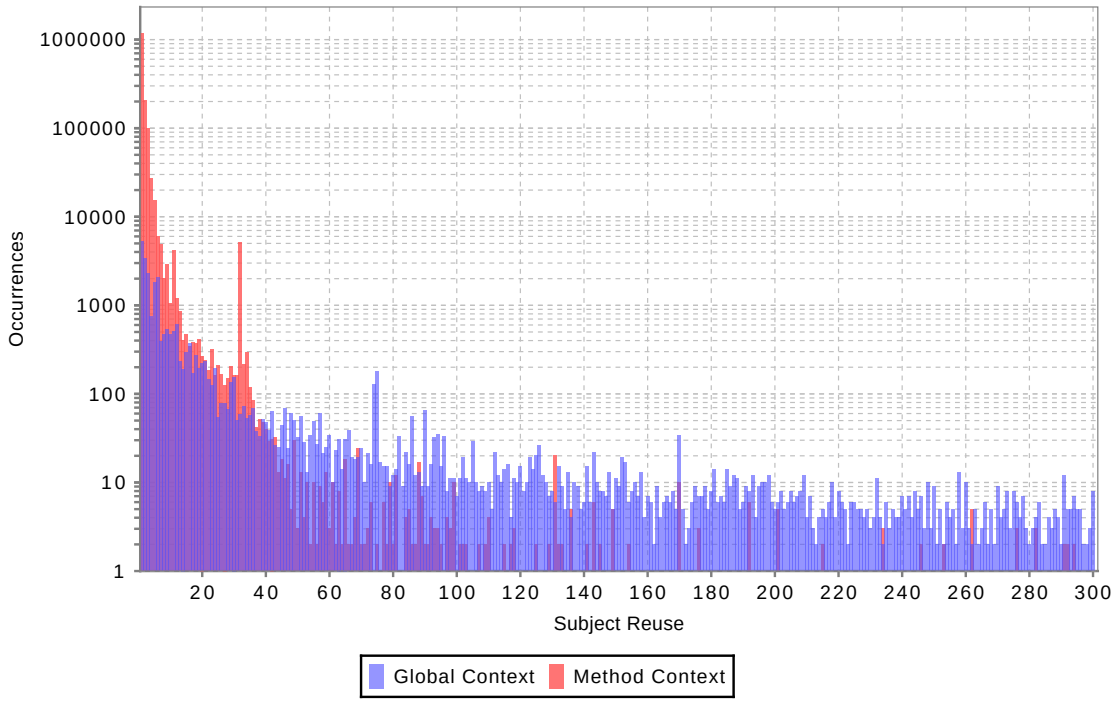
Similar to the site-oriented results, the L1 and L3 programs show reuse rates that are significantly higher for the respective contexts that they target. The number of distinct subjects and uses is also, again, mostly greater for the DaCapo programs. The well-known paucity of subjects allocated by compress and mpegaudio is also clearly apparent in these results, thus reinforcing earlier arguments that these programs are likely to be of little use in evaluating the proposed transformations.

Query Redundancy Graphs

While the overall prediction and reuse rates give some sense of the relative potential of query elimination techniques, these indications are still relatively opaque since they fold together results from thousands of sites and millions of usage sequences. To provide a more in-depth view of the reuse patterns, data from the caching models was processed further to show the distribution of site-oriented and subject-oriented reuses. A sample of these charts are provided in figures 3.5 through

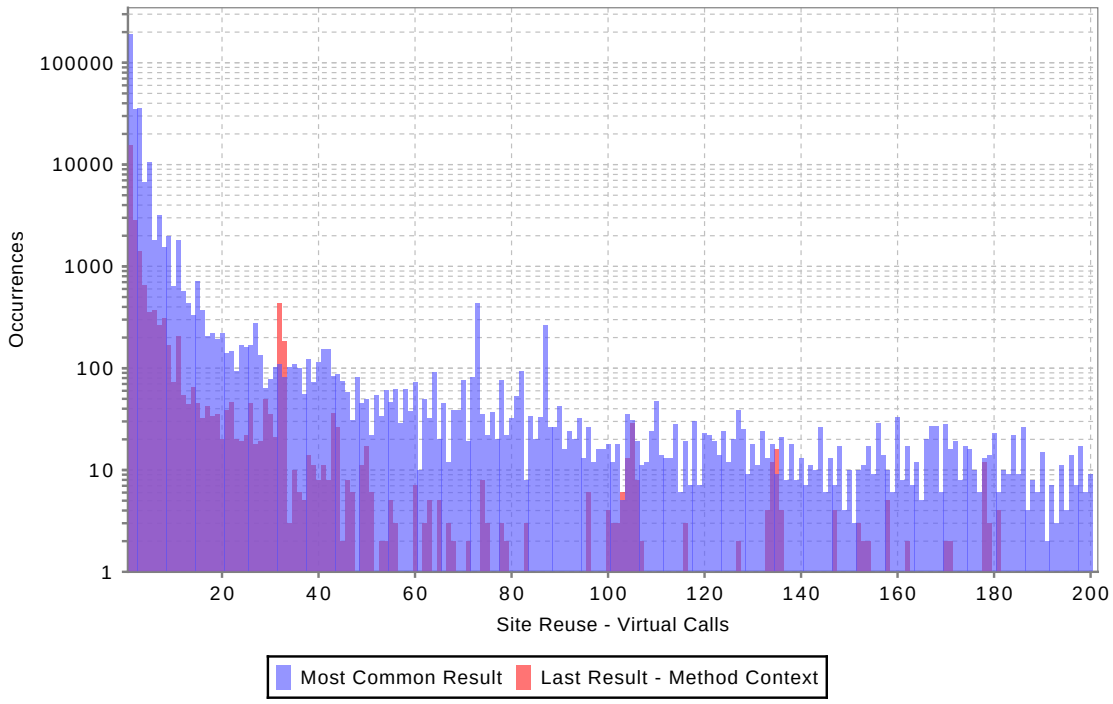


(a) Call Target Redundancy

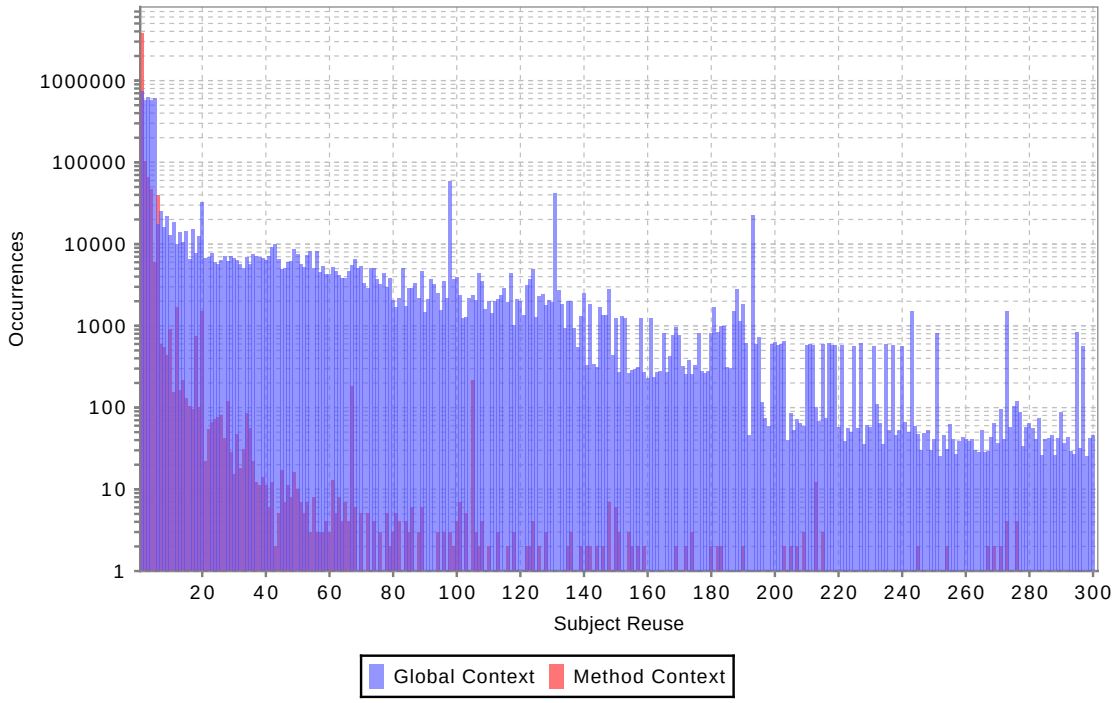


(b) Multiple Object Queries

FIGURE 3.5: **Intraprocedural Target and Subject Reuse for** xalan

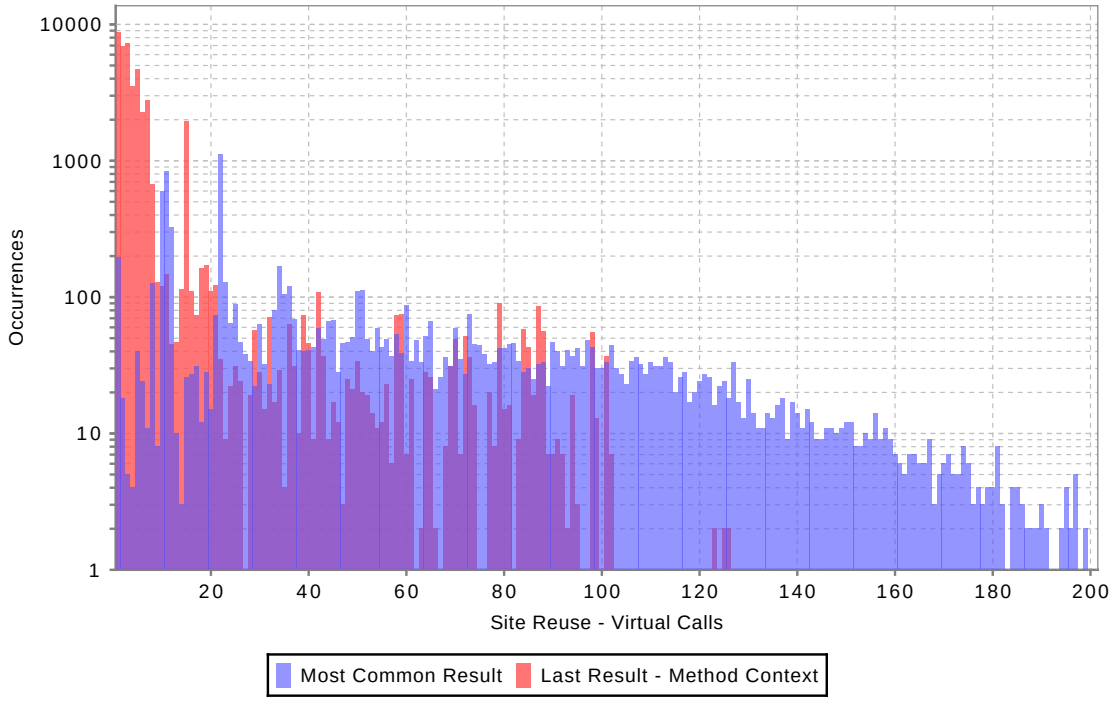


(a) Call Target Redundancy

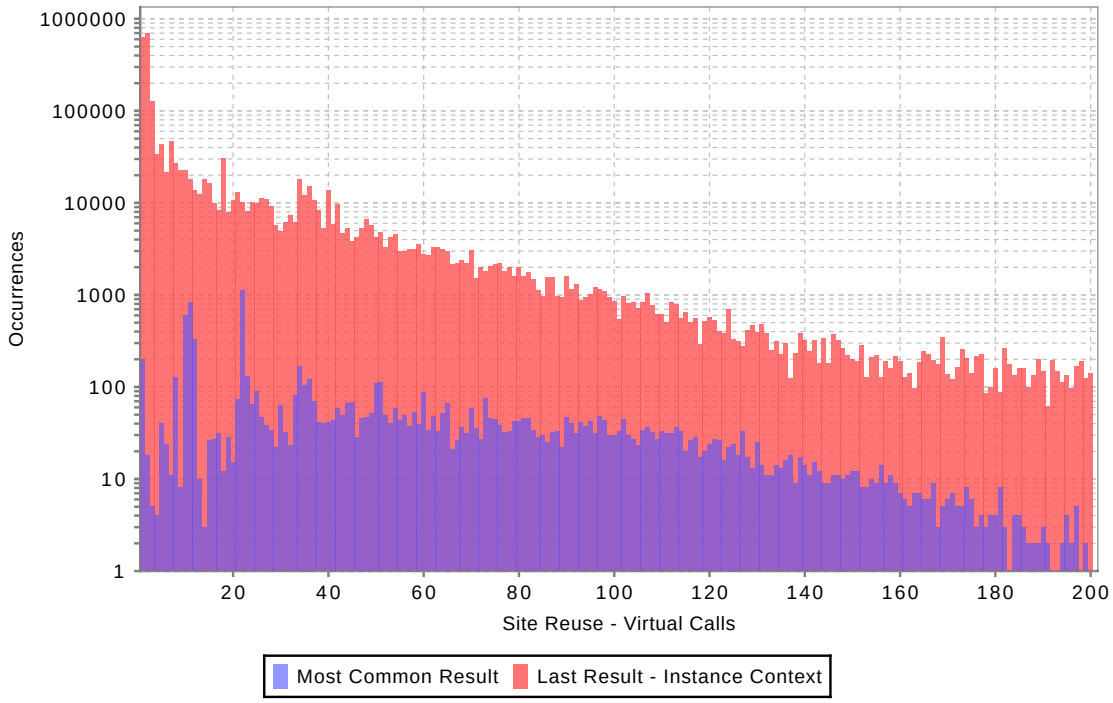


(b) Multiple Object Queries

FIGURE 3.6: **Intraprocedural Target and Subject Reuse for** bloat



(a) Intraprocedural Target Redundancy



(b) Instance-Lifetime Target Redundancy

FIGURE 3.7: **Comparative Target Reuse for** mtrt

3.7 to illustrate several key comparative examples. The complete set is given in appendix B.

These histograms contrast the observed run-lengths of site-cache or subject reuses. Each bar represents the number of occurrences of a run of a particular length of site-cache hits or subject reuses. Note that the counts (along the y-axis) are presented using a logarithmic scale. The blue distributions represent the *best* and *global* results, whereas the red distributions represent the occurrences of repetitions within the targeted contexts. In general, when the red results exceed the blue results, this is an indication that, for at least some queries, there is a significant amount of short-term redundancy that can be tied to the target context and presumably eliminated. The graphs do, however, omit the effects of one-time uses, as well as very high-use sites and subjects. For example, the subject reuse chart for L3 (see figure B.16) shows little apparent reuse, but this is because the typical period of reuse is much longer for the primary subjects. This is, of course, by design since the L3 program is intended to apply millions of calls to objects with a fixed coupling.

The first comparison, in figure 3.5, shows favorable intraprocedural reuse for xalan. There are many examples of sites exhibiting from 2 to 35 repetitions of the same query result over the course of a single (dynamic) invocation scope. Moreover, there are also many examples of the same subject being reused across these same scopes. Together, these distributions provide a more nuanced view of the behavior of xalan. Indeed, they suggest, in contrast to the low summary rates listed in the preceding tables, that much redundancy can be associated with uses of a particular subject, and thus should be amenable to the proposed techniques. On the other hand, similar charts for bloat (figure 3.6) show much less intraprocedural redundancy, both in terms of site reuse and subject reuse. The bloat results indicate much stronger reuse over the global context, suggesting that global techniques such as guarded inlining and inline caching are likely to achieve better results than context-specific optimizations. The third comparison, showing two presentations of site-reuse in mtrt (figure 3.7), highlights the differences that can occur in redundancy particular to each of the two targeted contexts: intraprocedural and instance-lifetime. In this case, mtrt shows significantly greater potential for capturing redundancy associated with the instance-lifetime scope.

3.3.3 Predictions

By studying both the reuse summaries and histograms for each benchmark, some predictions for the success of later efforts can be distilled. In general, those programs that show above average aggregate reuse rates with many sites and subjects are considered good candidates. However, the reuse distributions are also factored into the decision, promoting those that appear to have both favorable site and subject reuse (such as xalan – figure 3.5), while discounting those where global reuse appears more dominant (as in bloat – figure 3.6).

For the intraprocedural scope—the focus of chapter 4—the L1 program shows clear signs of being amenable to local redundant query elimination, primarily because of its distinctly favorable average reuse rates, but also because redundancy over the intraprocedural context is evident in its distribution charts (see figure B.16). In addition, antlr, pmd, xalan, jess, and jack all show some indication that they may benefit from a local reuse optimization. On the other hand though, each

of bloat, chart, fop, hsqldb, jython, luindex, lusearch, db, javac, and mtrt show at least one feature that is unfavorable. For some (e.g., bloat) it is a characteristically poor distribution combination, for others (e.g., hsqldb) it is an indication that little polymorphic behavior is present.

Overall, many of the benchmarks exhibit noticeably stronger potential over the instance-lifetime context. This is a natural consequence though of the fact that instances typically persist longer than intraprocedural contexts, allowing more redundancy to accumulate. However, there are two important factors that need to be considered when interpreting the instance-lifetime results. First, some of this redundancy occurs in `static` methods, and since such methods have no actual instance context, all of the reuses are associated with a default "no context" identifier. This may have the effect of overstating the total amount of instance-specific redundancy. A second issue to consider is that, as described in chapter 5, there is an additional cost incurred to install a specialized code environment on each subject that exhibits polymorphic coupling with another object. Thus the gains from saved query operations must outweigh the costs of these installations. Keeping these two factors in mind, there are indications that the instance-specific transformation will have a positive effect on antlr, jython, luindex, lusearch, xalan, db, and mtrt. To a lesser extent, there are also some reasons to suspect that chart, fop, hsqldb, jess, and jack will show some moderate improvement. Of course, as intended, the reuse rates for L3 indicate strong potential. There is, however, little evidence to suggest that specialization will help for bloat, pmd, or javac.

Due to their very low site and subject diversity, and their array vs. object emphasis, there is good reason to suspect that redundant polymorphism elimination will have little effect, positive or negative, on both compress and mpegaudio.

3.4 Related Work

A good overview of the Jikes RVM architecture is given by Alpern et al. [2000], although many refinements have been made to the system in the ten years since this report. Most notably, the project, which started out as an internal IBM research initiative, became an open-source platform in 2001 [cf. Alpern et al., 2005]. Since that time, hundreds of research projects have used Jikes RVM as their basis making it, arguably, the most widely studied JVM platform. Details regarding the adaptive optimization system in particular—its architecture, selection strategy, and compilation process—are covered by Arnold et al. [2004]. In addition to Jikes RVM, several other "industrial strength" Java Virtual Machines use a similar selective and adaptive optimization approach. An overview of these and other adaptive systems is given by Arnold et al. [2005].

Although performance evaluation is often oversimplified in publications relating to Jikes RVM and other JVMs, the analysis presented in §3.2 is not the first to observe problematic instabilities that can occur within executions, or over a group of executions. For example, Gu et al. [2006] studied the degree to which the behavior of different garbage collection strategies could distort the results of certain measurement strategies. Similarly, Mytkowicz et al. [2009] demonstrated how equivalently parameterized executions running on different processor architectures could lead to over- or under-valuation of certain improvements. Arguing the need for more rigorous measurement frameworks,

Georges et al. [2007] demonstrated that a number of performance measurement methodologies used in published studies of Jikes RVM were flawed and, in some cases, misleading. The insights of Georges et al. are adapted in §3.2 to derive a valid measurement methodology that is focused on the maximum potential of compilation alternatives.

A wide variety of metrics for characterizing the dynamic behavior of benchmarks have been proposed and investigated. The publishers of the DaCapo benchmark suite [Blackburn et al., 2006] are primarily concerned with memory allocation behavior and its implications for garbage collection. They provide a number of different analyses of allocation and object relationships on their web site (http://www.dacapobench.org). Extending this work, Jones and Ryder [2008] presented more detail on the distribution of object lifetimes. Focusing on low-level issues, Georges et al. [2004] examined how measures of instruction throughput and cache usage could be associated with different phases of a program's logic, rather than given as a single aggregate value that includes all application and virtual machine behavior. In related work, Eeckhout et al. [2003] investigated how the choice of JVM architecture affects the low-level caching and throughput behaviors of various Java benchmark applications. The one study that is most relevant to the work of this thesis though is that of Dufour et al. [2003], who provide a series of measures which outline the typical demographics of polymorphic dispatches seen in Java benchmarks. In their approach, Dufour et al. use a number of "bin" metrics that partition events into several classes. For example, dispatch sites that exhibit one, two, or more than two distinct targets. The profile metrics given in §3.3.2 represent a mammoth extension of this concept in that they effectively associate a bin counter with every combination of site and context, or subject and context. The accumulated counts are distilled back down to a single summary measure in tables 3.3 through 3.5, but can also be used to give a more comprehensive perspective of the distribution of behaviors, as seen in figures 3.5 through 3.7, and in appendix B.

Although redundancy elimination is a very widely studied problem, none of the seminal works include any significant attempts to forecast or contextualize the impact of the transformations they propose, as is done in this chapter. Occasionally a more in depth analysis is provided in retrospective studies. For example, Driesen and Hölzle [1996] and Zendra and Driesen [2002] evaluate the costs inherent in known strategies for implementing and optimizing virtual dispatch mechanisms. The work that is perhaps most similar, at least in intent, to the analysis presented here is that of Ball and Larus [1993] who examine the behavior of branch instructions in various contexts with the aim of using this information to reorganize code to improve branch prediction.

3.5 Summary

Jikes RVM is a suitable platform for developing and evaluating the optimizations put forward in the next two chapters for several reasons. First, working within Jikes RVM's just-in-time compilation framework allows access to low-level implementation details of Java polymorphism operations that are simply not exposed in bytecode representations. Also, even though Jikes RVM is an experimental platform, its architecture has been significantly refined to the point that its standard configuration is acknowledged to be competitive with other state of the art JVMs. Thus it provides a realistic basis

for presenting performance comparisons, and achieving performance improvements over this standard configuration represents a substantive research advancement. Finally, Jikes RVM is a widely studied system, especially among the Java optimization research community. Thus presenting optimizations in terms of its internal structures and processes provides an expression that is easily comprehended, and hence extensible, by other researchers.

However, as demonstrated, Jikes RVM is a complicated system, and the progress of its adaptive optimization system should not be misunderstood as producing steady performance improvement which ultimately converges to a stable throughput level. Factors such as garbage collection, caching disruptions, and the non-deterministic identification and update of hot code implementations collude to produce throughput that is, in fact, highly unstable over individual executions. Despite this erratic behavior though, it is possible to observe the progressive improvement of code by aggregating the best timing results seen over multiple benchmark executions. Based on this insight, a new, statistically rigorous methodology for measuring performance is derived which filters out the noise of system disruptions to reveal the maximal code potential of various compilation alternatives. This methodology is used as the primary mode of comparison in subsequent evaluations.

To present results that are comparable with other published studies, improvements are measured over the standard SPECjvm98 and DaCapo suites of Java benchmark programs. Some of the programs in the former suite are, however, recognized to have questionable levels of diversity in their use of objects and object-oriented idioms—i.e., their use of virtual dispatches and type tests. In order to give a clearer indication of which programs should be considered more relevant in subsequent evaluations, several analyses of their code and uses of polymorphism are presented. These measurements suggest that, in general, the DaCapo programs present a larger, more complex code base to the optimizing compiler, and contain a wider variety of polymorphism sites and subjects. Thus, in later sections, results from these programs are considered more demonstrative of the broad effects of redundant polymorphism elimination. In particular, results for the eclipse and xalan benchmarks should be viewed as highly relevant since these programs, in addition to being large and complex, are also based on widely used and well-developed open-source code frameworks.

3.5.1 Contributions

In seeking a stable and effective method for isolating the effects of changes in compilation strategy, the analyses presented in §3.1.2 and §3.2 provide the following, previously undocumented observations:

- In addition to inlining along active execution paths, Jikes RVM also performs a significant amount of spurious inlining based on shallow statistical skews in the estimated call graph structure. While this inlining has little effect on overall throughput, since it occurs on inactive pathways, it does have a significant impact on just-in-time compilation workload, and may also be an indirect cause of performance instabilities due to its effect on the scheduling of adaptive optimization updates.

- For many standard benchmarks, the evolution in iteration scores over the course of an execution on Jikes RVM is highly variable and distinctly chaotic—that is, the behavior is inconsistent. Thus averaged timing results taken from a particular iteration (a commonly used methodology), should not be treated as representative, nor should they be accepted as statistically rigorous. Furthermore, although the "adaptive sampling window" approach proposed by Georges et al. [2007] does offer a statistically acceptable measurement technique, it does always not succeed at converging on a result for a low, fixed coefficient of variance.

- Even though the progress of Jikes RVM's adaptive optimization system is largely obscured by the variance in iteration scores over any given execution, it is possible to reveal the rate of this progress by extracting the best results across a group of similarly configured executions. Moreover, while the rate differs from program to program, all of the SPECjvm98 and DaCapo programs demonstrate a clear statistical convergence to a best (minimal) timing score.

These insights are combined in §3.2.1 to present a new performance measurement methodology which is specifically tailored to focus on the effects of compilation while discounting other factors such as garbage collection and caching disruptions. The key premise of this method is that it is not pegged to a particular iteration or window of iterations: the best result is extracted whenever it occurs. The measurements given in table 3.2 and §A.1 confirm that these *best* results are not extracted prematurely. Furthermore, while developed for evaluating modifications to Jikes RVM, this methodology is equally valid and useful for analyzing the compilation performance of other adaptive run-time environments, assuming that the measured benchmarks produce regular timing results.

In an initial application of this more robust measurement scheme, the baseline comparison of inlining strategies given in §3.2.2, while not demonstrating any new performance results, do offer an important and consistent corroboration of the improvements reported by Ishizaki et al. [2000]. These results also affirm utility of the class test and method test guarded inlining strategies proposed by Detlefs and Agesen [1999]. With respect to both previous works, the results shown in figure 3.4 overlap with earlier studies of the SPECjvm98 programs, and provide new, comparative measurements for the DaCapo programs.

The major quantitative results offered in this chapter are the detailed characterizations of uses polymorphism in the SPECjvm98 and DaCapo suites, as described in §3.3.2. These analyses represent a substantial extension of the measurements of Dufour et al., but also largely confirm their assessment that the SPECjvm98 program contain mostly degenerate uses of polymorphism. Also of note is the fact that, from a site-oriented perspective, the last result computed is often more reusable than the most common result. Thus suggesting that a global, yet adaptive technique such as polymorphic inline caching (see §2.4.2) could be more beneficial as an optimization than static guarded inlining.

3.5.2 Directions

Performance Measurement

In deciding when to promote a method to the next optimization level, Jikes RVM's adaptive optimization system uses a heuristic function based on what is termed its "compiler DNA." This is essentially a table of (hardcoded) weights that rate the expected speedup from techniques enabled at each level. Currently, the values for this table are derived from training runs that measure the average performance over a single execution of 15 iterations of the SPECJVM98 programs. Moreover, these results are gathered by proceeding directly to the targeted optimization level without engaging the adaptive optimization system. This ad hoc approach bears some obvious weaknesses. Applying the more rigorous, compilation-focused measurement scheme outlined in §3.2.1, over a variety of processor platforms, would seem to be an obvious, and more appropriate method for deriving these heuristic weightings.

Polymorphism Analysis

As indicated in §3.3.3, the polymorphism query profile analyses do suffer from some imprecision which may distort the apparent redundancy associated with each of the local contexts—intraprocedural and instance-lifetime. In particular, for the both contextual models, redundancy is measured over all occurrences of polymorphism operations. By performing a local flow analysis on each profiled method, these results could be refined to focus only on queries of local subjects in the case of intraprocedural redundancy, and subjects loaded as fields of the current `this` reference in the case of instance-lifetime redundancy. A further refinement of the instance-lifetime results could also be achieved by either filtering out results from `static` method contexts, or perhaps separately modelling these results as coupled with the class in which they appear.

Chapter 4
Intraprocedural Redundant Polymorphism Elimination

The *intraprocedural* context—the scope of individual procedures, functions, or methods—is the most widely studied in program optimization. In particular, approaches to redundancy elimination based on analyses over the control flow of subroutines have a long history. The work of Cocke [1970] on the elimination of common subexpressions dates back 40 years, while the seminal work of Morel and Renvoise [1979] on the elimination of partial redundancies is now more than 30 years old. In addition, many important extensions and refinements of these techniques have been developed since their original publication. Surprisingly though, despite the fact that late binding polymorphism is nearly as old as the work of Morel and Renvoise, its implementation has largely been ignored as a candidate for redundancy elimination techniques. Site-oriented approaches to suppressing redundant polymorphism, such as devirtualization and guarded inlining, are well studied, but to date only Nguyen and Xue [2004] have considered how the results of polymorphism queries can be reused across the intraprocedural context.

This chapter presents an approach to eliminating intraprocedural polymorphism redundancies that is significantly more general than the approach of Nguyen and Xue. It targets all of the single-subject polymorphism operations in the Jikes RVM intermediate representation, including the four explicit polymorphism operations found in Java bytecode (see §2.3), as well as implied type tests, and queries used in the implementation of guarded inlining.[1] Furthermore, a study of intraprocedural applications of polymorphism in design patterns is used to guide the integration of modern enhancements to redundancy elimination such as *selective* and *speculative* optimization. The latter, while beneficial, also introduces complications when dealing with queries applied to potentially `null` references. To deal with this issue and ensure that exception semantics are preserved, a novel approach to guarding speculative computations is developed and included. An evaluation of several facets of the approach is provided by measuring performance gains over runs of the benchmark programs described chapter 3. Several statistically significant improvements are observed which align, at least partly, with predictions from §3.3.3. Notable results include a 13.6% speed up of the `L1` program when guarded inlining is disabled (12.3% when enabled), as well as improvement across the DaCapo suite even when combined with guarded inlining.

[1] Array store operations (`aastore`) entail a two-operand dynamic type test which is more elaborate, and not considered here.

4.1 Problem

As outlined in §2.3, the mechanics of polymorphism operations and their associated costs are often masked by the simple source-level syntax for virtual dispatches and type tests. Behind such syntax, a virtual dispatch operation involves a minimum of two dependent load operations to identify the target method address. Other polymorphism operations, however, can exact a much higher cost.

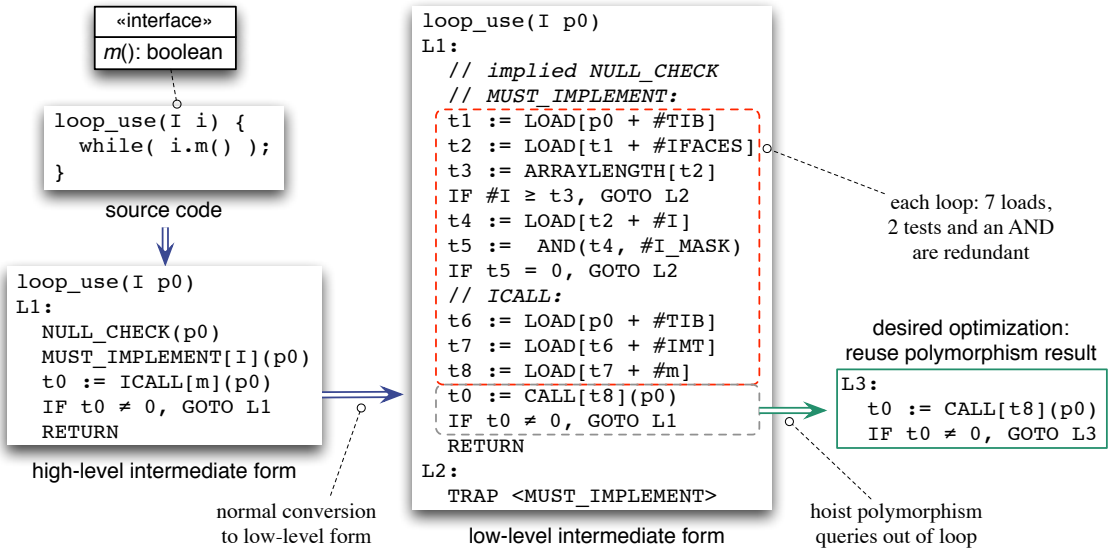


FIGURE 4.1: **Interface Dispatch Redundancy**

Consider the implementation (in Jikes RVM) that underlies a simple interface dispatch, as depicted in figure 4.1. The first thing to note is that the invoke requires two preceding tests to ensure valid execution: a `NULL_CHECK` which throws an exception if the subject (receiver) is `null`, and a `MUST_IMPLEMENT` check which throws a different exception if the subject fails to implement the declared interface type. This second check is necessary to prevent inconsistencies between the uses and definition of the subject which may arise from separate compilation and loading. On the Intel x86 platform, the `NULL_CHECK` can be folded into an implied trap on the first load applied to the subject. However, the type check and interface dispatch together expand to seven load operations (including the array length access) and several logical tests.[2] Moreover, these operations are executed on every iteration of the loop since they cannot be separated from the first call. Given an understanding of the semantics of this code, the optimal transformation of the low-level form seems trivially obvious: simply hoist all of the query operations out of the loop and only repeat the indirect call to the computed (and verified) target. Unfortunately, such a transformation cannot be obtained using standard optimization techniques which must make conservative assumptions about the results of load operations. Specifically, general load elimination strategies must perform

[2] The operations are relative to the TIB structure depicted in figure 2.1.

sophisticated analyses to prove that the contents of the targeted memory does not change. Indeed, such analyses would deem the suggested transformation invalid since the virtual table and IMT entries do change (concurrently) in Jikes RVM as new, optimized method implementations become available from the adaptive optimization system. Fortunately this behavior is not actually an impediment to the optimization of polymorphism queries since the effective semantics of type information does, in fact, remain invariant. In other words, updates to the TIB do not affect the outcome of execution—method implementations may change, but their behavior does not.

The following sections continue to focus on examples involving interface dispatching for several reasons. First, interface dispatches involve more overhead that normal virtual dispatches, and thus stand to offer more benefit if optimized, especially when they require additional `MUST_IMPLEMENT` checks. Second, as previously noted in §2.4.1, interface dispatches typically require more heavy-handed analyses to be eliminated via devirtualization. Third, and most importantly, interfaces are the preferable means (in Java) for creating the opaque forms of subtyping polymorphism that are a key element of design patterns. Thus lessons from design patterns are likely to be particularly applicable to interface dispatching and its consequent costs.

4.1.1 Lessons from the Builder Pattern

The BUILDER design pattern represents a state machine-like entity whose purpose is to assemble an encapsulated aggregate based on a sequence of externally provided directives. A simple usage example is given in figure 4.2.

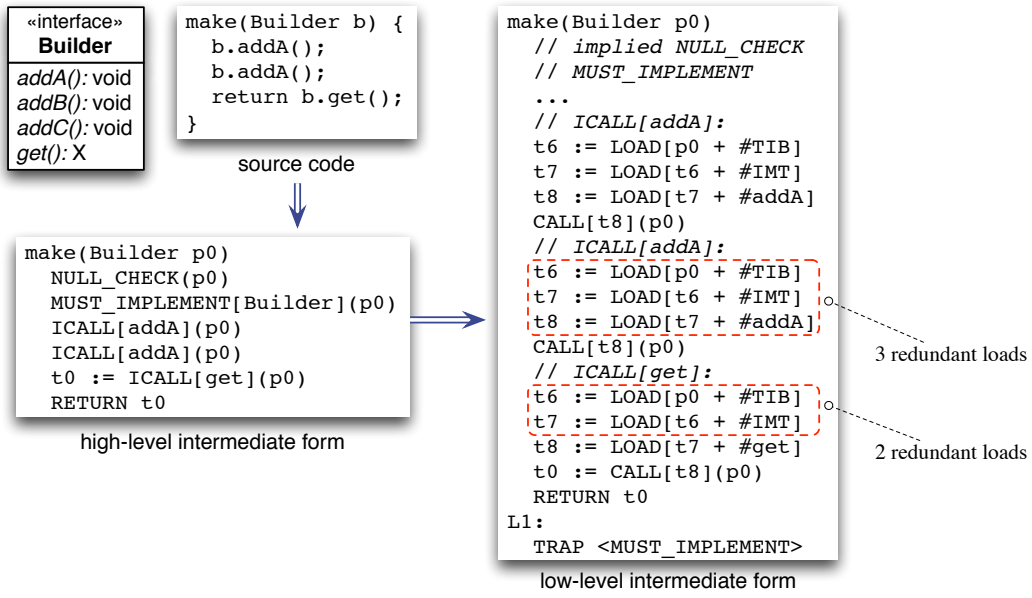


FIGURE 4.2: **Builder Redundancy**

To permit a variety of encapsulated part and result types, the builder reference is often passed as an interface type, allowing substitution of any conforming implementation. This necessitates a `MUST_IMPLEMENT` check since the precise type of the builder is obscured. Furthermore at least two, and probably more interface dispatches are likely to be applied to assemble and render the result. As illustrated in figure 4.2, these dispatches, at minimum, redundantly load the TIB and IMT reference several times, and may also redundantly load entries from the IMT. The given example does not include any looping, but repetitive applications are not uncommon. The key feature that distinguishes the scenario from other interfaces uses such as ITERATOR (seen next) is that it involves at least one, definitely executed polymorphism query with results that can definitely be reused.

While BUILDER's specific purpose is the creation of new objects, its usage exhibits similarities to many other patterns and commonly used abstractions. For example, the STRATEGY, STATE, and TEMPLATE METHOD patterns represent approaches to defining objects that present one or more actions which are intended to be composed by an external "director" context. The common object-oriented approach to representing communication streams also entails similar usage patterns; examples include file input/output abstractions, network channels, tokenizers, encoders, compressors, and encryption schemes. On the other hand, a MEDIATOR explicitly represents a director of such objects. This chapter also considers any method that applies a number of polymorphic operations to an abstract (i.e., opaque) entity. The `Director`, `Branch02`, and `CastGuard` components of the `L1` program (see §C.1) are used to stress usage scenarios similar to that of BUILDER.

4.1.2 Lessons from the Iterator Pattern

ITERATOR is one of the simplest of the commonly recognized design patterns, yet its usage still contains some important lessons for the elimination of redundant polymorphism. In particular, ITERATOR is the only design pattern that explicitly focuses on repetitive use as part of its intended purpose. Moreover, unlike the BUILDER scenario, the redundancy that occurs in uses of ITERATOR is dynamic rather than static. In other words, reusable query results come from a previous visit to the same site rather than being carried forward from a statically similar expression. This means that an understanding of the control flow must be incorporated into the optimization to distinguish uses that definitely repeat from those that may repeat (or not execute at all) since, in the case of the latter, the results may not be available to reuse. Indeed, the interaction depicted in figure 1.2(b) distorts the usual code expression of ITERATOR in order to reveal the redundant queries.

Figure 4.3 shows the typical expression of ITERATOR. A similar structure results from the translation of Java's "for-each" syntax over collections. The second `MUST_IMPLEMENT` check is included in the intermediate form since the uses span more than one basic block—by default, Jikes RVM makes no effort to detect such inter-block redundancies. Like the example in figure 4.1, the queries are trapped within the loop. Those related to the `hasNext` call can be hoisted, given an understanding of the effective semantics, since the call executes at least once. However, the query to obtain the `next` method is more problematic. The call does not necessarily execute, and thus classic, conservative approaches to redundancy elimination are prevented from exposing the redundancy by moving or

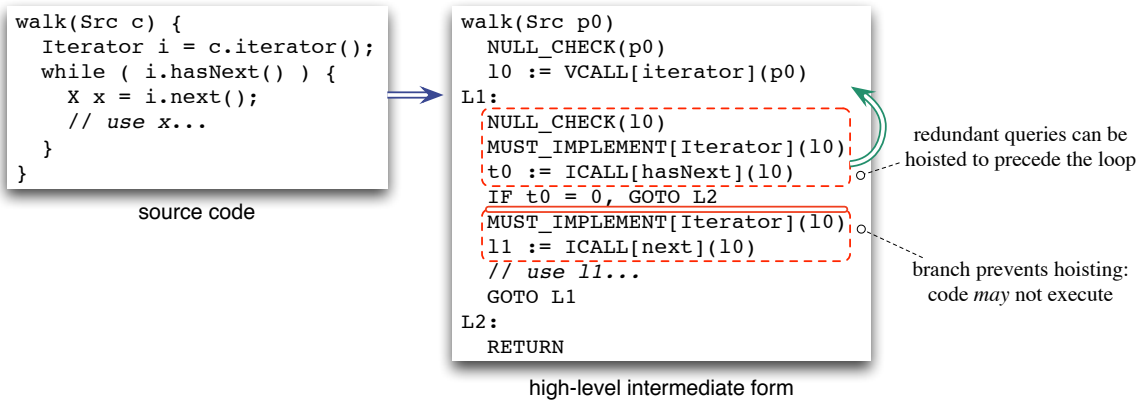


FIGURE 4.3: **Iterator Redundancy**

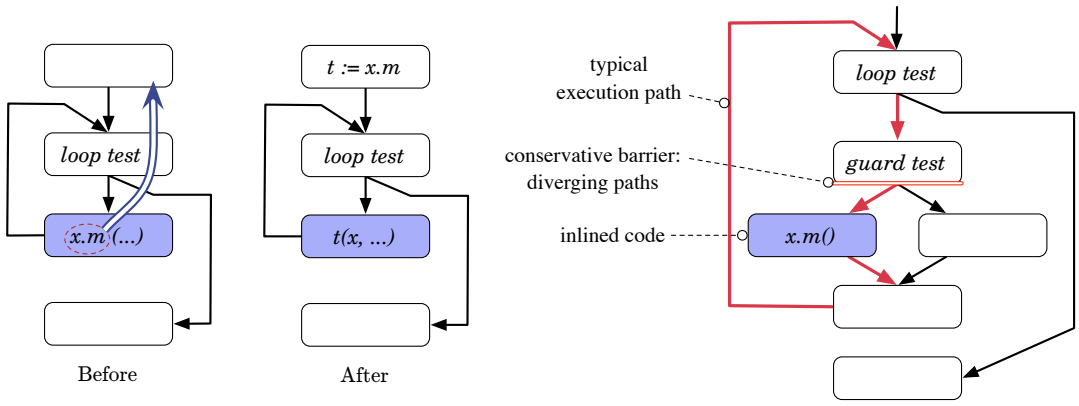
duplicating the query computation. In order to optimize the code so that the `next` target query is not repeated for executions that iterate more than once, a *speculative* transformation is needed. In short, the query must be moved to a location (before the loop) where it always executes, making the result available on the first and subsequent iterations. Such a transformation is called speculative because it relies on a presumption that the repetitive case is more common, and that the savings from improving that case outweigh the cost of the extraneous computation when no repetition occurs. However, while speculation often results in faster execution, it also has the potential to introduce unintended changes in program semantics. For example, speculatively hoisting division computations is typically regarded as *unsafe* [Kennedy, 1972] since it may lead to a divide-by-zero trap where no such operation would have occurred in the original program. In the case of polymorphism, speculatively moving the queries also entails moving the run-time checks that precede them. To maintain correct execution behavior, a valid optimization needs to take further steps to ensure the effects of these checks remain the same as in the original program since the program's logic may actually rely on the exceptions being triggered at their original location.

As indicated in figure 4.3, a speculative transform typically involves moving code across a conditional branch instruction. Putting aside the issue of safety for the moment, the decision of whether to speculate is often simple when the branch is identified as a loop control. The choice is more complicated though for other branches, specifically those that reside inside loops. To stress the relevant cases, the `Iterator` and `Branch01` components of the `L1` program (§C.1) include redundant uses of polymorphism that require speculation across loop-control branches, as seen in common expressions of ITERATOR, and skewed branches within loops which are a common artifact of guarded inlining. Both are important since they represent the simple ITERATOR idiom as well as examples that arise from more elaborate uses of repetitive polymorphism.

4.2 Proposed Solution

The query component of a polymorphism operation can be seen as a composite expression similar to other expressions over arithmetic, logic, and load operations. By combining this view with an understanding that polymorphism data is invariant for any given subject, this suggests that standard redundancy elimination schemes can be applied to eliminate redundant query computations. In the context of Jikes RVM, this means applying flow analyses to find opportunities for reusing results previously loaded from a subject's TIB and other type structures.

Specifically, the BUILDER example suggests a transformation based on a forward (availability) flow analysis, such as *global common subexpression elimination* (GCSE). The ITERATOR example, on the other hand, also motivates the use of a transform that includes backward (anticipatability) flow information, such as *loop-invariant code motion* (LICM). However, even in combination, these approaches still fail to capture some important cases.



(a) Speculative Transform

(b) Blocked by Inline Guard

FIGURE 4.4: **Limitations of Loop-Invariant Code Motion**

The problem is illustrated in figure 4.4(b). Guarded inlining typically introduces a conditional branch in front of the common execution path. When this occurs inside a loop, it prevents LICM from hoisting loop-invariant expressions in the inlined region since the strategy only speculates across loop-control branches. Unrolling the loop and applying GCSE is of no use in this situation since the target expression remains on a conditional path and is thus never definitely available. Traditional approaches to partial redundancy elimination are of no help either, since they too are blocked by the diverging path. Choosing to disable guarded inlining is not an acceptable solution, since it has proven effective in eliminating site-oriented polymorphism redundancy. Finally, attempting to speculate across all branches is also not a viable alternative since it permits some pathological cases which are distinctly sub-optimal, such as hoisting expressions into rather than out of loops.

To address the limitations of GCSE, LICM, and other forms of conservative partial redundancy

elimination, without over-speculating, several approaches have been developed based on probabilistic flow analyses [Bodík et al., 2004; Scholz et al., 2003] and cost/benefit analyses [Horspool and Ho, 1997; Gupta et al., 1998; Cai and Xue, 2003; Lin et al., 2004; Scholz et al., 2004]. However, while comprehensive, these methods remain too computationally intensive for a just-in-time framework such as Jikes RVM's compilation system.

An alternative, which achieves general speculation at a reasonable cost, is to use a threshold scheme such as *isothermal speculation* [Horspool et al., 2006]. In this approach, computations that reside in basic blocks with an observed frequency above a certain threshold are moved to less frequent zones, even if this means inducing speculation by crossing conditional branches. The technique has been shown to effectively hoist expressions along highly active control paths, and in particular over guards and out of loops [Pereira, 2007].

In its original formulation, the isothermal speculative PRE (ISPRE) approach used custom flow analyses, and off-line profile data to calculate speculation thresholds. However, the threshold concept can also be realized through an adaptation of the well-known *lazy code motion* [Knoop et al., 2004] approach to partial redundancy elimination. Additionally, the thresholds can be inferred from on-line branching statistics gathered by Jikes RVM. In constructing a system for eliminating polymorphism redundancy, this speculative threshold variation on lazy code motion [Brown and Horspool, 2009b] is used to effectively capture the considered cases while keeping a bound on the cost of compilation.

4.2.1 Code Analysis and Transformation

To implement a redundancy elimination transform over polymorphism queries (dubbed RPE), the polymorphism operators found in Jikes RVM's high-level intermediate form (HIR) are interpreted as composite expressions over the operations they expand to in the low-level form (LIR). This exposes subexpressions that are shared across multiple queries of the same subject. For example, interface dispatches on the same receiver load from the same IMT structure, whereas all polymorphism operations over the same subject consult its TIB reference. The following representations are used to model the various polymorphism operations employed by Jikes RVM:

```
VIRTUAL_CALL[m](o)      ⇒  CALL[METHOD(#m, TIB(o))](o)
INTERFACE_CALL[m](o)    ⇒  CALL[METHOD(#m, IMT(TIB(o)))](o)
INSTANCEOF[T](o)        ⇒  TYPE_TEST[T](TIB(o))
CAST[T](o)              ⇒  CAST(TYPE_TEST[T](TIB(o)))
MUST_IMPLEMENT[T](o)    ⇒  CHECK(TYPE_TEST[T](TIB(o)))
IG_METHOD_TEST[m](o)    ⇒  EQ_TEST(m, METHOD(#m, TIB(o)))
IG_CLASS_TEST[C](o)     ⇒  EQ_TEST(C.TIB, TIB(o))
```

FIGURE 4.5: **Polymorphism Query Expressions**

The `METHOD`, `IMT`, and `TIB` expressions represent simple offset load operations in the LIR. The `TYPE_TEST` expressions are more elaborate and have a different expansion based on whether the type to match is a class type, array type, or interface type. The `IG_METHOD_TEST` and `IG_CLASS_TEST` operations do not arise directly from the input bytecode but are introduced when the compiler

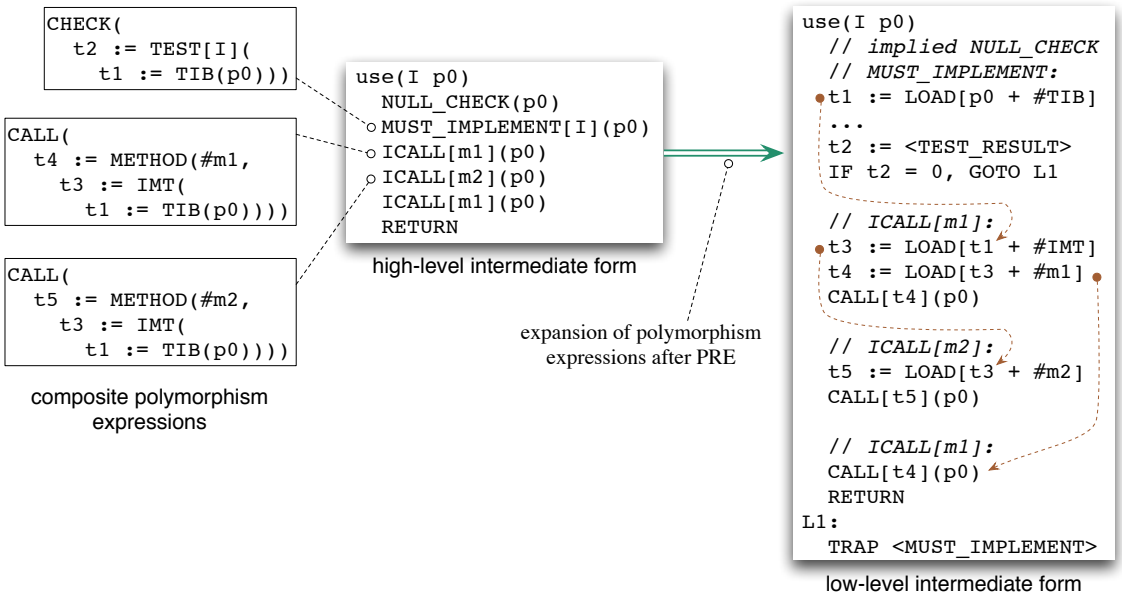


FIGURE 4.6: **Intraprocedural Redundant Polymorphism Elimination**

performs a guarded inlining transform. These are also candidates for redundancy elimination since they do entail a query and may be repeated within a loop (as in figure 4.4(b)). Indeed, Nguyen and Xue specifically target these operations in their approach. More importantly though, eliminating inter-guard redundancy is a key example of how the subject-oriented focus of this thesis serves to complement rather than supplant standard site-oriented techniques.

To identify which results can be shared, and which should be moved to expose sharing opportunities, a local variable is associated with each lexically unique subexpression. An analysis framework known as Selective, Speculative PRE (SSPRE) is then used to determine which query results are available and which are anticipatable along a given execution path. In the simple case, as shown in figure 4.6, available results are reused in the expansion of later queries. When the operations reside within a loop, a speculative threshold approach is used to judge when it is worth hoisting anticipatable queries to an earlier, less frequently executed location. If several hoisted expressions are to be inserted at the same location, a simple ordering scheme is used to ensure that, for example, a subject's `TIB` reference is loaded before computing one of its `METHOD` targets.

To reduce the cost of compilation, the SSPRE technique also uses a second, selective threshold to determine which code is worth optimizing. In developing this analysis and evaluating various threshold calculations, a previous study [Brown and Horspool, 2009b] showed that only roughly a quarter of the code fed to Jikes RVM's optimizing compiler needs to be considered and analyzed to achieve the full benefit of the technique. The results of this work are used to guide the choice of both the speculative and selective thresholds employed in the analysis of polymorphism queries.

4.2.2 Issues

Given an understanding of the basic strategy used to move and transform polymorphism queries, several issues present themselves which need to be addressed, or at least considered when interpreting the performance results.

Lexical vs. Semantic Query Equivalence

Using a lexical representation to perform redundancy elimination over expressions is known to have certain limitations [Alpern et al., 1988]. In the general case, when considering arbitrary arithmetic and logic operations, some composite expressions may have different lexical signatures but still represent equivalent computations. This can arise from the symmetry of certain binary operators ($a + b$ is equivalent to $b + a$) and from aliasing of local values ($a + b$ is equivalent to $a + c$ if $b = c$). Approaches have been developed to identify these sorts of semantic equivalences and eliminate partial redundancies among them [e.g., VanDrunen and Hosking, 2004b], however such techniques are, in fact, unnecessary for the set of polymorphism expressions considered.

Note that all the query expressions shown in figure 4.5 involve a single run-time variable; the values of the method and type identifiers are all known to the compiler. Thus each expression form has a unique construction and no symmetric alternatives are possible. (The `EQ_TEST` expressions are always constructed as shown.) Equivalences due to local aliasing are also precluded by the application of a copy propagation transform immediately preceding the query optimization phase. In combination, these two features ensure that the lexical representation is sufficient to capture all of the locally identifiable redundancies.

Multiple Analysis and Transformation Passes

Another well-known issue that can occur when attempting to eliminate redundancies is the localized obstruction of code motion due to subexpression dependencies [cf. Knoop et al., 1995]. A simple example is illustrated in figure 4.7. As a composite, the x result is clearly loop-invariant. However, treating each operation independently seems to suggest otherwise since the second operation has an input which is recomputed on every repetition. A recognized solution for this problem is to repeat the analysis and transformation several times. In the example, two passes are needed to hoist the first invariant computation, thus removing the problematic dependency, and then hoist the second in a subsequent application.

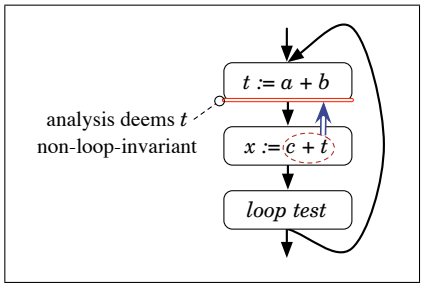


FIGURE 4.7: **Blocked Code Motion**

Fortunately, these sorts of dependencies do not occur in the given formulation of polymorphism query elimination. In the HIR form none of the queries have any interdependencies. And while the query expressions are represented as composites, the analysis is performed over the synthetic representations (seen at left in figure 4.6) not their expanded low-level form. The only input variables

that impose constraints on relocating the queries are the subject references. Thus, unlike other redundancy elimination strategies, multiple optimization passes are not required to achieve the full benefit of the approach.

Adaptive Optimization Updates

As mentioned previously, the data loaded from TIB and IMT structures does, in fact, change over time as these tables are updated to point to improved method implementations generated by Jikes RVM's adaptive optimization system. Moreover, these updates are performed concurrently with normal program execution and thus, may occur between repeated visits to the same dispatch site. A subtle consequence of this behavior, in combination with redundant query elimination, is that it is possible for the dispatch target used at a particular site to be out of date and refer to a less efficient implementation. In practice, however, such an event is not likely to be of any significant consequence. An outdated reference only persists until the end of the current dispatch context. The next time the context is invoked and the query performed anew, the improved target value is used instead. This behavior renders the issue moot with regard to the performance trials presented in §4.3.2 since numerous benchmark iterations are performed in a single execution, providing ample opportunity to complete an iteration over a fully-optimized code base. For arbitrary applications, the effect could be an issue when optimizing long-running methods in-flight (using on-stack replacement), however such situations can be detected, allowing RPE to be temporarily disabled. And such measures may not even be necessary since the problematic behavior is unlikely to occur due to the fact that sub-calls are more likely to be optimized before such long-running methods.

Querying `null` *References*

One significant challenge that arises from moving query computations is the need to ensure correct execution in the face of `null` subject references. For the explicit type test operations (`checkcast` and `instanceof`), `null` subjects are entirely valid and should produce a successful cast or a `false` result, respectively. On the other hand, an attempt to dispatch on a `null` subject must produce a `NullPointerException`. However, to be a strictly valid transform, the exception must be raised when execution reaches the call site, not during an attempt to pre-compute the target. Thus hoisting query computations to avoid redundancy creates a problem: if the query subject is `null` then it cannot be inspected to obtain type or method information, but it is also not acceptable to simply give up and trigger an exception at a location which precedes the original query. Indeed, code written to handle such exceptions may execute incorrectly if it assumes that all instructions leading up to the fault are completed.

To deal with this issue and permit query hoisting—speculative hoisting, in particular—the problematic computations are protected by an explicit test of the subject reference. The technique is illustrated in figure 4.8. If the subject is non-`null`, then all consultations of the type structure are safe. Otherwise the query results are assigned appropriate default values in a fallback section. The `NULL_CHECK` is retained at the original query site and converted to an explicit test instead of

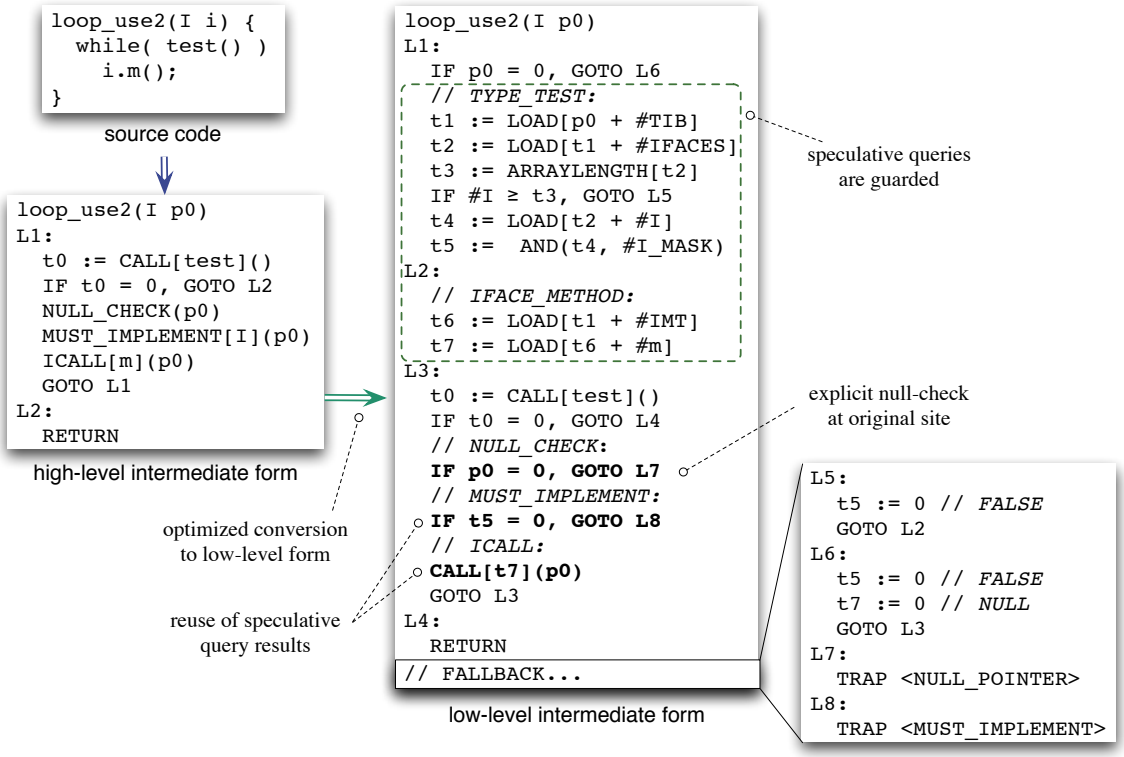


FIGURE 4.8: **Guarding Hoisted Queries**

an implicit trap. `MUST_IMPLEMENT` checks are also kept at the original site, but simply test a result computed in the hoisted section.

Fortunately, this guarding mechanism is only needed when the subject's nullness state is unknown. Thus, queries that follow other subject uses protected by the usual trap, such as `getfield` or `putfield` operations, or those applied to the `this` reference need not be guarded. To minimize the amount of guarding, an intraprocedural branching flow analysis is used to determine the nullness state of all subject references. Later, when attempting to insert query expressions at a hoisted location, the guard is suppressed if the subject is known to be definitely `null` or definitely non-`null`. Using the lazy code motion algorithm is also beneficial here, since its natural operation is to anticipate but then delay the insertion of computations until just before they are needed. This often has the effect of returning the initial TIB load to its original location following the `NULL_CHECK` for non-looping code, thus disposing of the need for an explicit guard.

Hardware Target Considerations

Intraprocedural redundancy elimination removes unnecessary computations by saving already computed results in local variables. Thus, basically, memory is traded for instructions. This trade-

off has important consequences when the high-level variable abstractions are realized in the target code format either as registers, or as stack-resident values. When too many variables are *live*—i.e., needed as inputs to subsequent operations—this creates an effect known as *register pressure*. In the simple case, this condition may require additional `move` instructions to rearrange the variable-to-register mappings. However, when not enough registers are available, this can also induce *spilling* which temporarily pushes values onto the stack to free a register for other uses.

Register pressure and spilling are quite common when targeting the Intel x86 platform since it only has 7 available registers, some of which are often constrained to be used according to certain conventions.[3] And while redundancy elimination over polymorphism queries does afford significant savings by removing several load operations—typically offsetting the cost of spilling query results—it can also increase register pressure leading to spills of other program variables which may not have occurred otherwise. Thus the challenge is to find a strategy that minimizes overall register pressure. However, co-ordinating RPE with spilling decisions is not practical in Jikes RVM, since the former is performed in the HIR optimization phase, while the latter occurs just before the MIR is assembled to machine code (see §3.1.1). Fortunately, one of the advantages of lazy code motion is that it acts to reduce register pressure by delaying hoisted computations until the latest point at which their results are needed to prevent redundant computation. Quantifying the effects of this strategy on register pressure is difficult, but to provide some sense of its impact §4.3.2 contrasts the performance of lazy code motion with its dual, busy code motion, which hoists computations to the earliest point they are needed, maximizing the interval over which redundancy results are kept.

Relocating and removing query computations can also lead to some unexpected secondary effects when targeting a specific hardware platform. As noted in §2.3.1, the dependent loads used in virtual table queries and other polymorphism operations can have a significant impact on the effectiveness of pipelined execution. Eliminating such operations, or moving them to less frequently executed zones, while likely to improve performance, may also have unanticipated effects on the throughput and scheduling of the 31-stage pipeline of Intel Prescott cores used for measurement in this thesis. On a larger scale, the reduced number of type structure queries that result from redundant query elimination should ease data cache demands. However, subject queries that occur in sub-calls, and spills induced by increased register pressure may simply shift, rather than reduce the overall cache burden. Due to the complexity of these secondary effects, no attempt is made to quantify or control their repercussions even though they certainly have some influence on the variability of RPE success.

4.3 Evaluation

Many options exist for measuring the effect of the presented RPE transform. In the past, researchers studying redundancy elimination have used static metrics, such as "number of instructions removed" [e.g., VanDrunen and Hosking, 2004a], as well as dynamic measures, such as the difference in "number of instructions executed" [e.g., Cooper and Xu, 2002]. While these measures have ob-

[3] Technically, the Intel x86 platform has 8 general purpose registers: `EAX`, `ECX`, `EDX`, `EBX`, `ESP`, `EBP`, `ESI`, and `EDI`. However, the `ESP` register is used exclusively as the stack pointer. Jikes RVM also restricts the use of `EAX` and `EDX` to serve as call argument and return values.

vious connections to the resulting code representation, the preceding discussion regarding hardware considerations reveals that the more important question is: What does the optimized code *do* on the target platform? In other words, what is the systemic effect of the transform? From this perspective, the most relevant measure is timing improvements, thus that is focus of the following evaluation. However, as outlined in §3.2, many factors that are unrelated to compilation can introduce significant variability in performance measurements. Thus the comparisons presented are tailored to focus specifically on the (mean) best result achievable through variations in compilation strategy.

The provided measurements attempt to decompose the various elements that are incorporated in to the RPE approach and show the relative effect that they have on timing results. The localized effects of query placement are contrasted by providing results for both busy and lazy code motion. The effects of tuning the approach to include guarded inlining, speculation, and just dispatch queries are also considered. The utility of selective optimization is also measured by showing reductions in compilation time. Together, these results provide justification for the various strategies employed.

4.3.1 Experimental Framework

The performance trials used to characterize RPE follow the general framework outlined in §3.2.1. To improve the accuracy of block frequency data used in making selective and speculative decisions, the run-time parameters were also adjusted to prolong the normal sampling period, delaying the transition from baseline to optimized code. Specifically, the "method sample size" buffer was expanded from its default of 3 samples (per processor) to 128, for a total of 256 samples on the dual-core test system. This buffer is used to periodically record which methods are executing and determine which ought to be passed to the optimizing compiler. Enlarging the buffer caused the VM to collect more data about which methods were executing frequently before invoking the optimizing compiler. This had the twofold effect of producing a more accurate picture of which methods should be optimized, while also extending the period over which branching statistics were gathered by the baseline code. Thus, when transitioning to optimized code, both the estimated call graph and intraprocedural branching statistics were more accurate, permitting better frequency-based decisions.

To choose the thresholds for selective and speculative decisions, insights from code inspections were combined with results from a study of the SSPRE technique applied to classic arithmetic and logic operations [Brown and Horspool, 2009b]. The latter provided good evidence for setting the selective threshold at 70% of the cumulative block frequency (designated CF=70%). In this approach, the frequencies of all a method's basic blocks are summed, then only those blocks that contribute to the top 70% of the total are retained for analysis. In regards to speculation, the previous SSPRE study was not able to identify a clearly superior threshold, although the relative frequency approach was shown to have preferable characteristics. In this approach, each block is assigned a frequency relative to the entry block. Thus, for example, a block which executes an average of 10 times per invocation is assigned a relative frequency of 10. Surveying a number of code examples suggested that a speculative threshold set at a relative frequency of 8 (RF=8) would likely be appropriate for RPE. Additional trial data provided in §A.3 shows that this threshold is, in fact, preferable to

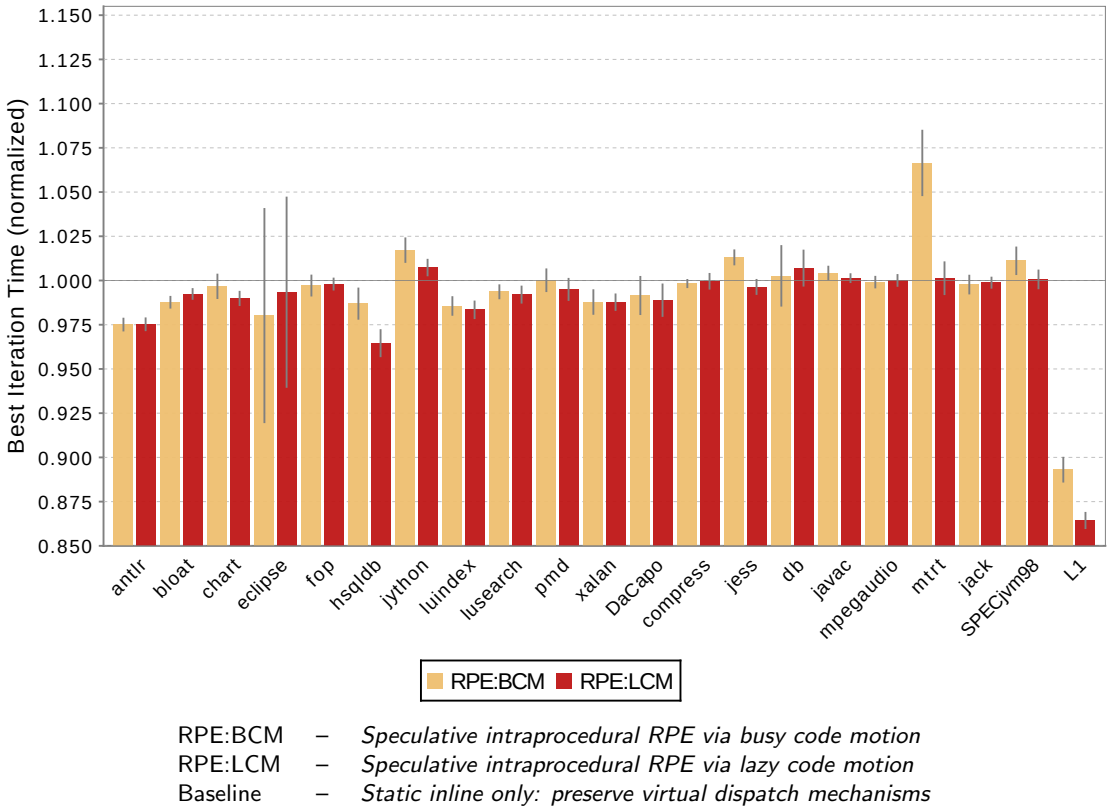


FIGURE 4.9: **Lazy vs. Busy Intraprocedural RPE Performance; No Guarded Inlining**

several other alternatives suggested by the SSPRE study.

For the RPE trials, it was also necessary to disable Jikes RVM's on-stack replacement mechanism since it is not compatible with intraprocedural redundancy elimination techniques. If enabled, using OSR to optimize methods in-flight occasionally results in invalid attempts to access uninitialized temporaries that hold redundancy results. This occurs because the results were not computed and saved by the previously unoptimized version of the code. Fortunately, disabling OSR had a negligible effect on the overall performance results.

4.3.2 Results

Figures 4.9 through 4.13 provide a summary of the key findings from trials of various RPE configurations. The complete set of results are provided in §A.3. As before, the performance charts show the mean-best iteration time for each program (averaged over at least 10 executions), normalized against the given baseline configuration. The error bars indicate the 95% confidence intervals for each result. Aggregate results for each of the two benchmark suites (DACAPO and SPECJVM98) are given to the right of the programs they contain. The focused L1 benchmark,

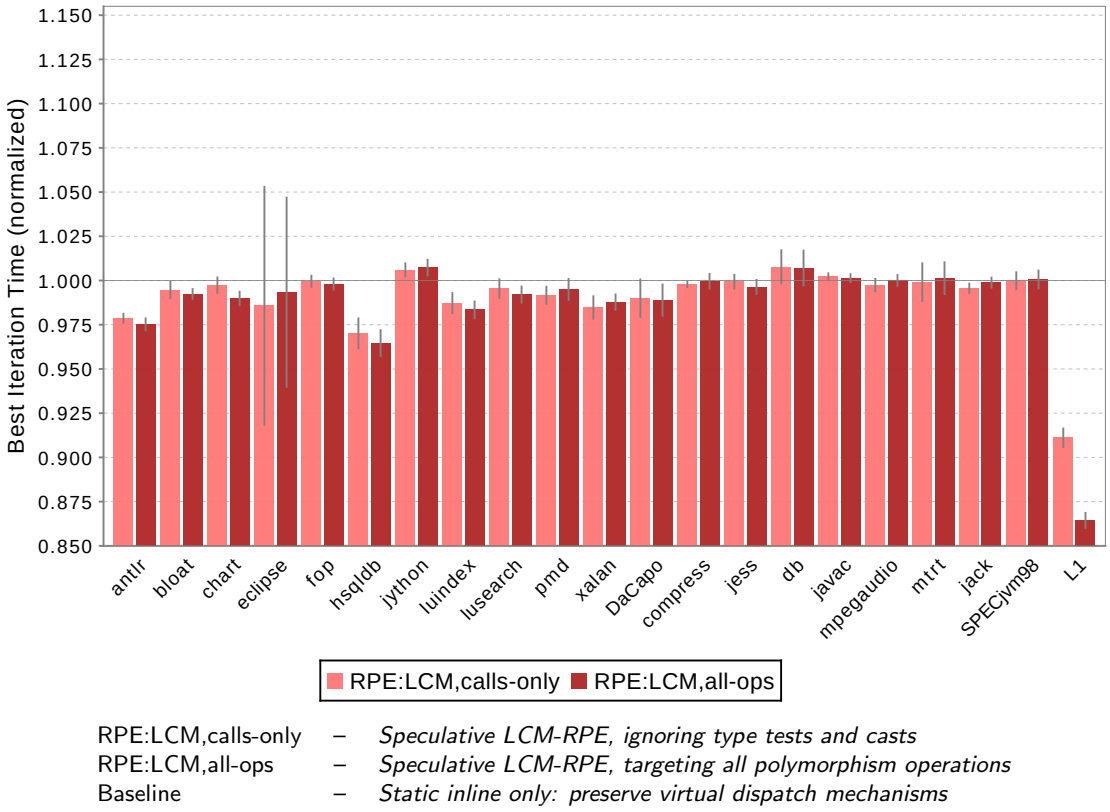
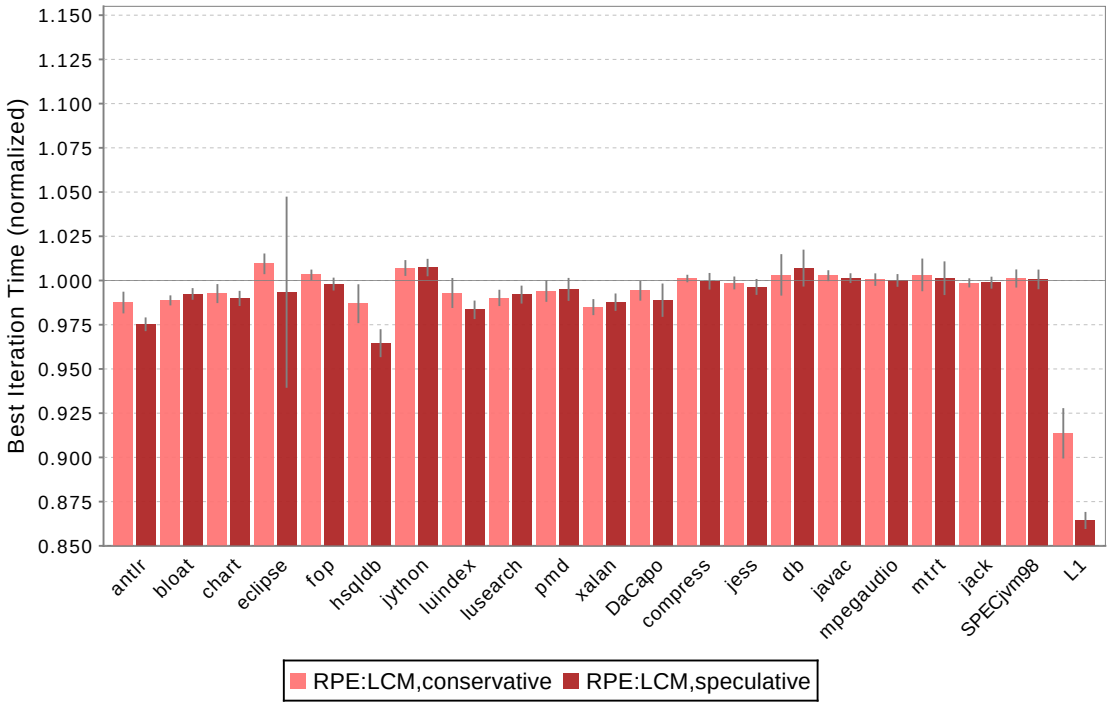


FIGURE 4.10: **Effect of Type Tests on Intraprocedural RPE Performance**

which highlights the effects of configuration alternatives, is shown on the far right of each chart.

The first comparison considers the effects of RPE over code where no measures have been taken to otherwise reduce polymorphism queries. Specifically, all of the guarded inlining techniques mentioned in §2.4.2 are disabled. The configurations perform both selective and speculative optimization, using the threshold values described above. The effects of query placement are illustrated by contrasting applications of busy code motion (BCM) against the default lazy code motion (LCM) strategy.

In the case of the LCM configuration, antlr, hsqldb, luindex, xalan, and L1 all show statistically significant improvement in the sense that their confidence intervals do not overlap with those of the baseline results. Overall, the programs in the DaCapo suite show an average speedup of 1.1%, with less than 0.5% overlap with the baseline confidence interval. Moreover, improvements are consistent across the DaCapo benchmarks, with the exception of jython which understandably confounds the process through its attempts to replicate Python dispatching using Java primitives. Also notable is the fact that none of the programs show a statistically significant degradation when using the LCM approach. The BCM results are weaker, but still show significant improvement over the same subset of programs. However, jython, jess, and mtrt are significantly worse using BCM. For both strategies, the L1 results underscore the potential of the technique by showing dramatic improvements: a 13.6%



RPE:LCM,conservative – *Conservative LCM-RPE: use traditional LCM transform*
RPE:LCM,speculative – *Speculative LCM-RPE: hoist frequent queries out of loops*
Baseline – *Static inline only: preserve virtual dispatch mechanisms*

FIGURE 4.11: **Effect of Speculation on Intraprocedural RPE Performance**

speedup when using LCM, and a 10.7% improvement using BCM.

Figure 4.10 shows the effects of excluding type test operations and only eliminating redundancy across dispatching queries. Both configurations use the default speculative LCM strategy as a basis. The findings are mixed, but a notable feature is that the performance differences are more pronounced over the DaCapo programs which are acknowledged to be more representative of object-oriented programming styles. These programs may use type tests in idiomatic ways that lead to removable redundancy, or they may rely more on dispatching polymorphism and relocating their occasional use of type tests may cause more problems than it solves. The polymorphism query profiles for chart, hsqldb, jython, and mtrt are somewhat predictive of the effect of including type tests—positive for the first two and negative for the last two (see figures B.3, B.5, B.6, and B.14). As for the L1 program, the difference is more prominent since its uses of interface dispatching entail frequent `MUST_IMPLEMENT` checks which incur a significant overhead.

The effects of using speculative code motion are illustrated in figure 4.11. Here, the first series shows the improvements realized through an application of the classic (conservative) LCM algorithm applied to all query expressions. In contrast, the default speculative approach produces a noticeable improvement across the DaCapo suite. Specifically, each of antlr, hsqldb, and luindex show a sta-

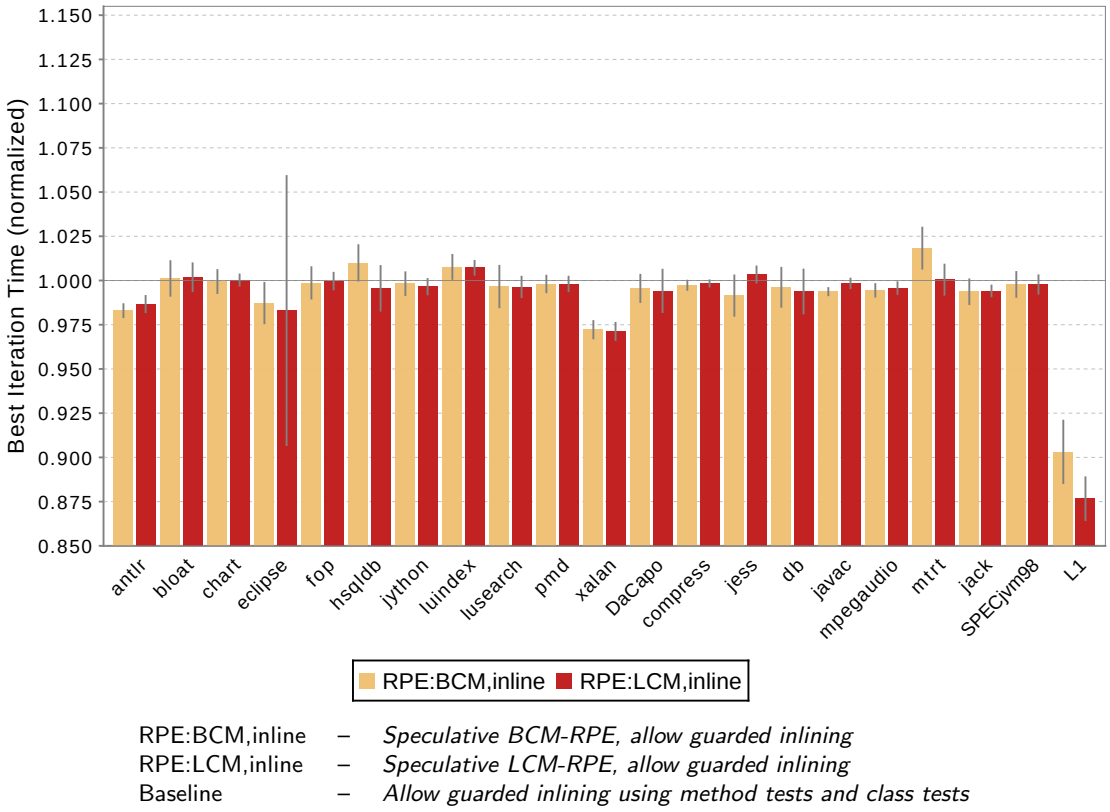
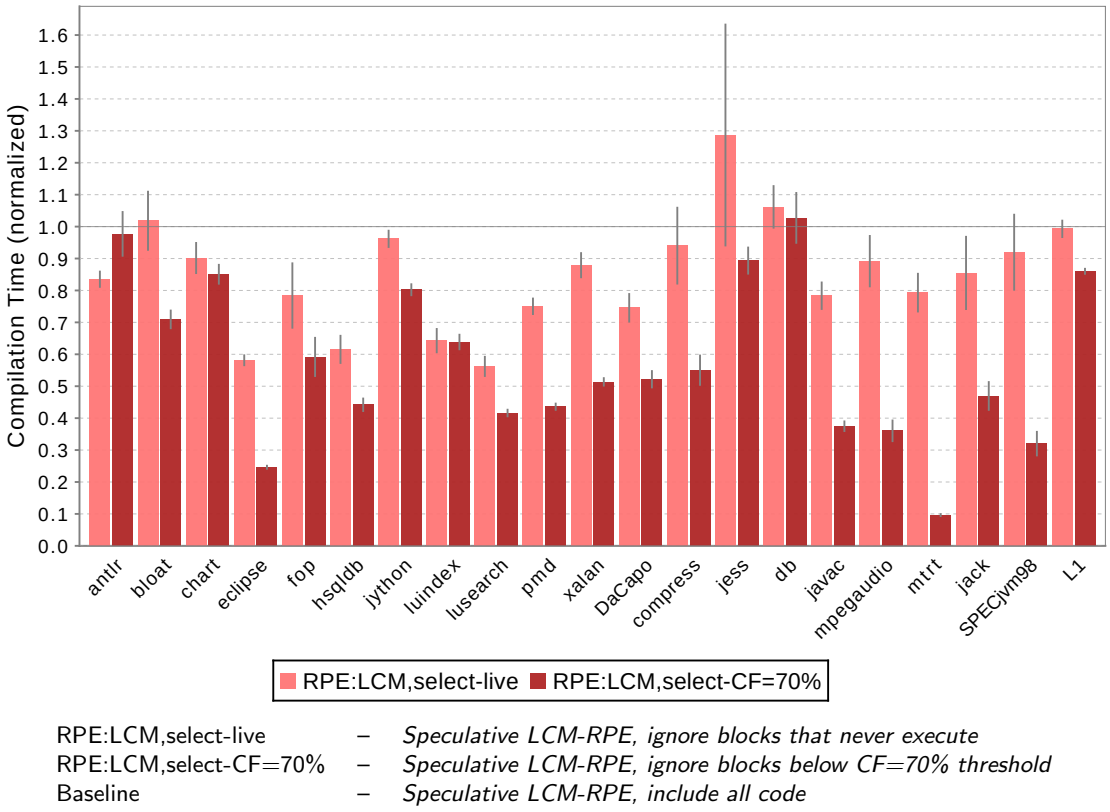


FIGURE 4.12: **Lazy vs. Busy Intraprocedural RPE Performance; with Guarded Inlining**

tistically significant improvement over the conservative approach. Also, the average result for the DACAPO suite as a whole is 0.5% better for the speculative configuration. The difference for the L1 program is, again, more striking since it includes several examples which require speculation to hoist queries out of looping regions.

The final performance comparison, presented in figure 4.12, reveals the effects of combining RPE with the guarded inlining optimizations already included in Jikes RVM's standard configuration. In this case, code fed to the RPE stage already includes inlining transforms protected by the guard operators shown in figure 4.5. These guards may still entail significant redundant querying if they are located within loops, as seen in figure 4.4(b). Enabling guarded inlining shifts the overall baseline, resulting in an approximate 12% improvement relative to the baseline configuration used for figures 4.9 through 4.11. Both the LCM and BCM strategies achieve modest speedups over this new baseline. Notably, statistically significant gains are observed for antlr, xalan, and javac. Again, the L1 result shows the most dramatic difference with a 12.3% improvement for the LCM strategy. Additionally, none of the results show a statistically significant degradation—even the BCM result for mtrt. Most importantly though, these results show how redundant query elimination supplements the site-oriented focus of guarded inlining. In particular, the gains realized for the L1 program are



RPE:LCM,select-live – *Speculative LCM-RPE, ignore blocks that never execute*
RPE:LCM,select-CF=70% – *Speculative LCM-RPE, ignore blocks below CF=70% threshold*
Baseline – *Speculative LCM-RPE, include all code*

FIGURE 4.13: **Reduction in Compile Time due to Selective Optimization**

cumulative: the best overall result coincides with an application of both techniques.

Figure 4.13 presents one more comparison which provides justification for the use of selective optimization. In this case, the chart shows the reduction in compilation time when infrequent code blocks are excluded from the analysis and transformation. The baseline in this comparison is the time spent applying speculative LCM-RPE with no selective focus. The primary selective technique (CF=70%) is contrasted here with a simple selective approach that considers all blocks predicted to run at least once over the course of a program's execution. For the given configurations, a selective focus never results in significantly poorer performance (cf. §A.3). However, it is clear that selection dramatically reduces the compilation effort. In particular, the average reduction in compile time over the DaCapo suite is nearly 50%.

4.3.3 Commentary

The two main performance comparisons in figures 4.9 and 4.12 confirm a number of the predictions made in §3.3.3. As expected, the L1 program is particularly amenable to redundant query elimination, even when guarded inlining is enabled. Additionally, the L1 results also provide strong

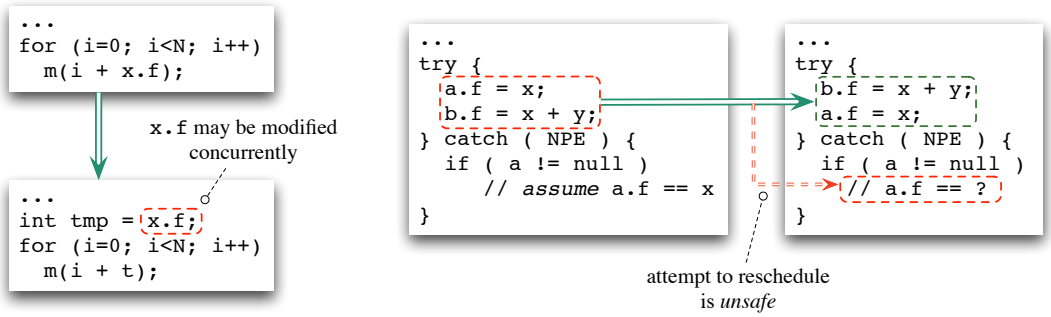
empirical support for many of the arguments and implementation choices presented throughout the preceding sections. In regards to the other benchmarks, many of the outcomes are as expected. Each of antlr, pmd, xalan, and jack showed consistent improvement. Results for the other DaCapo program, as well as jess and javac were mixed, but mostly positive. The technique had a negligible effect on the non-object-oriented compress and mpegaudio programs. And applications to db and mtrt were often more problematic than helpful. The one significant surprise was hsqldb which, despite its strong results, showed no obvious signs of being amenable to RPE.

Returning to the question of whether the technique is effective on mature, diverse object-oriented programs in general, the aggregate results for DaCapo are encouraging. In particular, the consistent and significant results for xalan should be seen as a broad indicator of success since its code base is well-developed and it presents a relatively large amount of this code to Jikes RVM's optimization system during execution. And, while the observed improvements may seem modest, it is important to remember that these gains are on top of the many sophisticated techniques already employed by Jikes RVM. Furthermore, unlike other redundancy elimination techniques, RPE focuses on a limited group of instructions which, although common, still typically comprise only a tenth of the overall instruction stream of the standard benchmarks. In contrast, the L1 results highlight the fact that programs which make more frequent use of the targeted idioms stand to benefit significantly.

4.4 Related Work

The approach of Nguyen and Xue [2004] exhibits a number of similarities to the technique developed in this chapter. There are, however, some important distinctions which are worth highlighting. In their approach, Nguyen and Xue only consider redundancy arising from the repeated evaluation of `IG_METHOD_TEST`, `IG_CLASS_TEST`, and `MUST_IMPLEMENT` expressions. Curiously, expressions that arise from the explicit polymorphism operators (`VIRTUAL_CALL`, `INTERFACE_CALL`, `INSTANCEOF`, and `CAST`) are ignored. Nguyen and Xue also use a different approach to avoid recomputing expressions confined to loops. Their technique associates an integer code with each expression. Initially, each code is set to `0`, indicating that the corresponding expression has yet to be evaluated. The query sites are then transformed to test the code before attempting to compute the guard or check result. After the first evaluation, the code is updated to reflect the result of the query. This approach is similar to the more general predication technique developed by Scholz et al. [2003]. The drawback of using such "sentinel" variables (integer codes or predicates), as opposed to code motion, is that it introduces numerous additional local variables which may dramatically increase register pressure. As previously noted, such an effect can be particularly problematic when targeting the limited register set of the Intel x86 platform.

Removing redundant polymorphism query expressions also bears some resemblance to the problem of redundant load elimination. Many approaches have been developed for this more general problem. Scalar replacement [Carr and Kennedy, 1994] and register promotion [Lu and Cooper, 1997] are techniques which avoid loads and stores by transforming memory-based computations to instead act on local variable representations. Initially though, these approaches did not account for



(a) Invalid Load Elimination

(b) Invalid Exception Reordering

FIGURE 4.14: **Invalid Transforms**

memory aliasing behavior. Several attempts to provide more complex memory models were later developed to eliminate redundant load operations [Bodík et al., 1999; Cooper and Xu, 2002]; some specifically targeted the loads implied by Java field and array accesses [Hosking et al., 2001; Fink et al., 2000]. A number of speculative approaches to load elimination have also been developed. Kawahito et al. [2004] used an ad hoc technique to effectively achieve LICM of load operations. Lin et al. [2004] adapted the SSA-PRE technique to also perform speculative load elimination. Unfortunately, while load elimination has been shown effective when applied to code with exclusive access to arrays and heap structures, many of the approaches are unsound in the face of concurrent execution. Figure 4.14(a) shows a simple example which represents a typical load elimination transform. Given the problematic nature of Java's memory access model [Pugh, 1999], such a transformation cannot be guaranteed safe without incorporating expensive whole-program analyses to fully capture aliasing an thread relations. Fortunately, the results loaded from type information structures are unique: even if the values in memory change (due to AOS updates), their meaning for the execution of the program remains invariant. This property allows the RPE technique to safely relocate query computations without worrying about the usual aliasing and concurrency issues.

Eliminating redundant polymorphism operations also has some overlap with techniques for eliminating redundant array bounds checks [Bodík et al., 2000; Qian et al., 2002]. In a sense, arrays are "polymorphic" in their `length` value, which is also fixed over the life of each array. However, eliminating redundant uses of type information is simpler since there is no need to compare the results to the outcomes of local indexing computations.

Selective Optimization

In a just-in-time environment, there is substantial motivation to maximize the yield of compilation strategies since the compilation process shares processor and memory resources with the running program. To this end, several Java VM implementations take a selective approach, focusing their

effort on highly active code while giving other code minimal treatment [cf. Arnold et al., 2005]. In most applications, code is selectively divided at the method level of granularity. To date, only Whaley [2001] and Agosta et al. [2006] have developed intraprocedural selective approaches, although neither use selectivity to focus the efforts of local optimizations as is done in the SSPRE technique used by RPE. Jikes RVM's optimizing compiler does include a feature for performing some selectivity at the basic block level, but the technique is not discussed or documented in any published work.

Speculative Optimization

Originally, Morel and Renvoise, and those who followed with similar techniques, deemed that an optimal redundancy elimination strategy removes the maximal number of redundant computations over all static execution paths, without introducing any new computations that would not have occurred in the original program. A key issue not considered by this definition is the fact that some computations are visited more frequently than others, and thus stand to offer greater gains if removed or relocated to non-repetitive sites. Many modern approaches to redundancy elimination now recognize this disconnect and attempt to incorporate an understanding of the dynamic behavior of code to reduce the overall execution load. These approaches typically employ either code duplication (i.e., *restructuring*) or speculation to expose and eliminate dynamic redundancy. Bodík et al. [2004] provide a succinct overview of how these two alternatives relate to each other. In general, the speculative approach has received more attention since restructuring often leads to problematic secondary effects such as poor scheduling and code bloat. Loop-invariant code motion is a well-known, ad hoc approach to speculation, but as outlined in §4.1, the technique is often impeded by control flow that results from other important optimizations such as guarded inlining. Numerous, more general approaches based on branching statistics have been developed to address this deficiency (see p. 62), however many are too complex to be included in a just-in-time compilation system. A more suitable alternative is to use a speculative threshold approach [Horspool et al., 2006]. The RPE transform is based on SSPRE which extends the classic formulations of lazy and busy code motion to incorporate speculation using a threshold approach. The original presentation of SSPRE [Brown and Horspool, 2009b] also describes a variety of approaches for choosing both speculative and selective thresholds.

Safety

The definition of *safe* code motion provided by Kennedy [1972] has led to a great deal of confusion regarding the difference between preventing invalid execution (i.e., faults) and avoiding extraneous computations. For example, the derived notion of *down-safety* refers to a flow analysis result that indicates whether a computation must eventually execute along all paths emanating from a given location [cf. Knoop et al., 1992]. This property is used (pervasively) by conservative PRE techniques to decide when it is acceptable to insert (i.e., hoist) a computation in order to permit the elimination of a subsequent redundant computation. However, avoiding the insertion of computations that are not down-safe does not necessarily prevent faults, even though some invalid transformations are prevented by this approach. For an optimization to be truly safe, it must prevent any reordering

that could affect the outcome of execution. In practice, this often translates into a need to correctly preserve the relation between instructions and their surrounding exception and trap handlers.

In the context of Java semantics, several approaches to permit code motion over *potentially excepting instructions* (PEIs) have been proposed. Basically, these techniques provide more freedom to reorder instructions by removing the sequencing constraints implied by run-time checks such as `null` checks and array bounds checks. Specifically, the approaches of Gupta et al. [2000], and Sundaresan et al. [2008] are predicated on the assumption that when a fault occurs in the optimized code, it is sufficient to trigger the same exception type as would have occurred in the original code. This, however, is not actually a sufficient guarantee for some cases. Figure 4.14(b) gives a simplified example that reveals how programmers may violate the assumption of these techniques by relying on subtle connections between instruction ordering and exception semantics. Odaira and Hiraki [2005] provide a more robust solution that dynamically de-optimizes code when an exception is triggered out of order. In this approach, exceptions that arise from relocated code cause execution to be suspended while the original instruction ordering is restored. Execution then resumes, allowing the expected sequence of instructions to complete before reaching the fault. The RPE technique provides a simpler alternative that accounts for both the motion of PEIs (e.g., `MUST_IMPLEMENT`), as well as the motion of other expressions (e.g., `INSTANCEOF`) across PEIs. The guarding mechanism described in §4.2.2 first ensures that it is safe to evaluate relocated expressions. If not, then appropriate default values are assigned to the result variables instead. Later, the pre-computed results are used at their original location, ensuring that any exceptions are triggered at the precise point where they would have occurred in the original program (see figure 4.8). The RPE approach thus provides a novel solution to the problem by separating the evaluation of checks from the realization of their effects.

In order to decide when it is truly safe to evaluate an expression, Murphy et al. [2008] develop an extension of the SSA-PRE technique that provides a clearer distinction between *fault-safety* and down-safety. Their approach combines the results of various run-time tests with the use of "tau" (predicate) variables to expose redundancy elimination opportunities that do not violate exception semantics. The RPE implementation similarly incorporates nullness information from operations such as branching tests and previous `NULL_CHECK`s to determine when it is safe to access a subject's TIB without guarding.

4.5 Summary

Polymorphism query expressions are uniquely amenable to intraprocedural approaches to redundancy elimination since their guaranteed invariance allows them to be easily relocated to any point that precedes their use. This chapter outlines a specific technique for deciding when and how to perform such relocations. The approach builds on the standard lazy code motion technique for eliminating partial redundancies by incorporating an efficiently computed form of general speculation, as well as an on-demand guarding scheme that ensures correct execution and exception ordering. The technique is significantly more comprehensive than previous similar techniques, and also arguably more appropriate for architectures with a limited number of registers such as the Intel x86 platform.

Several of the implementation choices for this technique, in particular the need for speculative hoisting, are guided by an analysis of repeated intraprocedural uses of polymorphism in several common design patterns. The BUILDER and ITERATOR patterns are highlighted as important examples, but the findings are also applicable to several common object-oriented idioms, as well as numerous other patterns including STRATEGY, STATE, TEMPLATE METHOD, and many of the micro patterns identified by Gil and Maman [2005] (e.g., FUNCTION OBJECT, DATA MANAGER, and OUTLINE).

To show the actual benefit of the technique, several performance trials are performed on the well-known SPECJVM98 and DACAPO benchmark suites. Consistent and statistically significant speedups are observed over many of the DACAPO programs which are acknowledged to be more representative of modern Java programming styles. In particular, improvements are observed when the RPE optimization is applied to standard polymorphism implementations, and also when combined with guarded inlining transforms.

To confirm the specific applicability of the RPE technique to usage scenarios seen in the design pattern examples, a custom benchmark program (`L1`) is included to emphasize the costs of polymorphism operations. The trial results from this benchmark clearly demonstrate that redundant polymorphism incurs a significant overhead which can be effectively eliminated by the presented approach. Moreover, the `L1` results for the various configuration alternatives underscore the utility of the comprehensive approach (including type tests in addition to dispatches), and the need for speculation.

4.5.1 Contributions

As intended, this chapter presents a new subject-oriented approach to eliminating intraprocedural polymorphism redundancies that successfully complements existing site-oriented techniques such as guarded inlining. In relating the specifics of the RPE implementation, the discussion also points to several novel developments which have relevance beyond the particular scope of this thesis.

First, the approach to performing redundancy elimination analyses over synthetic composite expressions (see §4.2.1) removes the need for repeated optimization passes. In combination with a careful choice of lexical expression representation, the result is a technique which effectively rivals more complicated single-pass strategies such as *global value numbering* (GVN) without the need to convert the input code to *static single assignment* (SSA) form [cf. Rosen et al., 1988].

The utility of the expression representation is further enhanced by applying it in the context of the SSPRE analysis and transformation framework used to implement RPE. This framework was conceived and developed—independently from the isothermal speculative approach (ISPRE)—to extend the well-known lazy code motion algorithm with speculative capabilities while remaining focused and efficient. By itself, the development of this framework is a significant contribution. Not only is the result a clear generalization of the LCM approach, it also represents a unique departure from the tradition of monotone analyses initiated by Kam and Ullman [1977]. A separate study [Brown and Horspool, 2009b] demonstrated that the framework is also applicable to the elimination of classic forms of redundancy over arithmetic and logic computations. The results in this chapter

provide confirmation that both the selective and speculative aspects of SSPRE add value to the standard LCM approach.

In applying code motion to eliminate redundancies in Java code, a challenge that immediately arises is the need to preserve correct execution behavior in the face of ubiquitous run-time checks such as `null` checks, type checks, and array bounds checks. Previous techniques for addressing this problem have either relied on faulty assumptions, or required elaborate analyses or fallback mechanisms to ensure a valid execution sequence. The approach to dealing with exceptional cases developed for RPE is, however, both simple and correct (see §4.2.2). The key to the approach is the division of the problem into three separate stages: 1) Determine whether it is safe to evaluate the check, if not produce a suitable default result. 2) Evaluate the conditions of the check, producing a boolean result. 3) Test the check result at the original site, triggering the appropriate exception if necessary. The first two elements can easily be hoisted without altering execution semantics, while the last step ensures that the effects of these checks are realized as intended in the original program. Additionally, when combined with a LCM strategy, the partitioning of checks only occurs as needed: many of the checks remain intact at their original location, while the few that induce redundancy in looping regions are split to allow their expensive elements (typically load operations) to be hoisted to less active regions. This particular behavior is worth noting since it provides yet another reason for preferring LCM over other PRE strategies, in addition to the classic argument that a lazy strategy reduces register pressure by minimizing the interval between evaluation and use of a result.

4.5.2 Directions

Object-Specific Optimizations

As suggested in §2.6.2, the RPE concept could be extended to consider queries of other fixed object properties. Certainly several of the common array operations are good candidates. Accessing an array's `length` attribute presents many of the same challenges as accessing an object's TIB. Also, this value is needed to implement frequent bounds checks that, in the same way as `MUST_IMPLEMENT` checks, are not immediately apparent in the source or bytecode representation. Hoisting bounds checks in their entirety is more challenging though, since they also entail a comparison against local values used as indices. Storing values into reference-type arrays (via an `aastore`) also implies a type check operation which may be redundantly evaluated. However, this operation is more complex than the other, single-subject polymorphism operations since it involves dynamically testing both the array and incoming reference. In many cases, this presents a challenge since redundancy only occurs in querying the former, but not the latter.

Design Pattern Optimizations

The PROTOTYPE pattern, and possibly the FLYWEIGHT pattern as well, present some interesting possibilities for extension of the RPE technique. In both cases, these patterns establish a similar typing relation across different object instances. Specifically, a PROTOTYPE's `clone()` method and

the lookup method of a FLYWEIGHT factory presumably produce objects with the same type. If this behavior could be guaranteed—possibly through the use of compiler-enforced annotations—then the results of some polymorphism queries could, in principle, be reused for several different subject references. Such an optimization would be particularly apt for FLYWEIGHT, since part of its intended purpose is to provide a more efficient representation for frequently used objects.

Trace-Oriented Optimizations

The subject-oriented focus of RPE might also prove to be a useful addition to the approach of optimizing over instruction traces rather than intraprocedural control flow graphs [e.g., Gal et al., 2009]. Trace-oriented optimization is, by nature, highly site-oriented—or perhaps, more accurately highly path-oriented. Indeed, one of the key advantages offered by the trace-oriented approach is that it obviates the need for flow analyses, simplifying speculation and automatically producing a selective focus. The challenge, however, as with other forms of specialization, is to keep a reasonable bound on the number of distinct traces. Taking a subject-oriented view may be helpful with this problem in that it may facilitate the unification of several traces which represent polymorphic behavior over the same code sequence.

Fault-Safe Code Motion

As indicated, the guarding scheme used to ensure correct execution in the face of `null` subject references has obvious applications beyond the scope of polymorphism operations. In general, the semantics of potentially excepting instructions provide a significant obstacle to code motion, speculative code motion in particular. Applying a similar partitioning to other run-time checks (notably array bounds checks, array store checks, and divide-by-zero checks) could also be used to further eliminate check redundancies and provide more flexibility in hoisting other expressions that would normally be blocked by such checks.

Chapter 5
Instance-Lifetime Redundant Polymorphism Elimination

The experiments of the preceding chapter demonstrate that there are significant examples of polymorphism queries which exhibit redundancy over the scope of a single method execution. Since the span of these contexts is often brief though, one might naturally wonder whether there exists a way to capture and reuse query results over a longer time frame. Exploiting redundancy across multiple invocations is challenging, however, since the only common space shared by a sequence of calls is the process stack. This space may be readily accessible, but the discussion in §2.5.1 reveals that the potential savings of exchanging redundant results via the stack are not likely to outweigh the costs of simply performing the queries directly. Fortunately, in an object-oriented setting the objects themselves can also be seen as defining a context which persists longer than a single method execution. Indeed, the range of some object contexts may even extend beyond the scope of an entire stack associated with one thread of execution. But how can the context defined by an object instance be used to cache redundant query results any more efficiently than when using stack space? The key distinction, which offers an advantage in the case of object contexts, is that objects are, by design, intended to have custom behaviors. Thus computations that can be identified as redundant throughout the whole lifetime of an object can be folded into the representation of the object itself, creating a new, polymorphic derivation.

This is the essence of optimization techniques such as *customization*. However, realizations of this idea still remain rooted in a site-oriented, or in this case, a code-oriented perspective: only modifications that are applicable to all instances of a particular class are included. On the other hand, taking a subject-oriented view reorients the problem to consider how the subjects—in this case, field values—accessible from a particular instance can be used to adapt the implementation of the holder based on the subjects' characteristics. Once again, this view presents an alternative to the traditional approach, and suggests a new avenue for developing optimizations that complement existing techniques.

In keeping with the theme of the current thesis, this chapter describes a subject-oriented approach which aims to eliminate the overhead of polymorphism queries that are redundant over the entire lifespan of an instance with particular field bindings. For normal object instances, the technique imbues candidates with the ability to modify their own implementation to use specialized code where the results of polymorphism operations applied to unchanging reference fields are pre-computed,

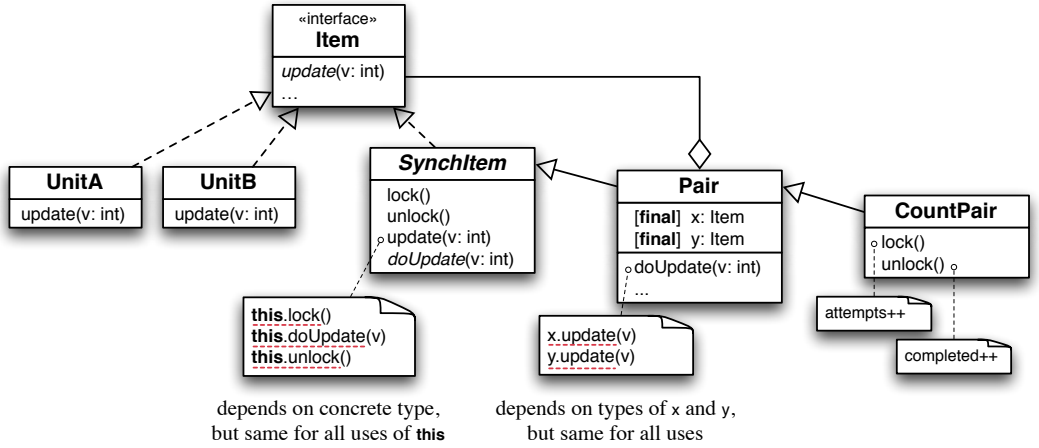


FIGURE 5.1: **Instance-Lifetime Polymorphism Redundancy**

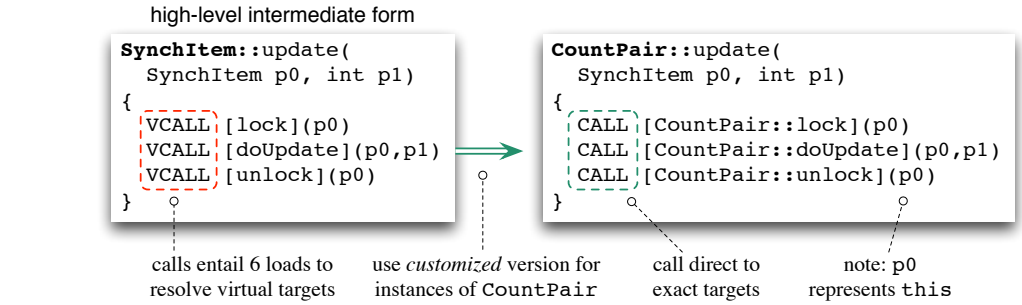
allowing virtual dispatches to be converted to direct calls and type tests to be eliminated. The notion of an "instance" is also expanded to include unchanging `static` reference fields associated with a particular class, and in such cases specializations of polymorphism queries applied to these fields are performed uniformly throughout the code base.

Given the predictions regarding instance-specific polymorphism redundancy laid out in §3.3.3, the measured effects of the proposed technique yield some surprising results. In particular, while the amount of observed polymorphic behavior is reasonably low, the analysis suggests some new directions for expanding the concepts of guarded inlining and inline caching to cover the lifetime of individual object instances.

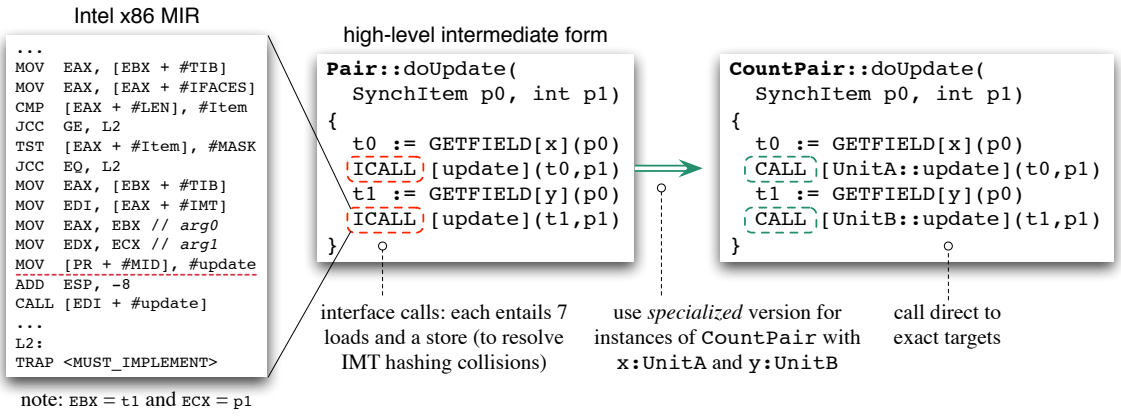
5.1 Problem

The utility of intraprocedural redundant polymorphism elimination rests on the assumption that at least some queries are repeated over the course of a method invocation. But what if no repetitions occur? Such "straight-line" methods are, in fact, quite common in object-oriented programs. This is confirmed by an examination of the basic block frequencies observed by Jikes RVM's optimizing compiler, which shows that roughly 83% of all hot methods from the DaCapo and SPECjvm98 programs have a maximum relative block frequency less than 2.0 [Brown and Horspool, 2009b]. This means that many, commonly used methods perform no definite looping.

At the same time though, it is conventional in object-oriented programming to define objects as compositions of other objects, and create polymorphic behavior by delegating to subcomponents. Indeed, this is the essence of the second key design principle espoused by the Gang of Four (cf. §2.2.4). In many cases, these delegation relationships are used frequently, yet remain fixed over the lifetime of the composing object. For example, consider the simple design illustrated in figure 5.1. In this



(a) Customization



(b) Specialization

FIGURE 5.2: **Instance-Specific Optimizations**

scenario, a `Pair` creates a fixed aggregate of two coupled `Item` delegates. Every time an `update` is applied to the pair, the message is forwarded on to each of the parts, and since the parts may be any `Item` type, these forwarded dispatches are necessarily polymorphic calls. However, resolutions of the targets of these calls are clearly redundant on the second and subsequent applications of an update operation. Self-calls—those applied explicitly or implicitly to the `this` reference in Java—also need to be implemented as polymorphic calls, but exhibit similar redundancy once invoked from any particular instance. In the example, the (inherited) implementation of `update` defined in `SynchItem` uses local methods that can, or must be overridden by subclasses. However, when executed from an instance of `CountPair` the targets of these calls are determined. Thus the challenge is to remove these examples of redundancy while preserving the polymorphic nature of the code.

Figure 5.2 outlines two approaches to refining the method implementations of `CountPair` to avoid the overhead of redundant queries that resolve virtual and interface dispatch targets. In this case, implementations that are normally inherited from the parent classes are replaced with specialized

versions for a particular instance of `CountPair`. The self-calls in `update` are devirtualized using the *customization* technique which binds each call directly to the local implementations. As long as customized implementations are created for all methods, this transformation is valid for all instances of `CountPair`, and no additional guarding or preparation is required. On the other hand, the call targets needed in the implementation of `doUpdate` are determined by the `x` and `y` fields, which may be instantiated as any concrete subtype of `Item`. As an example, if the `x` field is known to be an `UnitA` value and the `y` field is known to be an `UnitB`, then the interface dispatches can be similarly devirtualized by creating a *specialized* method implementation. In this case, the specialized version offers significant savings. Not only are the loads involved in an interface call (and its associated `MUST_IMPLEMENT` check) avoided, but in this case, a direct binding also removes the need for an additional low-level store used to resolve IMT hashing conflicts [Alpern et al., 2001].[1] However, this latter transformation is predicated on two key assumptions: (1) the exact run-time types of `x` and `y` are known, and (2) these types are guaranteed never to change for all uses of the specialized `doUpdate`. In the example, the second criterion is satisfied by the fact that `x` and `y` are declared as `final`—that is, once they are initialized, their values cannot be altered. The first premise, however, is more difficult to assure, especially since numerous instances of `CountPair` may be created with differently typed values for these fields. Thus some additional measures need to be taken to verify the particular bindings of a `CountPair` before it is possible to take advantage of this optimization opportunity.

5.1.1 Lessons from the Decorator Pattern

The Decorator design pattern exemplifies a very common idiom in object-oriented programming wherein one object is used as a stand-in or replacement for another. Typically the replacement has access to the original value, but also adds new functionality in a way that is transparent to clients. An example is the layering of image representations as "filters" depicted in figure 1.4. In this case, the client (the `ImageProducer`) is unaware that the original representation has been extended with a cropping window. Like Decorator, the Proxy pattern also represents a similar construction where the replacement has an equivalent type to the original. On the other hand, Adapter defines a stand-in with a different type signature, essentially repackaging the capabilities of an object so that it can be manipulated in terms of a different abstraction. What is common to each of these forms is that the relationship between the stand-in and the delegate is often polymorphic—i.e., multiple inner/outer combinations are possible—yet the binding between two particular instances is typically permanent and established during construction of the outer value. Moreover, most of the actions of the outer value are defined in terms of operations applied to the inner value. In other words, an invocation on an outer method almost surely entails one or more operations on the inner value. In choosing a representative to model these sorts of relationships, Decorator provides the most straightforward example since the default behavior of the majority of its methods is to simply relay

[1] When targeting the PowerPC architecture, this value is saved in a specially designated register. For the Intel x86, since so few registers are available, the value is stored in a field of the current process handle.

the call on to the decorated item.

Several other design patterns also exhibit a fixed coupling with similar structure and behavior to Decorator and Adapter. What distinguishes them from the previous examples though is how they are perceived from a design perspective. While a Decorator essentially augments an existing abstraction, patterns such as Bridge, Strategy, Chain of Responsibility, and Facade each express a relationship where the inner delegate (or delegates) complete the abstraction of the outer representation. The Template Method pattern is a variation on this notion in which an incomplete representation delegates to itself, presuming that concrete subclasses complete the missing elements. The Model-View-Controller pattern, and the more general Mediator are also often implemented using fixed relationships, but the communication between elements is typically bidirectional and no single object acts as the dominant wrapper. However, despite their differing architectural roles, what unifies these examples is that, just as in the basic Decorator, the relationships are polymorphic yet bound permanently during the initialization of a specific instance coupling, and operations on the holder usually lead to operations on the delegate(s). Gil and Maman [2005] label this general form a Canopy.

In fact, this form of coupling is so prevalent that the Java language supports it directly through the *inner class* concept. A simple example is given in figure 5.3. In the source representation, an inner class is defined inside the scope of another class, and instances have direct access to the fields and methods of the outer instance in which they are created. Such an arrangement does not, however, exist in the JVM bytecode format. When compiled, the definition of the inner class is actually enhanced to simulate the intended behavior by holding a hidden, `final` reference to the outer instance. Note, however, that the inner/outer terminology is inverted here relative to the earlier discussion: the "inner" `B` values, in effect, act as wrappers over instances of type `A`.

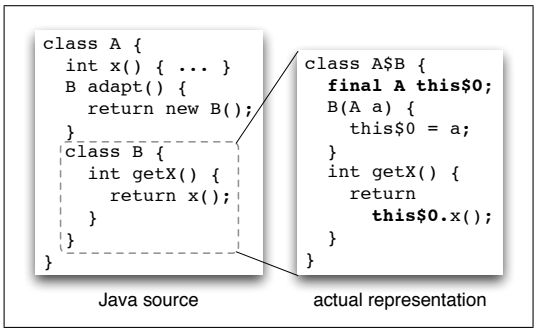


Figure 5.3: **Java Inner Class**

Inner classes are frequently used to provide special purpose views or handles on an object because they permit access to the private state of instances of the enclosing type. Figure 1.2 highlights an example of this common idiom, taken from Java's standard `ArrayList`, in which an iterator over the list is defined using an inner class, essentially providing an adapted view of the list.

The point is that such layered structures, regardless of their intent, can all be transformed using the optimizations outlined in figure 5.2. Moreover, the characteristic which makes specialization of polymorphism operations on inner fields particularly appealing relative to other kinds of specializations over field or parameter values is that only a limited number of inner/outer type combinations are likely to occur. This permits specializations to be reused by many similarly typed instance/field combinations, significantly reducing the amount of specialized code.

The L3 benchmark, described in §C.2, is designed to highlight the cost of repeatedly resolving

the polymorphic relationship between a wrapper and its delegate. The program builds a number of DECORATOR-like instance chains with differing inner/outer type combination and invokes a heavy load of dispatches that are relayed down the chains. Although a simple construction, the resulting behavior confounds guarded inlining since the dispatch sites appear highly polymorphic even though the targets are extremely regular with respect to each wrapper instance.

5.1.2 Lessons from the Singleton Pattern

While the majority object-oriented code is formulated in terms of normal object relations, class-based languages also provide another instance-like construct in the form of `static` class data. The `static` fields of a class are not aggregated in memory the same way that fields of an object are, but they do, effectively, have similar semantics: access to the fields may be restricted to a particular scope (e.g., `public`, `private`, etc.), and is often mediated by methods of the class—`static` methods in this case; likewise, `final` references may appear polymorphic from a compile-time perspective, but are definitely bound to a specific type once initialized. Thus, a similar technique to the one illustrated in figure 5.2(b) can also be applied to uses of such fields. The difference, however, is that there is always only a single "instance" of a class' `static` data. This simplifies the situation greatly, allowing a unique specialization to be derived for any use of such fields.

The idea of using a class as a wrapper is exemplified by the SINGLETON pattern. In a sense, the architecture is similar to that of a BRIDGE. The exposed, `static` methods and fields of the class represent the outer, in this case, globally accessible handle, whereas the instantiated singleton object provides the implementor, or delegate that supplies the implementation logic. The MULTITON pattern is a generalization of SINGLETON where multiple inner values are created. Often the MULTITON class acts as a directory of special-purpose instances.

The motivation behind SINGLETON and MULTITON is that they effectively provide a *module* abstraction. Modules, like objects, represent entities that export certain capabilities, carry state, and have a well-defined initialization process. However, unlike ordinary objects, modules are typically accessible from anywhere in the program, and because of this they often persist for much longer time frames. In the context of Java, classes—not to be construed with `Class` objects—are a natural choice for creating module abstractions since the definition of each class is maintained in separately loaded files, and the JVM standard defines an explicit process, similar to object construction, for initializing the `static` fields of a class [Lindholm and Yellin, 1999]. The *assembly* construct defined by the .NET/CLI system also has some characteristics of a module, but with a less formal initialization procedure.[2] Aspects [Kiczales et al., 1997], a construct defined in, for example the AspectJ extension of Java, similarly represent module-like metaobjects which, in effect, decorate a program's logic. Together, these examples, and the wide variety of other ad hoc techniques in which `static` data is used to simulate a global "object" representation, suggest that it is also worthwhile to include `static` fields in a systematic approach to eliminating object-specific polymorphism redundancy.

[2] Although the loading of a assemblies can be monitored using event handlers, there is no analogue of the class initializer routine for assemblies.

5.2 Proposed Solution

To achieve the specialization reduction suggested in figure 5.2(b), it is first necessary to identify which object instances may benefit from such a transform. As the example indicates, objects that contain reference fields that are declared as `final` are good candidates since the JVM guarantees that these references are never modified after they are initialized in an object's constructor. As a consequence, all applications of polymorphism operations to particular values for such fields must produce the same results since their types definitely remain invariant. However, programmers are not always meticulous in applying the `final` modifier. Indeed, Unkel and Lam [2008] demonstrate that, for many Java programs (including the SPECJVM98 suite, but not the DaCapo programs), fewer than a fifth of all reference fields are declared as final, but often more than half can be deemed *stationary*—that is, all writes to the field occur before any reads. Thus, it is reasonable to presume that a more broadly defined set of fields may be used as candidates. And, of course, a specialization transform is only worthwhile for methods that actually use the identified candidate fields.

The next challenge is to determine which particular specializations to generate, and how to invoke them given a particular inner/outer type combination. Since the specialization bindings are determined by information that is held within an object, rather than constant parameter values identified at the call site (as in traditional approaches to specialization), it seems only natural that the candidate objects themselves should contain the logic for inspecting the types of their fields and switching their implementation to use an appropriate specialization. Additionally, this logic may even inspect the run-time type of the wrapper object to facilitate customization, as in figure 5.2(a). However, this process must not interfere with the implementations used by other objects of the same type which may contain differently typed fields, and hence require distinctly different specializations. Thus, in general, the procedure must be instance-specific, with the exception that specializations over unchanging `static` fields may be applied universally to all methods.

5.2.1 Code Analysis and Transformation

Because the Jikes RVM compilation process operates as an integrated part of the run-time environment, it is fairly straightforward to extend an object's implementation with a call that transfers back into the compilation system, permitting a reflective inspection and modification of code associated with the subject instance. In the case of the proposed optimization, such a procedure must first determine whether an instance contains fields that support specialization; then examine the fields to determine their run-time types; apply this information to generate new, specialized code for the originating method; and finally, install this new implementation such that subsequent invocations on the same subject dispatch to the specialized code with polymorphism operations reduced or eliminated. However, since the ultimate aim is to simply reduce the total number of loads incurred by unnecessary polymorphism queries, the costs involved in performing this specialization process must, over time, become increasingly marginal in order to yield any net benefit. The following sections detail how this process is performed, and the measures that are taken to keep the overhead in check.

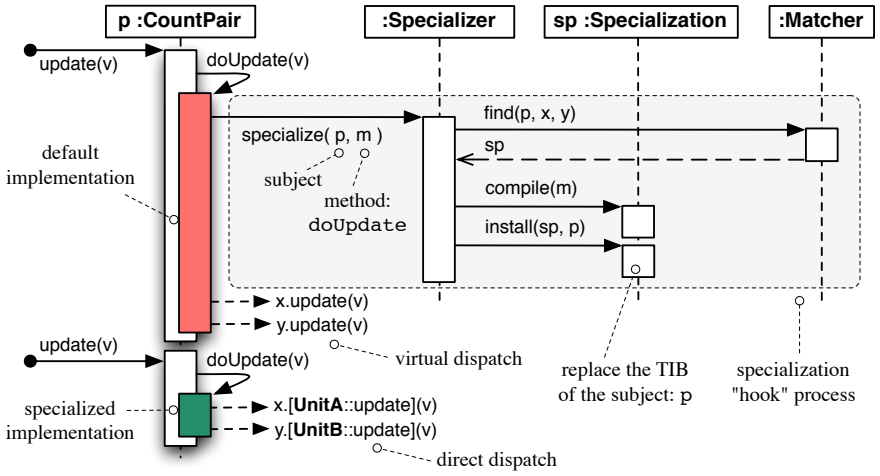


FIGURE 5.4: **Installation of Specialized Method Implementation**

Identification and Installation of Specializations

To have potential specialization candidates identify themselves, the usual optimizing compilation process (which is only applied to hot methods) is augmented to insert a call to a special "hook" at the start of instance methods which perform a load of any candidate field from the `this` reference. Later, when such a method is invoked, the hook is passed a reference to the current method representation and `this` value. Control then transfers from the main program into the compilation system. At this point, the run-time type of the instigating object is inspected, and based on the result a tailored "matcher" inspects the appropriate candidate fields to determine their run-time types as well. Together, the combination of wrapper and delegate types is then used to locate a specialization context which holds refined method implementations appropriate for the particular wrapper and field type values.

Once a specialization context has been identified, the hook then determines whether a version of the instigating method has been compiled for the particular inner/outer type combination. If not, a new version is compiled, applying a process (described in the next section) which uses the contextual instance information to reveal the exact types of certain operands, allowing polymorphism queries to be pre-evaluated thus paving the way for further optimizations. The new method implementation is then installed in a duplicate of the wrapper object's TIB structure which is maintained by the specialization context. Finally, the wrapper's TIB reference (as depicted in figure 2.1) is updated to point to the duplicate, specialized TIB instead of the primary version shared by instances with the same run-time type. The hook method then returns control to the program and resumes execution of the target method. However, since the originating object's TIB reference has been switched, all subsequent invokes applied to the same object proceed directly to specialized method implementations. The process is illustrated in figure 5.4 using the `CountPair` example from §5.1.

Over time, more wrapper/delegate combinations are identified in this way, and the spectrum of possible specialization contexts and specialized method implementations is expanded. Eventually, all the necessary method implementations are generated and the hook procedure reduces to a simple look up and install of the appropriate specialized TIB. Assuming that this case becomes the major determinant of whether or not instance-specific polymorphism specialization is able to improve performance, the look up process is refined to use a light-weight hashing scheme based on reference equality and pre-computed hash codes.

Compiling Specializations

To support the production of specialized method implementations, an optimization dubbed Type Expression Folding (TEF) is added to the HIR phase in Jikes RVM's optimizing compiler. This procedure performs a forward, branching flow analysis over the control flow graph of a method to identify the exact run-time types of as many local values as possible. The technique is similar to the analysis used to infer nullness in §4.2.2, except that the lattice of flow values (depicted in figure 5.5) is expanded to include all possible concrete types. The distinction with this approach, as opposed to other type refinements performed by Jikes RVM, is that values identified as having an *exact type* definitely represent concrete instances of the particular type, or are definitely `null`. Knowing the exact type for a reference permits any polymorphism query applied to it to be pre-computed and thus effectively eliminated, whereas Jikes RVM's notion of "precise" type, on the other hand, does not guarantee that a value is non-`null`.

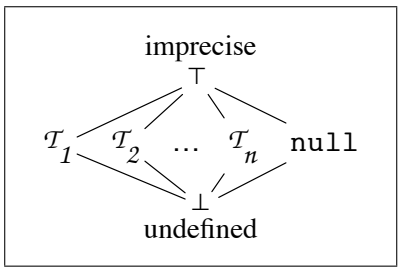


FIGURE 5.5: **Exact Type Lattice**

With exact type information propagated from the definition of variables through to their uses, it is possible to perform the following reductions:

- virtual and interface dispatches can be transformed to direct calls;
- class test and method test guards (cf. §2.4.2) can be replaced with direct jumps;
- `INSTANCEOF` operations can be reduced to a definite `true` or `false` value;
- `CAST` and `MUST_IMPLEMENT` checks can either be eliminated, or reduced to a direct trap; and
- certain tests against `null` or other constants can be directly evaluated.

Several of these transformations also facilitate subsequent branch folding and dead code elimination.

However, in practice, the opportunities to perform these reductions are rare given the type information that is typically deducible from an intraprocedural context. Thus, in order to be of any significant value, the type analysis needs to be enhanced with additional information, for example, the exact types loaded by `getfield` and `getstatic` operations. In the case of the latter, if the holding class' initialization process is known to have completed (a simple test in Jikes RVM) then

the types of these fields, if they are identified as unchanging, can be inspected directly from within the compilation process. This permits specialization of classes-as-instances, as described in §5.1.2, and is henceforth referred to as "CS" (Class Specialization). On the other hand, the types of instance fields depend on the particular instance that they are held by. In this case, the TEF process can be further refined by including a known set of bindings for the `this` reference and its fields, as supplied by the specialization "hook" procedure described in the previous section. Obviously, the code output by this particular application is only valid for the given context and others with equivalent type bindings, thus it is only installed in the specialized TIB duplicates. In the following sections this extended use of the TEF process is referred to as "IS" (Instance Specialization).

Capturing More Candidate Fields

As noted previously, restricting the specialization approach to only consider fields actually declared as `final` may significantly limit the number of examples identified and transformed by the contextualized TEF procedure. One alternative is to consider the approach of Unkel and Lam who outline a comprehensive analysis for identifying many more fields which are effectively `final` in the sense that once they are read, they are never subsequently written to. However, as with other whole-program analyses for Java, the results of this technique are highly dependent on the ability to successfully construct a precise call graph over the entire set of classes.

At the risk of engaging in a significant digression to incorporate such an analysis into the Jikes RVM just-in-time framework, for the purposes of the following experiments, it was decided to instead use a simple yet easily computed approximation of the set of *effective-final fields*.[3] To identify these candidate fields, the Soot framework [Vallée-Rai et al., 2000] was used to perform a flow-insensitive analysis over each of the classes in the studied benchmarks looking for fields that are only written to in constructors (or class initializers) of the class in which they are declared. These fields are deemed effectively final in the sense that, once their enclosing instance has been initialized, all subsequent uses in normal methods—`static` methods for `static` fields, and instance methods for instance fields—witness the same field value, and hence the same field type.

In total, this analysis identified 1588 `final` reference fields in the benchmark classes, including both `static` and instance fields. Additionally, nearly 4000 other fields were also identified as being effectively final, although a small number needed to be eliminated from consideration due to unsafe uses during the execution of their holder's constructor(s). The evaluations given in §5.3 contrast the resulting effects of including these fields as candidates when identifying and specializing for specific instance and class contexts.

5.2.2 Issues

Similar to the intraprocedural RPE approach, there are a number of technical issues that arise from the implementation of instance-specific specialization described in the previous section.

[3] This term is used to avoid confusion with the more formal notion of *stationary* fields defined by Unkel and Lam.

Code Size Increase

One problem that is common in applications of customization and specialization is an inordinate increase in total code size (cf. §2.4.3). Fortunately, the specific focus on reducing polymorphism operations allows the specialized methods to be reused for many different wrapper/delegate value combinations, since the optimizations are based on the types of the fields, not their specific values. In practice, although the total amount of optimized code does increase for some benchmarks, the measurements given in table 5.1 confirm that, for the most part, this growth is reasonable. A few of the programs, notably compress, db, and L3, show large relative increases in code size when including effective-final fields, but it should be noted that these programs have a very small code footprint to begin with.

Adaptive Optimization Updates

As noted in §4.2.2, the method addresses held in TIB structures do not actually remain constant over the course of an execution. When certain methods are identified as hot, the adaptive optimization system recompiles them with more aggressive optimization and replaces the links to these methods in the TIB and IMT structures of all classes which inherit the particular method implementation. The specialization process, however, creates duplicates of the primary TIB structures, thus it is necessary to monitor this update process and install the improved method pointers in the specialized TIBs as well.

The upgrading of method implementations also creates a problem for code that has been compiled to bind directly to an old implementation for such methods. Normally, Jikes RVM discards its handle on the representation of old code, causing it to be reclaimed by the garbage collector.[4] To avoid this behavior, it was necessary to disable the retirement of old code.

Although the possibility of binding to older method implementations is a potential weakness, as pointed out in §4.2.2, there is good reason to believe that the adaptive optimization system will transform methods at the fringes of the call graph before those that call them. Thus, ultimately, the specializations are likely to link to subcalls that have been comparably optimized.

Exact Type Tests

A problem related to the duplication of TIB structures is that Jikes RVM optimizes type tests against terminal types—those that cannot have any subtypes—by performing a direct pointer comparison against the subject's TIB reference. Obviously such comparisons will yield invalid results if applied to subjects that have been specialized. Fortunately though, it is possible to correct for this situation by adding an extra level of indirection and comparing against the pointer to a subject's type representation (see figure 2.1). This reference is equivalent in both the primary and duplicated TIBs for each individual type.

[4] The garbage collector only inspects references emanating from known variables. It is ignorant of references embedded directly in the code representation.

Multiple Wrapper Layers

In some cases, a candidate wrapper instance may hold another wrapper as one of its fields. When this occurs, it would be inappropriate to bind a dispatch on the inner wrapper directly to the default implementation which contains the hook process described above. This would cause invokes from the outer to inner wrapper to continually attempt to specialize the inner value, whereas normally the process is only performed once on the first use of any default implementation containing the hook. To avoid this degenerate behavior, specialized dispatches are always bound to a version of the target implementation with no hook. These implementations are compiled as needed. Of course, normal virtual or interface invokes applied directly to the inner value (from an unspecialized context) install and subsequently resolve to the specialized code for the inner instance.

Indirect Calls

The study of Driesen and Hölzle [1996], discussed in §2.3.1, revealed that a significant factor in the cost of virtual dispatching is the burden incurred by performing indirect calls to a computed target address. Predicting the outcome of such calls, in order to maintain pipelining throughput, is challenging and requires significantly more hardware resources than is needed to anticipate the results of simple conditional branches. Indeed, the basic premise of inline caching is that it is more efficient to perform a test, branch, and direct call, rather an indirect call. Unfortunately though, Jikes RVM's assembler for the Intel x86 platform does not support actual direct calls to an explicit target address. Instead, directly bound calls are represented in the MIR by first placing the constant address in a register and then performing an indirect call on the register. A review of the available literature on the Prescott architecture and Intel x86 instruction set [Intel, 2009b,a] did not reveal whether this particular use of indirect calls would be reduced to a direct call in the microcode translation. Thus, it remains unclear whether the specialization transform described here is also able to reduce the branch prediction load, in addition to eliminating the loads and other computations that comprise polymorphism queries.

5.3 Evaluation

In order to assess the utility of the proposed approach to specialization, performance trials were run using a variety of configuration alternatives. As in previous measurements for the intraprocedural RPE technique, the trials here are divided into two major groups: those that permit method test and class test guarded inlining, and those that only allow inlining of fixed targets thus preserving all virtual and interface dispatches. In both cases, these baseline code preparations precede the application of any specialization techniques.

Overall, two dimensions are explored in the comparisons of performance. The scope of the TEF optimization is widened from a purely local application to first include class context information, and then instance context information as part of the instance-specific specialization process. The latter two forms are also measured using only `final` fields to identify candidate contexts, and then using the

additional effective-final fields identified in the inspection of the benchmark class implementations. Some statistics regarding the number of actual specializations created and installed, and the diversity of subject types and methods observed are also collected and summarized in table 5.1.

5.3.1 Experimental Framework

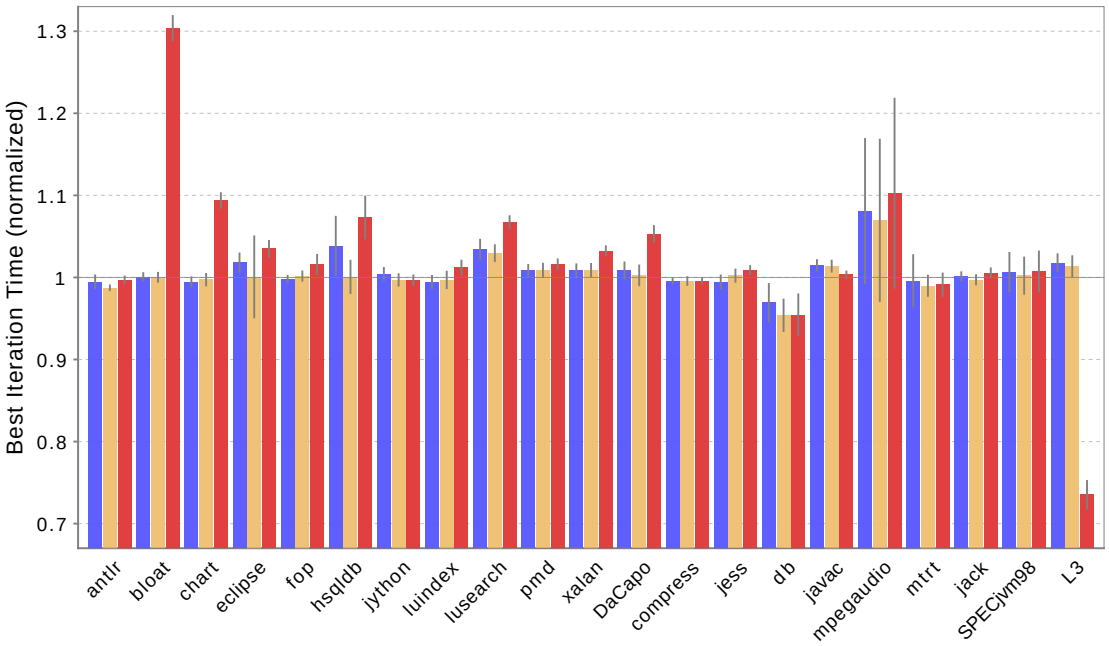
Similar to the intraprocedural RPE trials, Jikes RVM's "method sample size" parameter was increased for the specialization trials from the default value of 3 to 256. Again, this modification extends the sampling window of the adaptive optimization system to collect more data before selecting which methods to recompile at the next optimization level. This change serves to stabilize the optimization process somewhat, ensuring that the most active methods are optimized first, and thus further reducing the chances that a specialized dispatch will be bound to a frequently used, yet under-optimized implementation.

As mentioned previously, it was also necessary to alter the adaptive updating process so that it does not discard references to old code and updates to unspecialized methods are propagated to all specialized duplicates of a TIB. As before, the on-stack replacement mechanism is also disabled since, in this case, it is unable to distinguish between different specializations of a method, and hence causes invalid recompilation events.

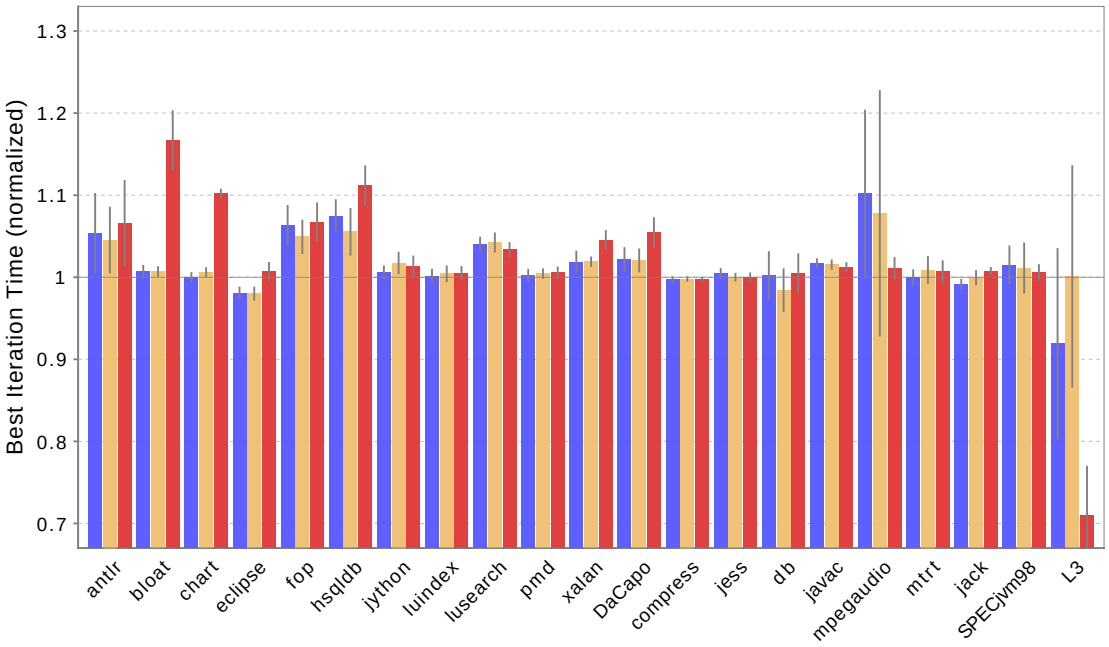
5.3.2 Results

Figures 5.6, 5.7, and 5.8 illustrate the performance results of the specialization trials. Once again, the measurements compare the mean-best iteration time obtained over a minimum of 10 benchmark executions, as outlined in §3.2.1. For each program, the scores are normalized relative to either the "Static Inline" or "Method Test" configurations, with the run-time modifications noted above. The raw timing scores for all of the contrasted configurations are given tables A.5 and A.6. The vertical lines extending from the bars represent the 95% confidence interval for each measurement sample. The L3 program, which stresses instance-specific polymorphism couplings, is listed on the far right of each chart.

The first set of comparisons given in figure 5.6 show the yield of applying TEF with no supplementary context information relative to runs that include class context information (CS), and those that perform instance-specific specialization (IS). For these trials, only uses of `final` fields are optimized. With no guarded inlining, only db shows a significant improvement among the standard benchmarks. With guarded inlining enabled, the results for eclipse are uncharacteristically stable, and show a significant improvement when applying TEF. For the most part, the local TEF and CS configurations yield statistically indistinguishable results, which is to be expected given that, as indicated in table 5.1(a), no class fields are identified when only `final` fields are considered. A follow-up examination based on these unexpected results revealed that an earlier phase of Jikes RVM's optimizing compiler actually substitutes `final static` values into the IR code as constants, thus allowing TEF to discover the field types without reverting to an inspection of the class data. The notable outcome of these trials is that, for most of the benchmarks, instance specialization (IS) incurs a sig-



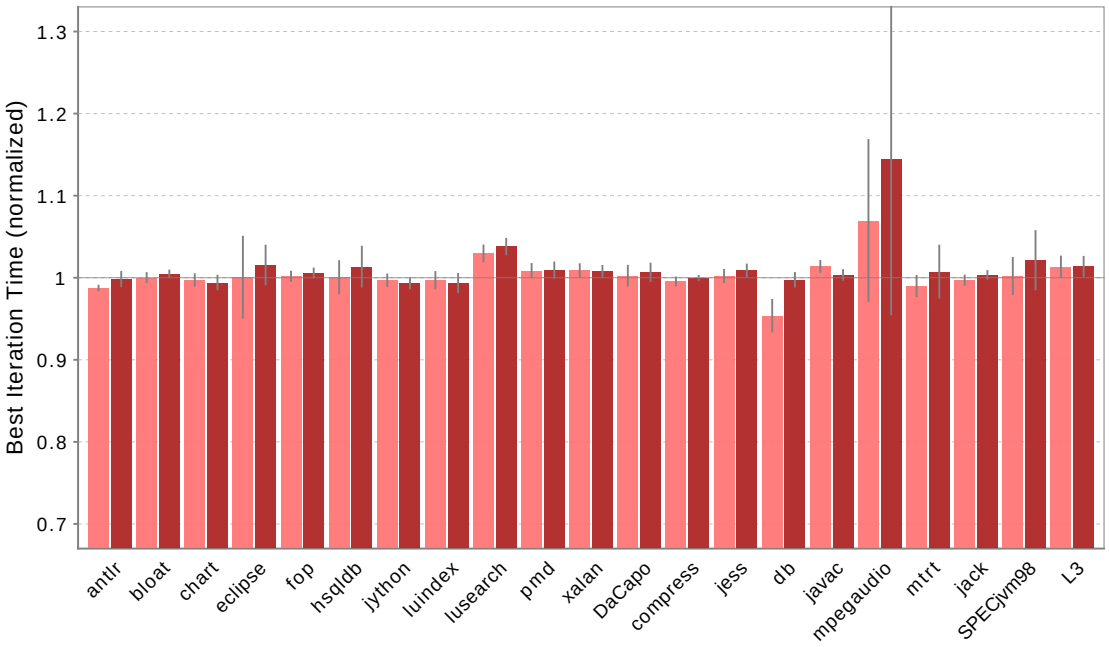
(a) Guarded Inlining Disabled



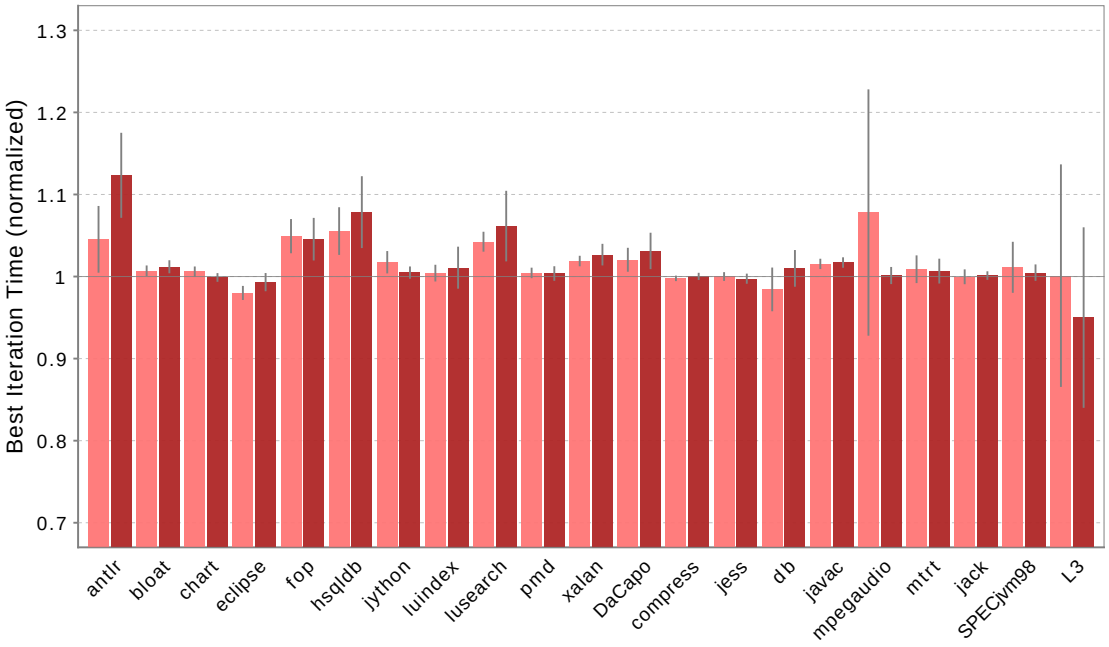
(b) Guarded Inlining Enabled

RPE:TEF	–	*Local (Intraprocedural) Type Expression Folding*
RPE:CS	–	*Specialization on* `final` *class fields*
RPE:IS	–	*Specialization on* `final` *instance and class fields*
Baseline (a)	–	*Static inline only: preserve virtual dispatch mechanisms*
Baseline (b)	–	*Allow guarded inlining using method tests and class tests*

FIGURE 5.6: **Instance RPE Performance**



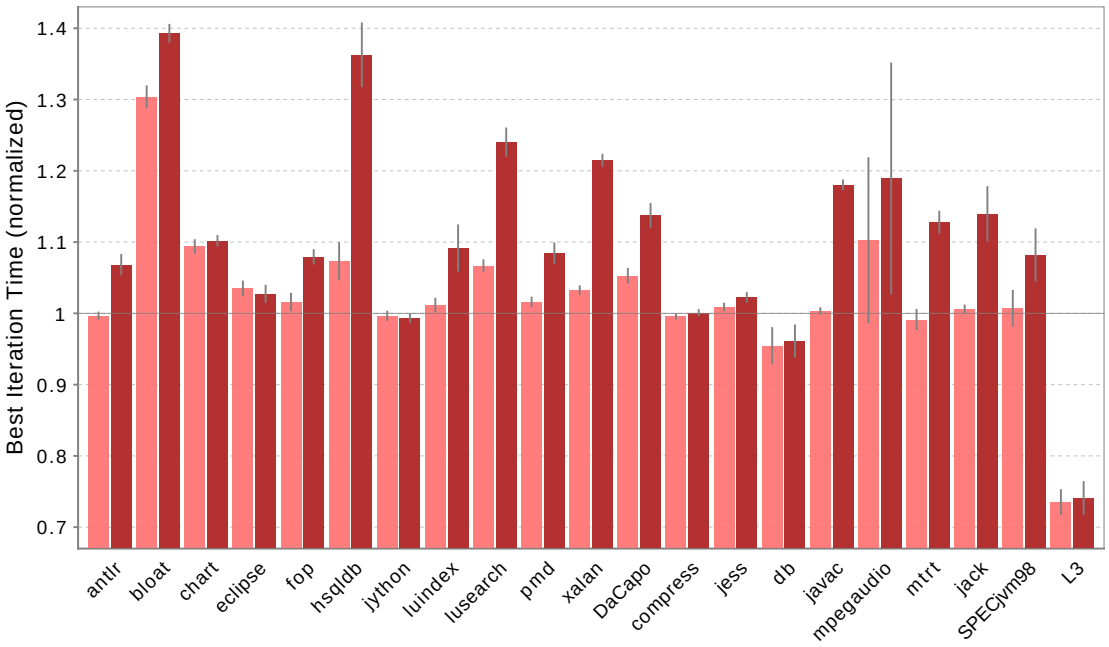
(a) Guarded Inlining Disabled



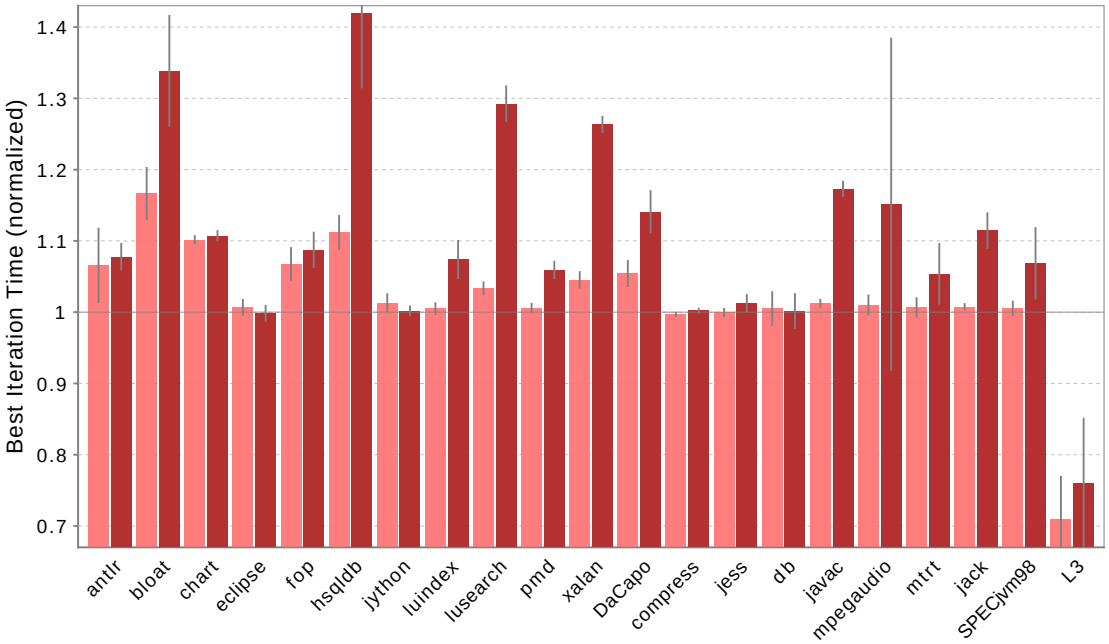
(b) Guarded Inlining Enabled

RPE:CS	–	*Specialization only on `final` class fields*
RPE:CS,EFF	–	*Specialization on all effective-final class fields*
Baseline (a)	–	*Static inline only: preserve virtual dispatch mechanisms*
Baseline (b)	–	*Allow guarded inlining using method tests and class tests*

FIGURE 5.7: **Performance of RPE Class Specialization with Effective Final Fields**



(a) Guarded Inlining Disabled



(b) Guarded Inlining Enabled

RPE:IS	–	*Specialization only on `final` class and instance fields*
RPE:IS,EFF	–	*Specialization on all effective-final class and instance fields*
Baseline (a)	–	*Static inline only: preserve virtual dispatch mechanisms*
Baseline (b)	–	*Allow guarded inlining using method tests and class tests*

FIGURE 5.8: **Performance of RPE Instance Specialization with Effective Final Fields**

(a) `final` Fields Only

	Class		*Instance*					
Benchmark	*Contexts*	*Methods*	*Contexts*	*Contexts/Type*	*Methods*	*Installs*	*Code Size*	
antlr	0	0	2.3	1 : 1 : 1	2.3	24670.3	+15.9%	*
bloat	0	0	54.0	1.8 : 1 : 25	347.7	79929481.0	+ 5.4%	
chart	0	0	1.7	1.2 : 1.2 : 2	2.0	22323183.7	+ 0.5%	
eclipse	0	0	97.3	1.3 : 1 : 18	355.3	3028594.3	+ 0.6%	
fop	0	0	6.7	1 : 1 : 1	9.7	316.7	−44.2%	*
hsqldb	0	0	2.7	1.5 : 1.5 : 4	5.3	8603.3	−19.8%	*
jython	0	0	1.3	1 : 1 : 1	1.3	252141.3	− 2.0%	
luindex	0	0	8.3	1 : 1 : 1	11.7	313877.7	+ 4.8%	
lusearch	0	0	3.3	1 : 1 : 1	3.7	3109841.3	+19.9%	*
pmd	0	0	1.3	1.5 : 1.5 : 2	2.0	389686.7	+ 5.3%	*
xalan	0	0	22.0	1.2 : 1 : 6	45.0	1455456.7	− 3.9%	*
compress	0	0	0	0 : 0 : 0	0	0	− 1.1%	
jess	0	0	0	0 : 0 : 0	0	0	+ 5.3%	
db	0	0	0	0 : 0 : 0	0	0	+ 5.1%	
javac	0	0	0	0 : 0 : 0	0	0	+ 2.6%	
mpegaudio	0	0	0	0 : 0 : 0	0	0	+ 1.1%	
mtrt	0	0	0	0 : 0 : 0	0	0	+ 3.9%	
jack	0	0	2.3	1 : 1 : 1	2.3	11695.7	+ 4.1%	
L3	0	0	3	6 : 6 : 6	72	92.1	− 5.7%	

(b) With Effective-Final Fields

	Class		*Instance*					
Benchmark	*Contexts*	*Methods*	*Contexts*	*Contexts/Type*	*Methods*	*Installs*	*Code Size*	
antlr	1	4.0	34.0	1.7 : 1.2 : 3	111.4	3023490.6	+ 24.7%	*
bloat	1	1.7	85.3	2.5 : 1.5 : 78	620.0	76461603.3	− 10.8%	*
chart	1	1	7.3	1.3 : 1.8 : 6	14.7	22012143.0	+ 22.9%	*
eclipse	0	0	99.0	1.3 : 1.3 : 18	335.3	3169576.7	+ 3.0%	*
fop	0.6	0.6	20.3	1.4 : 1 : 6	69.3	84822615.0	− 30.6%	*
hsqldb	6	13.0	23.0	1.1 : 1 : 2	71.3	12298858.3	− 8.7%	
jython	0	0	1.6	1 : 1 : 1	1.6	252143.0	− 3.8%	
luindex	2	2	28.0	1.1 : 1 : 2	69.3	2684787.0	+ 5.5%	
lusearch	0	0	24.8	1 : 1 : 1	44.8	55732162.4	− 2.8%	
pmd	2	2	94.0	5.1 : 1 : 9	218.0	9257321.7	+ 17.8%	*
xalan	9.2	25.6	44.8	1.1 : 1 : 6	105.0	42946961.2	− 2.6%	
compress	0	0	2	1 : 1 : 1	3	5886.3	+348.4%	*
jess	2	9	9.6	1 : 1 : 1	26.0	30641.3	+ 51.2%	*
db	0	0	2	1 : 1 : 1	4	24870.7	+161.3%	*
javac	2.7	11.0	26.7	1.7 : 1.2 : 7	77.3	22716112.3	+ 3.4%	
mpegaudio	0	0	11.0	1 : 1 : 1	61.0	6287614.0	+ 2.4%	
mtrt	0	0	7.6	1.8 : 1 : 6	6.7	1930623.3	+ 23.3%	*
jack	1	2	7.6	1.7 : 1.1 : 2	52.3	7091243.3	+ 14.9%	*
L3	0	0	3	6 : 6 : 6	72	92.2	+122.6%	*

Contexts – *Total number of class or object instance signatures specialized*
Contexts/Type – *Diversity among instances of same type (mean : median : max)*
Methods – *Total number of specialized methods*
Installs – *Installations of specialized TIBs*
Code Size – *Change in output machine code over the baseline configuration*
** indicates statistically significant change*

TABLE 5.1: **Observed Specializations; Guarded Inlining Enabled**

nificant additional overhead, with the distinct exception of L3, where instance specialization offers a dramatic 27% improvement over static inlining and a 29% gain over guarded inlining. These results indicate that there are perhaps some engineering improvements to be made in the specialization process, but that valuable improvements can be obtained under the right circumstances.

Figures 5.7 and 5.8 illustrate the effects of expanding the set of candidate fields to include effective-final values. The first set of comparisons examine just class specialization. Little change is noticeable for most programs which, again, is unsurprising given the limited number of observed specializations seen in table 5.1(b). However, for the programs that do show a change, it is often a degradation. Since the CS configuration does not incur any run-time overhead and does not generate multiple code specializations, this is unlikely to be the result of poor caching performance. One possible explanation is that binding to out-of-date target implementations may be occurring and reducing the maximum yield. The contrast of including effective-final instance fields is more dramatic in figure 5.8, which indicates that expanding the set of specialization candidates uniformly results in a reduction in performance. Combining these results with the strong positive improvement seen for the L3 program suggests that further refinements are needed to make the specialization process more selective of which instances are targeted.

Table 5.1 presents a summary of the actual opportunities found by the type analysis and specialization procedures. Entries with a decimal place indicate an average of distinct counts observed over multiple executions, whereas simple integer values indicate a consistently reoccurring result. In this case the data are taken from runs that include guarded inlining. Some items worth noting among these results are, the fact that all 72 of the possible method specializations for L3 are identified and installed (see §C.2), the complete lack of specialization opportunities in the SPECJVM98 programs when only considering `final` fields, and the significant diversity of results observed for bloat, eclipse, pmd, and xalan. At the very least, these latter measurements reveal the presence of the targeted instance-specific polymorphic couplings described in §5.1. Also unusual is the significant drop in optimized code produced for fop, suggesting that, perhaps, the specialization process is impeding the identification of code that would normally be promoted by the adaptive optimization system. Again, the increase in code size for compress, db, and L3 is actually quite small in absolute terms.

5.3.3 Commentary

The statistics in table 5.1 indicate that applications of the IS transform to the bloat benchmark exhibit a broad diversity of polymorphic couplings, specialize many methods, and install specializations on a vast number of objects. Yet the performance results for bloat are among the worst seen. On the other hand, even more specializations are installed in runs of the fop program, but this does not produce a similarly exaggerated slow down. A closer examination of the particular types specialized by these two programs reveals that those selected in bloat are often very short-lived—many could be classified as expressions of the FUNCTION OBJECT pattern—whereas those employed by fop typically persist longer and are more likely to be reused. This behavior is corroborated by a more careful inspection of the distributions of repeated subject uses shown in figures B.2 and B.4. In

the case of bloat, there is significant global reuse, but instance-specific reuses are limited to shorter run-lengths. In contrast, fop shows some strong instance coupling over longer reuse runs. Similar reuse behavior is also notable in the distributions for antlr (figure B.1) and luindex (figure B.7), and the performance results for these programs are similar to those of fop. Together, these results, along with the large improvements seen for the long-lived specializations in L3, suggest that the yield of specialization does progressively increase when applied to longer-lived instances.

Although the predictions offered in §3.3.3 were mostly unsuccessful in anticipating the performance gains of instance specialization, it is worth noting that many of the object couplings anticipated in §5.1.1 were, in fact, identified as specialization targets. In particular, a large percentage of the observed wrapper types represent adapted views of collections or other aggregates which are implemented through the use of Java's inner class mechanism.

5.4 Related Work

The term "specialization" has, over the years, been applied to a wide range of program optimizations. In general, a specialization incorporates information about the program—such as insights regarding the semantics of the source language, measurements of run-time behavior, or programmer supplied annotations—to create a more streamlined implementation which can only be used when certain conditions are satisfied. One of the oldest variants is *customization*, which appeared in an early implementation of the Self runtime [Chambers et al., 1989]. The approach was later generalized and described as *specialization* by Dean et al. [1995]. Their particular version used information about parameter values at the call site to identify and select a specialized target for the call. Tip and Sweeney [1997] were among the first to consider instance-specific specializations; their approach eliminated unused fields from certain objects in C++.

The technique that is most similar to the approach presented in this chapter is the "hierarchy mutation" scheme of Su and Lipasti [2006]. They also derive specialized code implementations for particular instances, however they focus on optimizations facilitated by knowing the values (not the types) of certain, mostly primitive, fields. Since the number of potential field values, and multiple value combinations, is far greater than the number of field type combinations, this approach is also more vulnerable to excessive code growth. Cheadle et al. [2008] similarly install instance-specific TIB references, but their focus is on tracking and modifying object behaviors, rather than optimizing execution. Other approaches which share some elements with the approach given here include, the intraprocedural specialization outlined by Suganuma et al. [2001] which also attempts to reduce type tests, the self-specializing Java interpreter presented by Shankar et al. [2005] which also inspects heap data, and the *metamorphic programming* technique described by Maurer [2004] in which objects are programmed to dynamically change their own implementation.

Since it is often challenging to infer the specializations that might be appropriate for a given program, many have proposed techniques for extending the syntax of programming languages, or providing tailored annotations to direct the generation of specializations. Examples for Java-like languages include: Volanschi et al. [1997]; Tourwé and De Meute [1999]; Schultz et al. [2003]; and

Liu et al. [2005]. Programmer-directed approaches that are specifically focused on specializations appropriate for design patterns include those of Friedrich et al. [2000] and Marquez [2004].

In their presentation of method test guarded inlining, Detlefs and Agesen [1999] mention an analysis for identifying "immutable fields" in support of eliminating guards. Briefly, the idea provides a transitive extension of the preexistence concept to consider invokes applied to fields that are unchanging—if the holder of the field was created when only one target was available for the invoke, then this must also be true of the field which must preexist the holder. Of course, the elimination of such guards is still a site-oriented technique, since it is applied to all contexts which use such fields. Unkel and Lam [2008] present a whole-program analysis for finding such unchanging fields, which they term *stationary*. Rogers et al. [2008] provide a method for extending the set of semantically stationary fields through the application of annotations and constraints that are enforced by the run-time environment.

5.5 Summary

Specialization is a widely studied and applied optimization technique, but few have considered how to align the idea with the notion that individual objects should be responsible for refining their own behaviors. The preceding sections have presented a novel approach in this regard by describing a type-oriented approach to instance-specific specialization for eliminating polymorphism queries. What differentiates the approach from previous similar techniques is the focus on optimizations enabled by localized type information, rather than those based on particular value combinations which may be much more numerous and prone to producing an unmanageable number of specializations. In contrast to the intraprocedural approach to redundant query elimination, the instance-specific technique avoids the need for additional variables by directly evaluating polymorphism queries for a given context and embedding the results directly into specialized method implementations. The technique is demonstrated as being particularly applicable to long-lived objects that maintain unchanging references to polymorphic delegates, as exemplified by the Decorator, Singleton, and other commonly used design patterns.

Performance measurements of the proposed technique indicate some significant improvements in reducing applications to `static` values, but the current realization of the instance-specific transform appears to be more costly than beneficial, at least for the DaCapo and SPECjvm98 benchmarks. However, the validity of the approach, under the right circumstances, is underscored by the very large speed up observed in applications to the `L3` program. In combination, these results along with an examination of specific benchmark behaviors indicate that the approach could yield better general performance if the specialization process were refined to selectively focus on longer-lived objects.

5.5.1 Contributions

Unlike other approaches to specialization which attempt to bind to a particular method or instance optimization at the site of the call or instantiation, the mechanism outlined in §5.2.1 provides a novel technique for effectively enhancing objects with the ability to self-specialize on demand.

Moreover, specializations created for one instance may later be reused by other similar instances, thus amortizing the cost of compilation. While the presented version is specifically focused on the elimination of polymorphism operations, the concept could easily be generalized and adapted to other forms of value-oriented specialization.

In the evaluation of their stationary fields analysis, Unkel and Lam [2008] present summary results which indicate that nearly half of all the reference fields in a wide range of Java benchmarks (including the SPECjvm98 suite) can be treated as invariant—many more than are declared as being `final`. The identification and exploitation of similarly defined "effective-final" fields, as described in §5.2.1 and summarized in table 5.1(b), provides additional empirical support for these conclusions, and in particular, demonstrates that such fields do appear in high-use objects since their implementations are targeted by Jikes RVM's adaptive optimization system and identified by the specialization process.

5.5.2 Directions

Module Specialization

Applications of the specialization process to class fields did prove promising, showing a statistically significant 2% improvement for eclipse with guarded inlining. This suggests that it may be worthwhile investigating the effects on other programs that make more frequent use of module abstractions in their designs. Specifically, it would be interesting to see the effects on programs that include significant use of aspect-oriented extensions implemented as class-based modules.

Selective Specialization

Another direction which seems worth exploring is the possibility of developing heuristics, based on object-lifetime demographics [Harris, 2000; Marion et al., 2007], for identifying long-lived objects to improve the selection of candidates which are likely to yield a net benefit from instance-specific specialization.

On the other hand, one simple way to prevent premature specialization would be to embed an invocation counter in every candidate instance. The specialization hook could then increment and test this counter, only proceeding to inspect the target instance and install a specialization once the counter has reached a certain threshold. This approach bears an interesting similarity to the action of a polymorphic inline cache (see §2.4.2) in that it monitors the behavior at a particular site and transforms the implementation after a certain number of applications. The difference, in this case, is that the change (switching to a specialized TIB) affects the entire implementation of an object, not just a single dispatch site.

Instance-Specific Guarded Specialization

It is also possible to conceive of an adaptation of the guarded inlining technique to instance-specific specialization. The statistics gathered in table 5.1 indicate that, in most cases, the coupling

of wrapper and delegate objects only results in one set of inner/outer types. Thus, similar to the premise of guarded inlining, there is an expectation of monomorphism—in this case, monomorphism of specializations. Based on this observation, it may be worthwhile investigating how a series of precise subject and field type tests could be inserted at the end of a constructor, and when matched lead to the installation of the expected instance specialization. Again, such an approach may offer significantly more value than the classic site-oriented technique since a short sequence of branching tests at the time of construction could be used to circumvent the overhead incurred by many subsequent polymorphism queries.

Chapter 6
Conclusions

The dominant idiom in object-oriented programming is to structure the logic of a program in terms of client-server relations: objects act as both clients of other objects, and as service providers. Moreover, it is a central tenet of modern object-oriented design that, whenever possible, such relations should be built in terms of opaque types, anticipating that individual instances will take on a variety of behaviors—that is, the relationships between objects ought to be treated as polymorphic in order to facilitate the reuse and future adaptation of program components. Indeed, the utility of this is principle is both validated and underscored by the wide-spread adoption of object-oriented design patterns which provide concrete, yet flexible strategies for constructing polymorphic object architectures.

While this approach is certainly advantageous from a software design perspective, in practice it imposes a significant additional execution penalty to resolve the exact behaviors of polymorphic relations. However, many have observed that the results of such resolutions are often highly redundant, and this insight has inspired a wide range of strategies for transforming object-oriented code to streamline the mechanisms used to implement dynamic inclusion polymorphism, virtual dispatching in particular. To date though, the primary emphasis of most polymorphism optimizations has been on exploiting observed or presumed skews in the distribution of targets resolved at the site of individual dispatch or type test operations. And while some approaches may incorporate run-time measurements, the ultimate objective of these transformations has been to effect a static reduction which is applicable to all input values. In contrast, the goal of this thesis has been to present a different perspective rooted in the observation that multiple applications of polymorphism to the same subject value must produce information that is reusable, thus permitting the reduction of subsequent query computations even if the first in a series of uses cannot be eliminated.

6.1 Summary of Contributions

The two main techniques proposed in this thesis have been shaped by an analysis of the common characteristics of intentional applications of polymorphism as seen in object-oriented design patterns. In some respects, this strategy can be seen as extending the ideas of Ball and Larus [1993] who aimed to classify the behavior of branches based on properties of the coding contexts in which they appear. Similarly, polymorphism queries represent complex conditional tests which, like branches, determine the path of execution, but are also often anticipatable under the right circumstances.

Looking back, the initial discussion of chapter 2 presented a concise history of the role of polymorphism in software design and shed new light on the ways in which the common, often myopic, understanding of polymorphism has influenced implementation and optimization techniques. An analysis of the uses of polymorphism in modern, mature object-oriented design led to the formulation of a new general strategy for reducing the overhead of polymorphic queries focused on repeated uses of the same query subject. This new approach was dubbed a *subject-oriented* technique, and contrasted against previous *site-oriented* schemes which fail to tie together results from several uses of the same value. In shifting the focus to consider reuse rather than total elimination or reduction of computations, it was noted that this reuse must occur relative to a certain context. Four contexts appropriate to the object-oriented programming model were identified, and an initial outline of strategies for performing a reuse optimization over each were explored. As a byproduct of relating reuse scenarios from various design patterns to these contexts, a new method of categorizing patterns based on their dynamic interactions was discovered.

After establishing this conceptual foundation, the presentation in chapter 3 proceeded to examine the practical aspects of actually implementing and evaluating the proposed techniques in an industrial-grade environment. The choice to use Jikes RVM as the main compilation and execution platform was justified, and the architecture and behavior of its adaptive optimization system was described and analyzed. Specific attention was paid to the observed instability in execution performance, and these insights were used to establish a new, rigorous methodology for measuring the effect of changes to the optimizing compilation strategy. Several analyses of the standard SPECjvm98 and DaCapo benchmark programs were also given, and used to reaffirm the widely held view that the SPECjvm98 programs show a distinct lack of diversity in measures such as code size, code complexity, code maturity, and polymorphic complexity. To develop the latter characterization, a set of detailed profiles measuring the outcomes of polymorphism operations were gathered and processed using a variety of redundancy models. These results were synthesized to formulate a number of predictions regarding the potential success of the proposed optimization techniques on the specific benchmarks used. It was noted that this particular anticipatory step is largely unprecedented in research regarding redundancy elimination.

Chapter 4 presented the first of the two major optimization techniques, in this case focusing on the elimination of intraprocedurally redundant polymorphism queries. The motivating discussion revealed the redundancies inherent in uses of polymorphism as seen in a variety of imperative idioms for manipulating objects as state machine-like entities; examples included applications of the Builder, Strategy, and State design patterns, as well as common input/output abstractions. The structure of uses in applications of the Iterator pattern were emphasized in particular to demonstrate the need for a *speculative* code motion strategy. The limitations of existing code motion strategies were summarized, and in response an entirely new selective, speculative redundancy elimination algorithm (SSPRE) was developed, and its application to the problem of redundant query elimination was described. To support the safe relocation of queries against potentially `null` subjects, an on-demand guarding scheme was also developed and described. The approach was

demonstrated to correctly preserve the semantics of potentially excepting instructions where other approaches may not.

Chapter 5 presented a completely different approach to eliminating repeated polymorphism queries relative to the span of individual object lifetimes. In this case, the motivating discussion examined commonalities across various forms of statically polymorphic, yet dynamically fixed object couplings. Patterns such as DECORATOR, ADAPTER, and BRIDGE were investigated, and their common realization through Java's inner class mechanism was described. Polymorphic delegation as seen in the TEMPLATE METHOD and SINGLETON patterns was also shown to have an effectively similar structure. To eliminate the redundancy inherent in regular applications of polymorphism by wrappers on their delegates, an instance-specific optimization procedure was develop and described. The essence of the approach is to introduce a mechanism whereby individual instances are imbued with the ability to inspect the specific types of their delegate fields and switch their own virtual dispatch table to use specialized method implementations where the results of polymorphism operations have been pre-computed. This emphasis on specializations derived from type rather than value information provides a guard against excessive code growth. In exploring ways to expand the set of specialization candidates, a coarse approximation of the set of *effective-final fields* was derived and employed successfully, confirming earlier work which indicates that many fields not declared as `final` can be effectively treated as if they were.

6.2 Support of Thesis

The discussion of §2.5.2 developed three central claims which the subsequent presentation has elaborated on, and ultimately validated. The identification of opportunities for reuse rather than total elimination of queries established the first claim that a subject-oriented approach to the optimization of polymorphism provided a new perspective not seen in previous formulations. The second and third claims asserted that this new view could be employed specifically to eliminate computations with respect to intraprocedural and instance-specific contexts.

In addressing redundancies among polymorphism queries over intraprocedural contexts, the presentation of chapter 4 demonstrated that such redundancies can, in fact, be identified and eliminated. Moreover, the use of a light-weight speculative approach was shown to be both viable and essential in optimizing queries protected by conditional branches, as in applications of ITERATOR and common code structures produced by guarded inlining. An evaluation of the technique demonstrated several statistically significant performance improvements over the more relevant DACAPO programs—gains which are notably impressive, given that polymorphism operations constitute a limited fraction of the total instruction stream, and exhibit strong, statically reducible monomorphic behavior in many cases. One particularly interesting aspect of the results is that some of the best improvements were achieved over the highly multithreaded programs hsqldb, luindex, and xalan. Also, the fact that the latter two programs are based on a well-developed open-source code base provides yet more evidence of the utility of the approach for realistic examples. At the same time, the results for the specially tailored L1 program prominently underscore that the approach is able to achieve substantial

improvements which are not obtainable using other state-of-the-art techniques.

In the case of the instance-specific treatment of redundancy, again, the evidence provided in chapter 5 demonstrates that the technique is both viable and successful in identifying and transforming the targeted patterns. And while the performance results over the standard benchmark programs are not as definitive as those from intraprocedural trials, there are indications that the technique could be refined to offer substantial benefits. The improvements seen for the `L3` program confirm this assertion, by demonstrating a full 29% gain over the best set of techniques available in Jikes RVM.

Together, the results of chapters 4 and 5 provide strong support for the validity of the subject-oriented perspective. Moreover, the gains observed over the guarded inlining baseline demonstrate that the subject-oriented techniques serve to complement rather than supplant existing site-oriented approaches. Furthermore, while the presentation has focused on the implementation of Java's polymorphism operations as realized in Jikes RVM, the concepts and described transformations may be easily adapted to other polymorphism data representations for languages with fixed type structures. Notably, the ideas are equally applicable to the semantics of similar operators in the .NET CIL format (`callvirt`, `ldvirtfn`, `isinst`, and `castclass`), as well as virtual dispatching and run-time type information (RTTI) queries in C++.

6.3 Commentary

For many years the true nature of design patterns has been debated. Do some represent abstractions that have yet to be promoted to formal language structures—*cadets*, as described by Gil and Lorenz [1998]? Or is the fact that they are not rigidly defined one of their essential characteristics, as suggested by Vlissides [in Chambers et al., 2000]? These are interesting questions with no simple answers. However, the results of this thesis contribute to the discussion by demonstrating that an optimization technique can be devised around both the formal semantics of a language feature—dynamic polymorphism in this case—and an understanding of its practical application in design patterns. In other words, effective optimization of pattern structures can be achieved regardless of whether they are formalized in the source language or not.

Also worth noting is the observation by Riehle [2009] that, over time, the prevalence of design patterns in a system increases as its code base matures. An obvious correlate of this development is that the utility of the optimizations proposed in this thesis, which have been demonstrated as applicable to almost all of the patterns outlined by the Gang of Four, will also be greater for such mature projects. Moreover, their utility will only increase as patterns gain wider adoption.

6.4 Directions

The theme of this thesis has been focused on a fairly simple observation that it is worthwhile considering the semantics of a sequences of uses applied to a value—a polymorphism subject in this case—rather than trying to effect independent reductions at each usage site. In addition to the

given optimizations, applying this perspective also suggests a number of ways in which some existing techniques relating to polymorphism could be enhanced.

6.4.1 Speculative Cast Elimination

Down-casting to gain access to certain functionality is a common, yet costly practice in object-oriented programming. In Jikes RVM, even though type checks are based on efficient bit-vector pattern matching, the process still entails four dependent loads and two tests to confirm the identity of a subject (see figure 4.1). Moreover, because such casts carry important semantics, they cannot be indiscriminately removed, even if they may seem unnecessary given a program's logic. However, the direct coupling of a cast which is expected to succeed followed immediately by a dispatch on the result presents an unique opportunity to relegate the full computation of the type check to the infrequent execution path. Figure 6.1 illustrates a simple example.

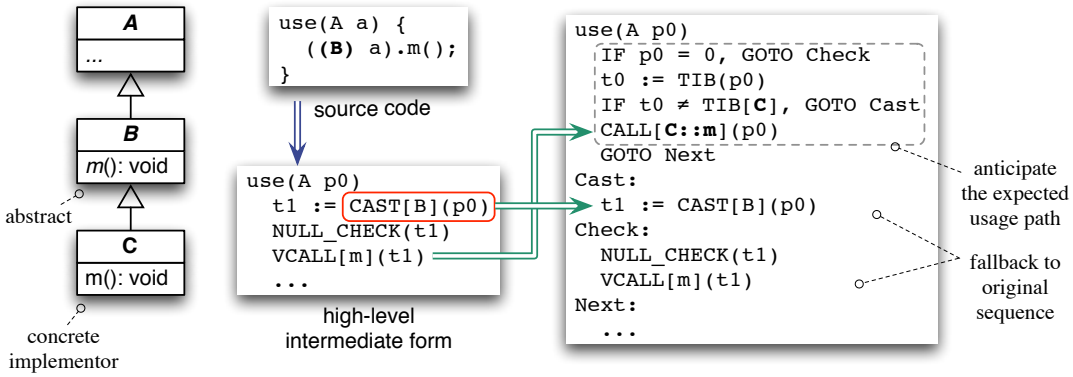


FIGURE 6.1: **Speculative Cast Elimination**

Here, a transform similar to a class test inline guard is used to anticipate the concrete type of the cast subject based on its expected form in the subsequent virtual invoke. If the guard passes, then this test, in addition to determining the call target, also asserts that the cast should succeed. If the guard fails, then the original sequence of heavy-weight queries are performed in the order necessary to preserve the original semantics. This approach is, essentially, a variation on the inline caching technique, but incorporates the subject-oriented perspective to fold several related computations (in this case, an additional cast) into the savings offered by the speculative guarding process. Furthermore, this technique is particularly suited to Jikes RVM since the AOS provides statistics from the running program that can be used to infer the most appropriate guard choice, rather than needing to rely on static heuristics to guess the best value. Also, some of the preexistence techniques proposed by Detlefs and Agesen [1999] may be further employed in combination with this insight to remove certain casts entirely.

For Java, the ability to reduce the costs of this particular scenario are significant since examples

similar to the one above are a common byproduct of the casts inserted to implement the semantics of Java's generic typing system.

6.4.2 Multi-Site Polymorphic Inline Caching

One of the interesting behaviors revealed by the study of redundancy in polymorphism queries presented in §3.3.2 is that, for many benchmarks, the last global result at a site is often more likely to be recomputed than the global best result. This suggests that an adaptive inline caching strategy may be an effective addition to Jikes RVM optimization strategy. Indeed, while the details are not publicly available, there are some indications that the just-in-time compiler included in Microsoft's .NET/CLR VM actually generates polymorphic inline caches for certain high-frequency dispatches.[1]

Building on the simple cast reduction scenario given previously, an obvious generalization of the concept is to realize that the result of one guarding test can be used to predicate several polymorphic operations. Thus rather than generate a separate PIC for each call site, the majority of the logic could be included in the first dispatch, and subsequent sites could simply reuse the guard result. This also reduces the need to maintain separate counters at each site—a transition to a different preferred target at the first site also implies a similar transition for other uses of the same subject.

6.4.3 Extended Dynamic Polymorphism

Another way in which the techniques of this thesis could be further developed is to consider their application to other extended forms of dynamic polymorphism. The classical categorizations of polymorphism outlined by Cardelli and Wegner [1985] suggest that inclusion within a family of statically defined types is the only significant form of dynamic polymorphism. However, over time, an understanding of the utility of late binding polymorphism has led to the development of several other forms. For example Ernst [2001] described the notion of *family polymorphism* where the particular values and relations of objects establish a polymorphic, computed subdivision of type instances. Similarly the notion of multiple subject dispatch (i.e., "multidispatch") is a form of polymorphism that is not defined by inclusion in any one specific type. A number of proposals for incorporating multidispatch into Java [e.g., Millstein, 2004; Cunei and Vitek, 2005] have also extended the idea to include forms of predicated dispatch. In these systems, conditions at the call site and/or properties of the receiver(s) determine exactly which polymorphic binding occurs.

Naturally, if these predicate tests or multi-subject queries remain invariant over the span of several uses of the same input values, then the reductions proposed in this thesis also apply to these even more complex selection predicates.

[1] Evidence of the use of inline caching appears in low-level code examples provided by the weblog post "JIT Optimizations, Inlining, and Interface Method Dispatching" at http://blogs.microsoft.co.il/blogs/sasha/archive/2007/02/27/JIT-Optimizations_2C00_-Inlining_2C00_-and-Interface-Method-Dispatching-_2800_Part-1-of-N_2900_.aspx

6.4.4 Polymorphic Tail Calls

Recursive tail calls—those that occur as the last step before the return of a subroutine—are a common idiom used to express iteration in the absence of an explicit looping construct, as is typical in functional programming languages. However, while often considered more "elegant" and flexible than imperative iteration, a significant limitation of recursion is that each step consumes stack space which may ultimately be exhausted, causing deep recursive progressions to fail where an iterative version would not. This possibility has motivated the development and application of *tail call optimization* (TCO) in the implementation of many functional programming languages. Briefly, the idea is that rather than expand the stack further, the implementation of a tail call is transformed to commandeer the stack space of the caller which has no further need of a local context since it is about to exit anyway following the tail call.

The utility of TCO is evident for functional programming languages, but there is also reason to treat it as relevant for object-oriented languages. Indeed, recursive tail calls are an essential tool in formulating object designs that strongly encapsulated to the point of being *autognositc* [Cook, 2009]. However, the challenge in developing TCO for object-oriented languages lies in the fact that tail calls are much more likely to be based on polymorphic dispatch—complicating the matter since multiple targets, requiring different stack adaptations, may exist.

Both of the major optimization perspectives discussed in this thesis (site-oriented and subject-oriented) would seem to show some potential for addressing this problem. On the one hand, if there is a presumption of monomorphism, then a straightforward adaptation of inline caching could be applied with appropriate stack manipulations preceding the direct transfer, but following the guarding test. In contrast, if the tail call is observed to be polymorphic, and is performed on a stationary field of the caller, then an instance-specific, specialized tail call procedure could be derived and installed on the caller.

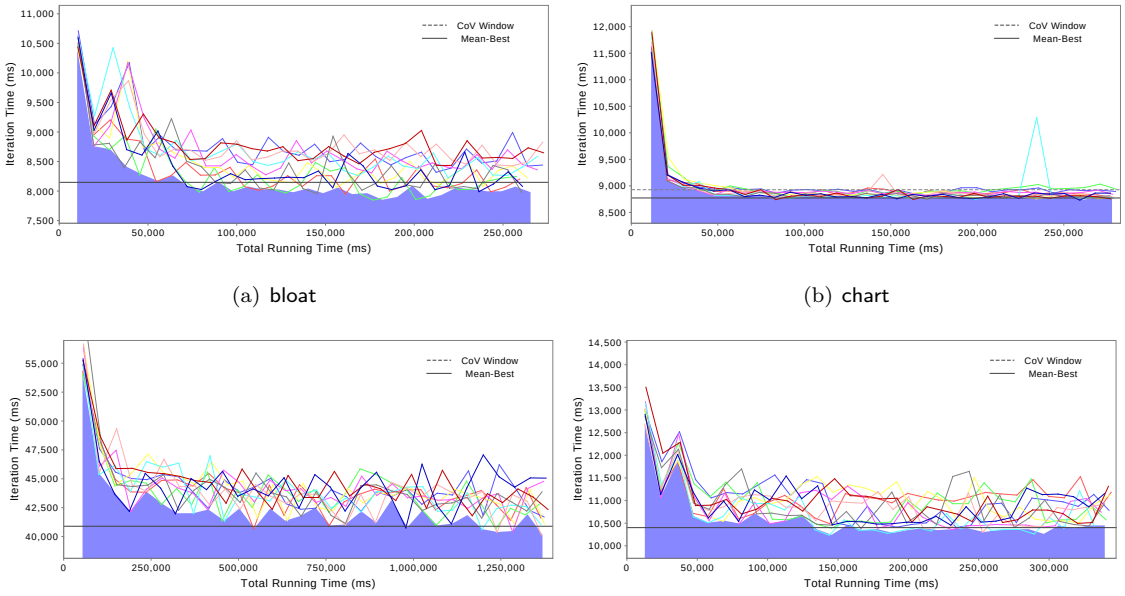
6.4.5 The Future of Polymorphism

Dynamic polymorphism is now a cornerstone of modern software development, and its application in object-oriented design, and design patterns in particular, is pervasive. Yet there are also indications that the conception of polymorphism as tool for supporting dynamic substitution rather than static reuse may continue to penetrate further into the fabric of software design and implementation. As more applications are developed with the aim that they will run continuously, polymorphism coupled with dynamic class loading provides a mechanism which allows such programs to be updated without having to be shut down. In terms of programming language development, recent efforts to incorporate composable trait-based object definitions [e.g., Odersky and Zenger, 2005], relational object queries (as in the LINQ extensions for C# [Hejlsberg et al., 2008]), and dynamically activated aspect-oriented advice [Herrmann et al., 2007] all rely on polymorphic typing and dispatching. Both of these trends indicate that the importance of improving the efficiency of polymorphism operations will only increase as both development patterns and language semantics make more extensive use of polymorphism to facilitate sophisticated dynamic binding and replacement mechanisms.

Appendix A
Trial Results

This appendix lists the raw numeric data and the remainder of graphical plots which form the basis of analyses of Jikes RVM performance given in §3.2, §4.3, and §5.3.

A.1 Performance Variation



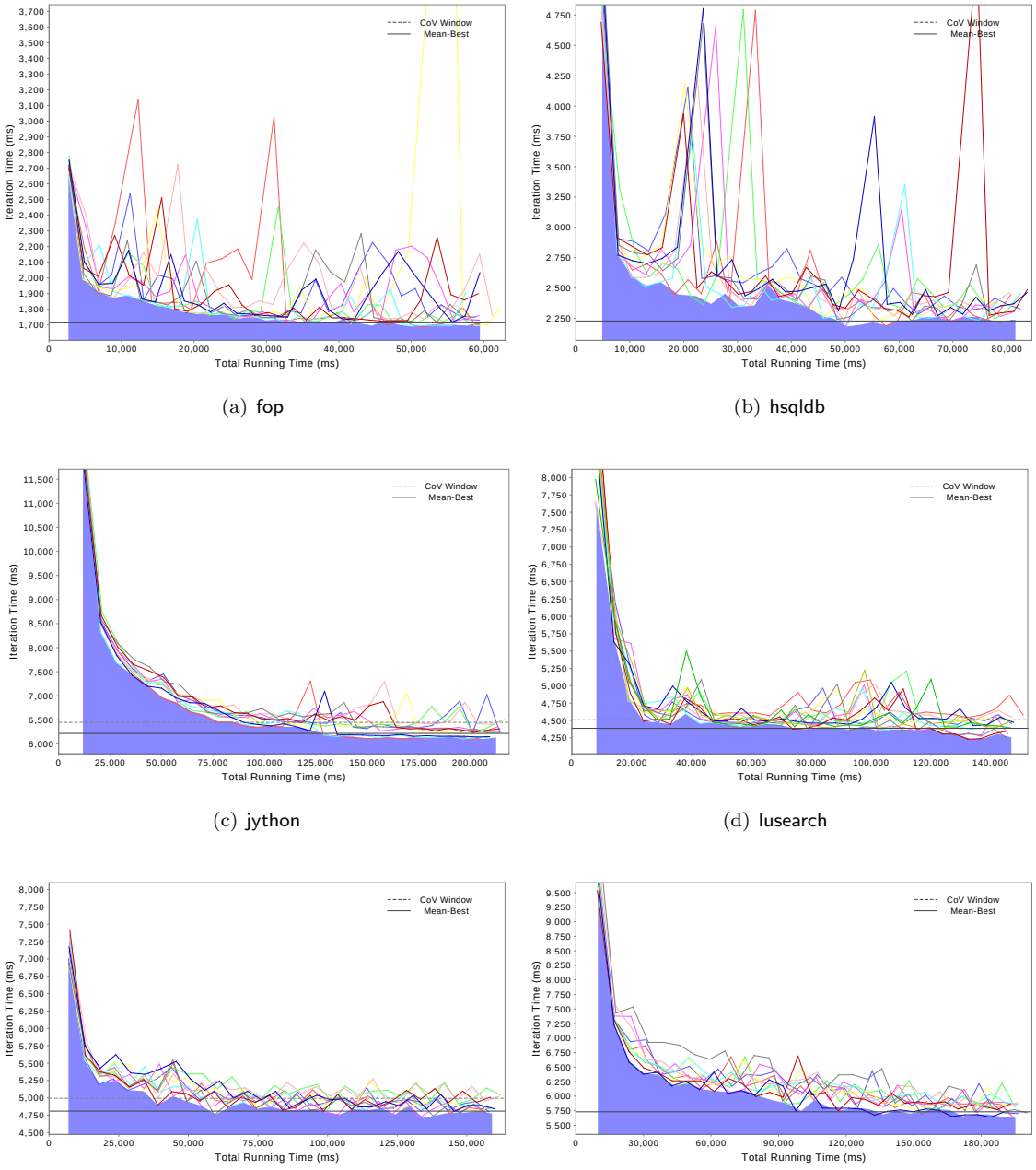
(a) bloat (b) chart

(c) eclipse (d) luindex

Figure A.1: **Performance Variation of DaCapo Benchmarks (part 1)**

As in §3.2, the variation in performance over individual benchmark iterations is depicted in figures A.1 and A.2 by plotting the interim scores (iteration times) of 10 different executions of each program from the DaCapo and SPECjvm98 suites.[1] Similar to figure 3.3 (which provides the plot for antlr), the blue shaded area represents the minimum score across all executions at the end of each iteration. Since this approach mixes data from several, independently evolving executions, the minimum scores are interpolated on the x-axis (total running time) using the accumulated mean iteration times. Again, the blue area represents the statistically inferred progression of maximum code potential produced by Jikes RVM's adaptive optimization system. The solid black line plotted across each graph represents the *mean-best* result, calculated according to the methodology outlined in §3.2.1. When the adaptive window method of Georges et al. [2007] successfully converges on a sequence of 10 iterations, for every execution, the mean result of this measure is also plotted as a

[1] The plots for L1 and L3 are presented as part of the explication of these programs in appendix C.



(a) fop

(b) hsqldb

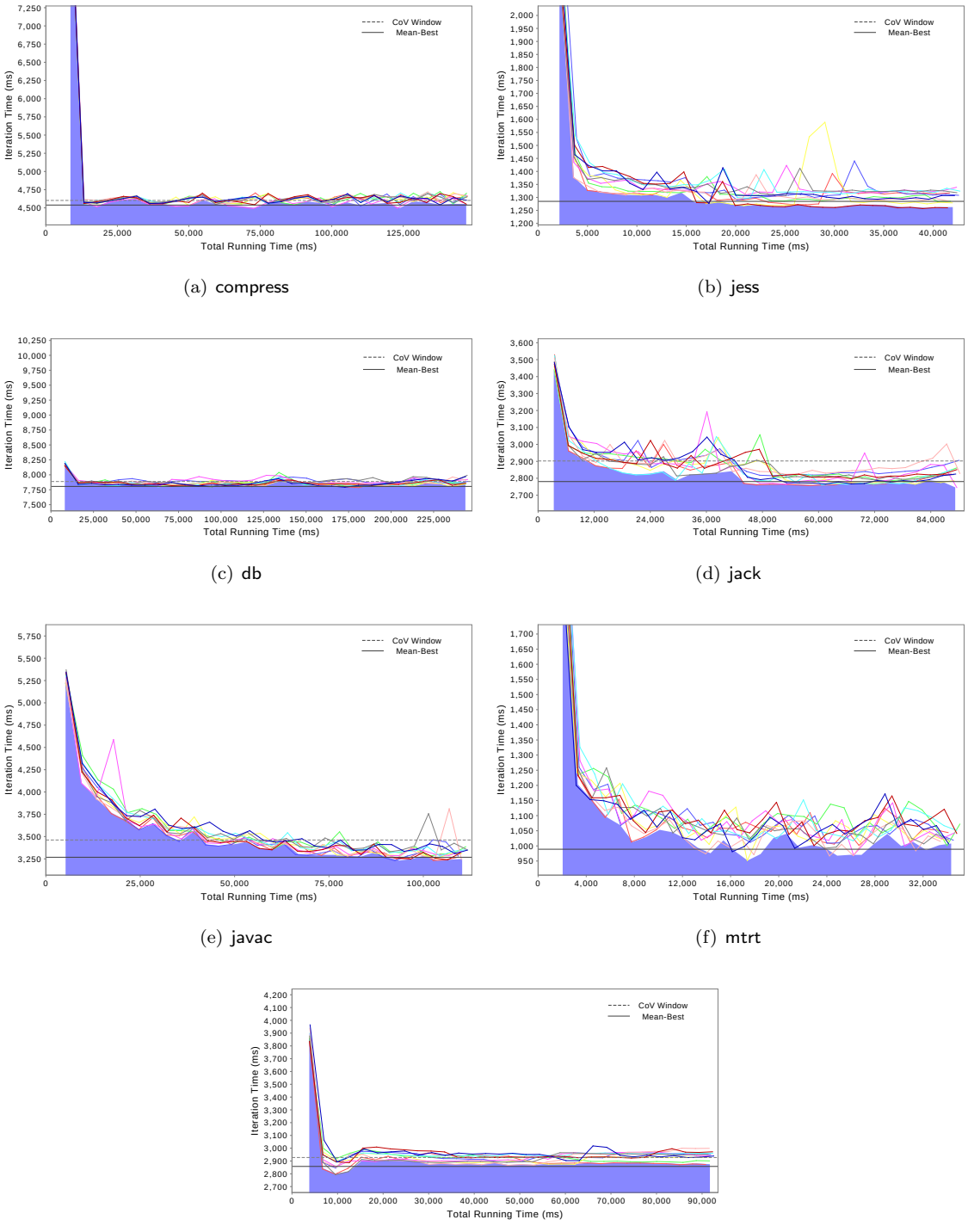
(c) jython

(d) lusearch

(e) pmd

(f) xalan

FIGURE A.1: **Performance of Variation DaCapo Benchmarks (continued)**



(a) compress

(b) jess

(c) db

(d) jack

(e) javac

(f) mtrt

(g) mpegaudio

FIGURE A.2: **Performance Variation of SPECjvm98 Benchmarks**

	Best Iteration Score - milliseconds (± 95% confidence)									
Benchmark	*Production*		*Method Test*		*Class Test*		*Static Inline*		*No Inline*	
antlr	2680.5	(21.2)	2746.6	(17.6)	3028.6	(13.5)	3147.0	(8.0)	3559.0	(12.3)
bloat	8443.5	(154.5)	8454.9	(96.8)	9036.4	(47.7)	9948.9	(57.1)	12497.5	(46.8)
chart	9212.1	(46.7)	9408.1	(48.2)	9672.3	(39.7)	10920.2	(79.7)	11719.9	(146.5)
eclipse	42180.9	(458.9)	42990.3	(565.7)	42448.2	(2570.3)	44801.5	(359.1)	49312.1	(1468.2)
fop	1792.5	(14.1)	1786.5	(23.6)	1835.1	(11.4)	1932.0	(12.2)	2036.9	(7.2)
hsqldb	2350.9	(47.2)	2467.9	(42.4)	2611.1	(36.4)	3149.8	(50.3)	3914.8	(86.4)
jython	6473.9	(47.7)	6551.0	(31.6)	6701.1	(49.6)	7143.1	(55.2)	9531.3	(37.7)
luindex	10882.8	(49.3)	10969.1	(98.9)	11441.7	(45.2)	11806.3	(63.5)	13324.0	(94.9)
lusearch	4620.2	(55.0)	4635.8	(37.4)	5212.9	(34.3)	5445.6	(26.8)	7101.7	(49.6)
pmd	4945.7	(44.4)	5059.0	(57.8)	6123.1	(58.0)	6528.4	(34.9)	8293.7	(168.7)
xalan	5855.6	(24.9)	6021.3	(32.9)	6201.7	(29.2)	7124.0	(35.4)	8409.2	(57.3)
compress	4538.5	(14.3)	4533.2	(10.2)	4536.5	(8.7)	4535.2	(10.0)	5873.2	(15.5)
jess	1333.3	(13.5)	1350.8	(8.8)	1342.5	(11.5)	1514.3	(5.9)	2293.9	(7.7)
db	7978.8	(113.1)	8136.7	(163.2)	8101.4	(166.8)	9527.7	(57.9)	9969.5	(16.6)
javac	3371.4	(12.2)	3394.0	(12.0)	3456.3	(13.3)	3685.4	(14.3)	4237.7	(25.5)
mpegaudio	2902.6	(18.8)	2927.6	(18.9)	2929.7	(11.9)	2971.1	(6.7)	3756.4	(21.4)
mtrt	1006.6	(12.2)	1268.9	(46.6)	1246.8	(21.9)	1701.6	(26.7)	2006.9	(32.8)
jack	2853.0	(17.2)	2854.0	(16.2)	2826.5	(11.4)	3086.9	(18.8)	3477.4	(10.9)
L1	2816.3	(48.2)	2823.7	(57.5)	2805.9	(45.3)	2839.5	(63.1)	2838.8	(13.1)
L3	5259.4	(646.6)	5137.9	(617.7)	5250.6	(678.4)	6794.5	(89.2)	6869.5	(63.4)
	Normalized Score - Harmonic Mean (± 95% confidence)									
DaCapo	1.0000 (0.0097)		1.0168 (0.0097)		1.0786 (0.0122)		1.1683 (0.0080)		1.3767 (0.0155)	
SPECjvm98	1.0000 (0.0080)		1.0376 (0.0130)		1.0353 (0.0093)		1.1426 (0.0076)		1.3876 (0.0089)	

Production – *Jikes RVM default: Speculatively inline using preexistence, patching, and guards*
Method Test – *Speculatively inline using method test and class test guards*
Class Test – *Speculatively inline using class test guards only*
Static Inline – *Only inline methods with fixed targets (e.g., static methods and constructors)*
No Inline – *Do not inline any methods*

TABLE A.1: **Jikes RVM Performance - Standard Inlining Configurations**

dashed line and labeled "CoV Window." The trials are executed using Jikes RVM's standard "production" configuration; summary results for these runs are given in table A.1.

Several observations are worth noting in the plots. First, significant variability in the progression of scores is apparent in bloat, eclipse, luindex, lusearch, xalan, jess, mtrt, and jack. In addition, fop and hsqldb exhibit extreme variations that exceed the minimum score by a factor of 2 or more. However, despite the common instability in execution behavior, an exponential decay in the statistical progression of minimum scores is apparent for all the programs. Moreover, the *mean-best* calculation clearly achieves a good approximation of the limit of this progression for all programs.

A.2 Baseline Results

The values given in table A.1 represent the mean-best performance results for the major inlining strategies employed by Jikes RVM. For each program, the scores are normalized against those from the first column to yield the plot given in figure 3.4. As described in §3.2.1, these values are collected by extracting and averaging the minimum iteration time from at least 10 different executions. In some cases, entries represent more than 10 executions because trials are performed in batches and

some of the multithreaded programs, such as hsqldb and lusearch, routinely cause faults Jikes RVM's garbage collection process, requiring more trials to be run. The suite aggregates are computed as the harmonic mean of normalized scores, thus giving improvements (or degradations) from each program equal weighting. When the difference between a baseline and comparative result exceeds the sum of their 95% confidence spans, the change is deemed statistically significant. For example, in comparison to the "Production" configuration, the average "Class Test" result over the DaCapo suite is weaker with statistical significance, whereas the result for the "Method Test" configuration is weaker, but not a statistically significant difference.

A.3 Intraprocedural RPE Results

Tables A.2 through A.4 provide the raw values used assess the intraprocedural redundant polymorphism elimination (RPE) technique described in chapter 4. Most of this data is plotted in the comparisons given by figures 4.9 through 4.13. Here, the first table (A.2) contrasts the improvements achieved by the two code motion strategies—busy code motion (BCM) and lazy code motion (LCM). The first four columns use the "Static Inline" configuration as a baseline for normalization, while the last three use the "Method Test" configuration with on-stack replacement disabled. Because of this latter adjustment, the Method Test values are not the same as those listed in table A.1. The confidence intervals for the suite average Static Inline results are also different since these values are now scaled to represent the baseline. Table A.3 gives a more extensive listing of results for various trials exploring speculation thresholds for the SSPRE transform. The CF and RF strategies are explained in §4.3.1. The MF strategy represents the same "maximum frequency" threshold scheme employed in the isothermal approach of Horspool et al. [2006]. Overall, these results confirm that, for the more relevant DaCapo programs, the RF=8 speculation threshold yields the best improvement. Table A.4 highlights the performance scores relative to reductions in compilation time when using a selective threshold to focus the SSPRE strategy. Again, the Method Test values are slightly different due to the overhead of collecting compilation statistics. The key outcome here is that the performance scores are relatively similar for various selective strategies, but using the CF=70% threshold, the compilation overhead is reduced by nearly 50% over the DaCapo programs, and nearly 70% for the SPECjvm98 suite.

A.4 Instance RPE Results

Tables A.5 and A.6 list the performance scores for the various configurations of instance-lifetime RPE evaluated in chapter 5. This data forms the basis for comparisons shown in figures 5.6, 5.7, and 5.8. Here, the results are partitioned based on whether guarded inlining was enabled or not. All of the trials listed in figure A.5 perform static inlining only, whereas those presented in figure A.6 permit method test and class test inlining. Once again, the values for the baseline configurations differ from those in earlier tables since, in this case, it was necessary to disable the retirement and garbage collection of old code which may still be targeted from specialized method implementations.

	Best Iteration Score - milliseconds (± 95% confidence)																	
Benchmark	*Static Inline*		*RPE:BCM*		*RPE:LCM*		*RPE:LCM,calls*		*Method Test*		*RPE:BCM,inline*		*RPE:LCM,inline*	
antlr	3147.0	(8.0)	3068.6	(10.5)	3069.1	(10.5)	3079.7	(8.1)	2709.4	(16.5)	2663.2	(9.9)	2673.3	(12.4)
bloat	9948.9	(57.1)	9826.6	(30.3)	9873.2	(27.5)	9896.2	(46.0)	8607.5	(61.6)	8617.6	(83.9)	8623.5	(67.6)
chart	10920.2	(79.7)	10884.2	(72.2)	10809.5	(40.9)	10891.5	(47.8)	9285.6	(30.5)	9280.0	(60.2)	9288.0	(29.0)
eclipse	44801.5	(359.1)	43913.2	(1930.8)	44505.4	(1191.9)	44161.8	(1549.5)	43045.7	(426.7)	42496.4	(489.3)	42314.9	(1577.3)
fop	1932.0	(12.2)	1926.5	(10.9)	1928.1	(6.0)	1931.2	(6.0)	1786.6	(5.7)	1784.2	(15.8)	1786.0	(8.4)
hsqldb	3149.8	(50.3)	3108.6	(26.9)	3038.2	(23.2)	3055.7	(26.7)	2446.3	(29.3)	2470.7	(24.4)	2435.5	(30.8)
jython	7143.1	(55.2)	7265.7	(47.5)	7195.4	(31.4)	7186.0	(26.0)	6511.6	(44.5)	6499.7	(41.9)	6488.9	(28.0)
luindex	11806.3	(63.5)	11636.3	(58.8)	11610.7	(55.4)	11654.8	(67.4)	10809.6	(46.7)	10892.7	(73.3)	10887.1	(42.4)
lusearch	5445.6	(26.8)	5410.8	(20.0)	5402.2	(24.7)	5420.8	(28.6)	4607.7	(43.1)	4592.0	(53.9)	4591.0	(26.5)
pmd	6528.4	(34.9)	6529.4	(40.1)	6495.6	(38.9)	6473.9	(31.5)	4998.9	(25.2)	4989.1	(23.3)	4989.2	(20.4)
xalan	7124.0	(35.4)	7037.3	(47.5)	7036.7	(31.2)	7015.1	(45.2)	5895.6	(50.7)	5731.6	(28.8)	5725.9	(28.3)
compress	4535.2	(10.0)	4527.4	(9.2)	4533.2	(18.9)	4525.8	(7.3)	4542.6	(6.9)	4530.4	(11.7)	4535.0	(7.9)
jess	1514.3	(5.9)	1534.1	(6.0)	1508.8	(6.0)	1513.2	(5.9)	1336.4	(7.2)	1325.0	(15.2)	1340.9	(6.1)
db	9527.7	(57.9)	9552.7	(160.4)	9594.5	(94.3)	9601.1	(89.9)	8361.8	(8.3)	8329.7	(92.0)	8309.8	(103.6)
javac	3685.4	(14.3)	3701.1	(13.1)	3690.3	(8.2)	3693.5	(6.9)	3397.3	(13.7)	3375.9	(6.8)	3391.7	(9.2)
mpegaudio	2971.1	(6.7)	2968.4	(8.9)	2971.2	(9.1)	2963.4	(10.4)	2912.3	(10.7)	2896.1	(10.3)	2899.9	(9.6)
mtrt	1701.6	(26.7)	1814.7	(31.0)	1703.8	(15.3)	1700.1	(18.0)	1038.0	(7.1)	1057.0	(12.0)	1038.5	(8.8)
jack	3086.9	(18.8)	3079.8	(15.5)	3083.1	(8.8)	3073.3	(8.1)	2819.5	(10.2)	2801.6	(19.7)	2802.9	(8.5)
L1	2839.5	(63.1)	2535.8	(19.1)	2454.3	(12.2)	2587.1	(14.8)	2825.7	(41.9)	2552.0	(49.8)	2477.2	(34.1)
	Normalized Score - Harmonic Mean (± 95% confidence)													
DaCapo	1.0000	(0.0067)	0.9915	(0.0089)	0.9889	(0.0064)	0.9900	(0.0076)	1.0000	(0.0069)	0.9955	(0.0077)	0.9941	(0.0084)
SPECjvm98	1.0000	(0.0057)	1.0111	(0.0075)	1.0006	(0.0050)	0.9999	(0.0048)	1.0000	(0.0037)	0.9978	(0.0070)	0.9977	(0.0052)

Static Inline	–	*Inline fixed targets only: preserve virtual dispatch mechanisms*
RPE:BCM	–	*Speculative intraprocedural RPE via busy code motion*
RPE:LCM	–	*Speculative intraprocedural RPE via lazy code motion*
RPE:LCM,calls	–	*Speculative LCM-RPE, ignoring type tests and casts*
Method Test	–	*Inline using method test and class test guards (no OSR)*
RPE:BCM,inline	–	*Speculative BCM-RPE, allow guarded inlining*
RPE:LCM,inline	–	*Speculative LCM-RPE, allow guarded inlining*

TABLE A.2: **Intraprocedural RPE Performance Results - Code Motion**

Benchmark	Best Iteration Score - milliseconds (± 95% confidence)											
	Static Inline		*LCM H:RF=8*		*LCM H:RF=16*		*LCM H:CF=1%*		*LCM H:MF=90%*		*LCM H:none*	
antlr	3147.0	(8.0)	3069.1	(10.5)	3056.4	(8.6)	3086.4	(11.1)	3085.5	(6.7)	3107.8	(17.6)
bloat	9948.9	(57.1)	9873.2	(27.5)	9901.2	(43.9)	9929.3	(57.6)	9889.8	(47.4)	9837.1	(22.9)
chart	10920.2	(79.7)	10809.5	(40.9)	10770.8	(89.8)	10930.6	(65.9)	10916.1	(134.6)	10839.4	(52.8)
eclipse	44801.5	(359.1)	44505.4	(1191.9)	44892.7	(490.6)	45784.1	(1510.9)	45193.2	(202.7)	45224.1	(237.7)
fop	1932.0	(12.2)	1928.1	(6.0)	1931.2	(8.0)	1980.6	(16.6)	1942.4	(5.9)	1938.2	(4.7)
hsqldb	3149.8	(50.3)	3038.2	(23.2)	3088.8	(89.6)	3149.4	(56.7)	3074.5	(27.9)	3108.5	(32.9)
jython	7143.1	(55.2)	7195.4	(31.4)	7223.1	(23.3)	7229.7	(58.7)	7231.9	(19.7)	7193.6	(28.1)
luindex	11806.3	(63.5)	11610.7	(55.4)	11821.0	(252.3)	11640.5	(70.9)	11793.5	(176.2)	11723.2	(93.7)
lusearch	5445.6	(26.8)	5402.2	(24.7)	5420.1	(26.3)	5400.3	(24.4)	5400.9	(25.0)	5392.0	(22.2)
pmd	6528.4	(34.9)	6495.6	(38.9)	6473.4	(50.2)	6519.0	(56.8)	6462.2	(51.4)	6489.1	(35.9)
xalan	7124.0	(35.4)	7036.7	(31.2)	7177.0	(33.9)	6995.3	(28.7)	7006.2	(40.4)	7016.8	(28.4)
compress	4535.2	(10.0)	4533.2	(18.9)	4508.1	(12.9)	4508.2	(20.9)	4513.7	(14.2)	4540.3	(7.2)
jess	1514.3	(5.9)	1508.8	(6.0)	1508.4	(13.1)	1511.9	(5.6)	1507.3	(9.5)	1512.2	(4.7)
db	9527.7	(57.9)	9594.5	(94.3)	9585.4	(87.9)	9534.1	(160.6)	9408.8	(215.2)	9558.0	(106.5)
javac	3685.4	(14.3)	3690.3	(8.2)	3695.8	(11.1)	3683.0	(11.0)	3687.6	(19.1)	3695.4	(9.4)
mpegaudio	2971.1	(6.7)	2971.2	(9.1)	2991.5	(26.9)	2963.3	(5.1)	2977.5	(19.7)	2972.6	(8.9)
mtrt	1701.6	(26.7)	1703.8	(15.3)	1712.4	(24.1)	1731.9	(35.1)	1718.7	(15.0)	1707.0	(14.8)
jack	3086.9	(18.8)	3083.1	(8.8)	3098.6	(11.2)	3076.3	(17.6)	3091.3	(13.3)	3082.8	(6.3)
L1	2839.5	(63.1)	2454.3	(12.2)	2473.1	(51.5)	2564.9	(11.4)	2565.8	(11.3)	2594.2	(38.9)
	Normalized Score - Harmonic Mean (± 95% confidence)											
DaCapo	1.0000	(0.0067)	0.9889	(0.0064)	0.9946	(0.0092)	0.9995	(0.0097)	0.9945	(0.0065)	0.9943	(0.0051)
SPECjvm98	1.0000	(0.0057)	1.0006	(0.0050)	1.0023	(0.0072)	1.0006	(0.0080)	0.9989	(0.0081)	1.0011	(0.0046)

Static Inline	–	*Inline fixed targets only: preserve virtual dispatch mechanisms*
LCM H:RF=8	–	*Speculative LCM-RPE, hot: relative frequency $\geq$ 8*
LCM H:RF=16	–	*Speculative LCM-RPE, hot: relative frequency $\geq$ 16*
LCM H:CF=1%	–	*Speculative LCM-RPE, hot: top 1% of cumulative frequency*
LCM H:MF=90%	–	*Speculative LCM-RPE, hot: $\geq$ 90% of maximum frequency*
LCM H:none	–	*Conservative LCM-RPE (classic LCM)*

Table A.3: **Intraprocedural RPE Performance Results - Speculation**

	Best Iteration Score - milliseconds (95%-CI)								Compilation Time - milliseconds (95%-CI)						
Benchmark	*Method Test*		*LCM S:all*		*LCM S:live*		*LCM S:CF=70%*		*LCM S:all*		*LCM S:live*		*LCM S:CF=70%*		
antlr	2610.3	(17.3)	2581.6	(8.0)	2586.5	(7.7)	2578.1	(8.8)	14313.2	(2493.2)	11954.0	(341.8)	13986.4	(980.3)	
bloat	8573.7	(66.1)	8613.1	(52.7)	8602.5	(29.9)	8596.3	(38.6)	14477.6	(1485.4)	14742.7	(1317.3)	10270.1	(400.2)	
chart	9350.3	(29.3)	9348.4	(51.0)	9350.3	(45.4)	9303.0	(41.5)	2579.4	(66.7)	2325.6	(121.6)	2194.4	(76.1)	
eclipse	41845.4	(407.8)	42549.6	(450.7)	42128.2	(482.4)	42211.0	(425.4)	124198.7	(2330.3)	72199.2	(1923.8)	30582.1	(582.9)	
fop	1797.3	(14.7)	1797.6	(9.2)	1798.3	(10.7)	1796.2	(10.4)	9172.8	(1183.2)	7192.5	(924.8)	5428.5	(548.1)	
hsqldb	2459.9	(17.2)	2520.7	(35.6)	2566.5	(37.8)	2509.6	(37.7)	20340.8	(1436.8)	12518.3	(864.2)	8988.0	(397.0)	
jython	6514.1	(43.2)	6768.6	(51.5)	6679.8	(45.9)	6615.5	(71.3)	63076.8	(11587.3)	60661.1	(1606.0)	50599.8	(1083.1)	
luindex	10774.2	(104.0)	10771.9	(55.9)	10826.2	(64.0)	10830.4	(52.1)	10809.3	(955.0)	6947.8	(394.5)	6902.2	(245.8)	
lusearch	4703.9	(36.2)	4690.7	(17.9)	4704.7	(24.9)	4696.7	(25.7)	14478.5	(922.6)	8138.5	(434.3)	6024.4	(149.6)	
pmd	4972.6	(24.0)	4985.6	(29.7)	4994.2	(30.6)	4978.1	(32.0)	10591.6	(389.6)	7947.5	(257.1)	4616.2	(103.2)	
xalan	5737.5	(32.0)	5781.0	(45.0)	5805.9	(33.3)	5764.9	(41.4)	29219.5	(3713.2)	25687.5	(1103.7)	15007.1	(340.5)	
compress	4522.7	(18.8)	4622.2	(9.1)	4520.3	(15.3)	4515.8	(11.8)	181.0	(24.3)	170.2	(21.5)	99.6	(8.3)	
jess	1344.9	(9.0)	1344.1	(9.3)	1347.5	(6.3)	1330.1	(8.6)	1090.4	(75.5)	1403.2	(377.2)	974.3	(44.5)	
db	8340.6	(117.0)	8455.3	(50.0)	8558.0	(96.9)	8410.6	(141.7)	208.9	(21.8)	221.8	(13.6)	214.6	(16.3)	
javac	3409.3	(15.8)	3418.1	(13.2)	3433.1	(16.5)	3401.2	(11.6)	17678.8	(1031.0)	13851.8	(732.7)	6624.5	(265.5)	
mpegaudio	2918.5	(24.8)	2922.9	(21.4)	2922.6	(17.6)	2925.8	(15.9)	5311.3	(590.8)	4736.4	(418.9)	1913.5	(171.9)	
mtrt	1036.8	(6.8)	1449.2	(113.2)	1162.2	(87.7)	1061.8	(10.7)	82823.0	(12069.9)	65690.1	(4890.8)	7978.8	(251.9)	
jack	2828.9	(21.4)	2817.1	(16.4)	2832.8	(10.9)	2825.4	(8.0)	5331.9	(820.0)	4558.1	(603.4)	2503.2	(231.8)	
L1	2825.6	(63.8)	2443.2	(7.0)	2534.1	(90.6)	2469.6	(35.5)	128.6	(5.0)	127.7	(3.3)	110.6	(1.0)	
	Normalized Score - Harmonic Mean (95%-CI)								Normalized Time - Harmonic Mean (95%-CI)						
DaCapo	1.0000	(0.0070)	1.0072	(0.0069)	1.0081	(0.0068)	1.0034	(0.0071)	1.0000	(0.0928)	0.7456	(0.0432)	0.5218	(0.0256)	
SPECjvm98	1.0000	(0.0074)	1.0477	(0.0202)	1.0211	(0.0170)	1.0026	(0.0074)	1.0000	(0.1110)	0.9198	(0.1175)	0.3202	(0.0369)	

Method Test – *Inline using method test and class test guards (no OSR)*
LCM S:all – *Speculative LCM-RPE, include all code*
LCM S:live – *Speculative LCM-RPE, ignore blocks that never execute*
LCM S:CF=70% – *Speculative LCM-RPE, ignore blocks below CF=70%*

Table A.4: **Intraprocedural RPE Performance Results - Selective Focus**

Benchmark	Best Iteration Score - milliseconds (± 95% confidence)											
	Static Inline		*RPE:TEF*		*RPE:CS*		*RPE:CS,EFF*		*RPE:IS*		*RPE:IS,EFF*	
antlr	3079.9	(7.0)	3063.7	(23.7)	3041.2	(8.6)	3074.9	(26.6)	3070.3	(13.1)	3290.4	(42.2)
bloat	9875.3	(50.3)	9881.6	(45.9)	9876.7	(53.8)	9922.8	(39.7)	12875.2	(145.3)	13756.3	(117.1)
chart	10942.4	(48.8)	10880.3	(62.8)	10912.7	(76.4)	10877.4	(90.7)	11968.2	(97.2)	12056.8	(72.7)
eclipse	43226.9	(2137.9)	44009.0	(477.9)	43256.4	(2130.3)	43898.4	(1017.3)	44739.3	(412.5)	44414.6	(494.3)
fop	1943.6	(7.7)	1939.7	(7.7)	1946.9	(11.0)	1954.9	(10.5)	1974.5	(22.5)	2098.0	(18.0)
hsqldb	3138.7	(50.1)	3259.0	(111.6)	3140.8	(61.6)	3181.1	(76.3)	3368.2	(79.9)	4277.7	(138.0)
jython	7167.7	(48.1)	7195.8	(54.7)	7145.9	(50.4)	7119.0	(44.2)	7143.6	(41.3)	7119.4	(38.8)
luindex	11669.4	(105.4)	11604.3	(86.8)	11633.7	(116.7)	11592.6	(128.8)	11807.8	(101.6)	12738.2	(371.1)
lusearch	5462.2	(25.5)	5647.8	(65.3)	5623.6	(53.1)	5669.9	(50.9)	5828.4	(41.0)	6774.3	(105.0)
pmd	6469.6	(28.7)	6524.0	(44.6)	6525.6	(51.9)	6529.9	(59.7)	6573.6	(38.2)	7015.0	(89.8)
xalan	7000.7	(39.3)	7058.2	(54.2)	7063.0	(52.8)	7056.1	(45.2)	7227.8	(38.3)	8504.8	(54.5)
compress	4527.1	(14.2)	4507.2	(12.6)	4507.0	(21.6)	4525.2	(11.5)	4508.2	(13.7)	4531.8	(16.6)
jess	1523.0	(7.1)	1514.3	(12.3)	1526.3	(11.3)	1536.6	(10.9)	1536.7	(7.2)	1557.2	(9.1)
db	9747.2	(54.6)	9449.3	(220.0)	9297.1	(187.1)	9722.6	(80.6)	9303.1	(242.5)	9368.3	(214.5)
javac	3704.5	(10.7)	3760.3	(22.6)	3755.5	(24.7)	3717.3	(21.8)	3717.0	(14.1)	4372.4	(22.9)
mpegaudio	2991.9	(14.3)	3233.5	(263.0)	3200.1	(293.8)	3422.9	(564.0)	3298.8	(344.7)	3558.6	(482.5)
mtrt	1720.8	(51.6)	1713.8	(53.6)	1703.4	(21.2)	1733.5	(54.6)	1705.6	(23.4)	1941.0	(25.3)
jack	3074.4	(11.9)	3078.6	(15.1)	3065.4	(17.3)	3085.2	(14.1)	3092.0	(16.4)	3504.1	(114.9)
L3	6774.9	(85.4)	6896.3	(69.7)	6866.6	(83.2)	6867.9	(79.0)	4981.3	(113.0)	5020.8	(150.7)
	Normalized Score - Harmonic Mean (± 95% confidence)											
DaCapo	1.0000	(0.0102)	1.0082	(0.0100)	1.0025	(0.0120)	1.0066	(0.0106)	1.0528	(0.0098)	1.1376	(0.0159)
SPECjvm98	1.0000	(0.0078)	1.0065	(0.0234)	1.0021	(0.0220)	1.0213	(0.0355)	1.0072	(0.0244)	1.0820	(0.0359)

Static Inline – *Inline fixed targets only: preserve virtual dispatch mechanisms*
RPE:TEF – *Local (Intraprocedural) Type Expression Folding*
RPE:CS – *Specialization only on* `final` *class fields*
RPE:CS,EFF – *Specialization on all effective-final class fields*
RPE:IS – *Specialization only on* `final` *class and instance fields*
RPE:IS,EFF – *Specialization on all effective-final class and instance fields*

TABLE A.5: **Instance RPE Performance Results vs. Static Inlining**

	Best Iteration Score - milliseconds (± 95% confidence)					
Benchmark	*Method Test*	*RPE:TEF*	*RPE:CS*	*RPE:CS,EFF*	*RPE:IS*	*RPE:IS,EFF*
antlr	2638.1 (12.1)	2779.1 (126.7)	2757.7 (104.2)	2963.5 (133.6)	2811.4 (136.0)	2843.7 (47.1)
bloat	8560.0 (59.3)	8619.5 (60.0)	8620.3 (45.3)	8661.7 (58.1)	9983.8 (307.9)	11459.1 (660.0)
chart	9287.5 (34.4)	9288.8 (46.2)	9343.5 (47.0)	9276.8 (38.6)	10232.7 (45.4)	10283.6 (59.4)
eclipse	42871.6 (425.2)	42049.1 (281.7)	42015.0 (317.9)	42578.7 (425.5)	43172.5 (445.1)	42803.2 (449.3)
fop	1793.1 (12.5)	1906.4 (42.4)	1881.3 (35.3)	1874.7 (44.5)	1913.7 (40.6)	1949.9 (43.4)
hsqldb	2467.6 (28.1)	2651.8 (47.4)	2604.3 (68.8)	2661.2 (105.2)	2743.3 (57.7)	3504.3 (258.4)
jython	6510.8 (26.4)	6551.2 (46.4)	6624.3 (81.1)	6542.0 (41.1)	6597.7 (77.0)	6523.1 (40.2)
luindex	10838.5 (74.4)	10846.3 (91.9)	10883.7 (97.9)	10955.5 (264.9)	10892.5 (80.9)	11638.6 (282.6)
lusearch	4618.5 (46.3)	4802.6 (38.5)	4814.5 (50.7)	4902.3 (193.3)	4772.6 (38.4)	5969.2 (112.9)
pmd	4987.0 (26.1)	4998.2 (33.8)	5008.9 (26.2)	5005.3 (38.5)	5015.4 (30.5)	5282.2 (57.5)
xalan	5693.3 (32.8)	5799.7 (71.6)	5800.8 (29.9)	5845.2 (68.4)	5951.4 (61.8)	7192.8 (61.0)
compress	4524.9 (17.4)	4510.4 (14.0)	4514.8 (9.9)	4525.6 (14.7)	4511.2 (10.4)	4534.9 (12.4)
jess	1336.9 (7.5)	1342.6 (7.6)	1336.9 (5.6)	1333.4 (6.7)	1336.3 (6.8)	1353.5 (15.5)
db	8205.6 (228.8)	8222.5 (234.3)	8077.0 (209.0)	8287.1 (173.9)	8247.3 (188.0)	8218.5 (194.2)
javac	3397.9 (9.1)	3455.2 (17.0)	3450.3 (17.4)	3455.9 (17.7)	3439.9 (16.6)	3986.4 (33.7)
mpegaudio	2941.8 (33.1)	3241.2 (297.9)	3171.2 (438.1)	2945.4 (27.3)	2972.2 (38.6)	3387.8 (683.1)
mtrt	1043.2 (14.2)	1042.6 (9.3)	1052.5 (16.4)	1050.1 (14.6)	1050.1 (13.5)	1099.2 (43.7)
jack	2828.1 (12.2)	2804.5 (14.6)	2827.2 (22.3)	2831.4 (11.5)	2849.8 (10.4)	3151.8 (68.6)
L3	5477.8 (719.5)	5034.1 (631.3)	5483.7 (736.3)	5203.9 (596.1)	3888.9 (323.9)	4160.5 (498.2)
	Normalized Score - Harmonic Mean (± 95% confidence)					
DaCapo	1.0000 (0.0069)	1.0216 (0.0139)	1.0204 (0.0134)	1.0312 (0.0210)	1.0543 (0.0176)	1.1409 (0.0291)
SPECjvm98	1.0000 (0.0099)	1.0149 (0.0225)	1.0113 (0.0299)	1.0048 (0.0089)	1.0055 (0.0093)	1.0685 (0.0495)

Method Test – *Inline using method test and class test guards (no OSR)*
RPE:TEF – *Local (Intraprocedural) Type Expression Folding*
RPE:CS – *Specialization only on* `final` *class fields*
RPE:CS,EFF – *Specialization on all effective-final class fields*
RPE:IS – *Specialization only on* `final` *class and instance fields*
RPE:IS,EFF – *Specialization on all effective-final class and instance fields*

TABLE A.6: **Instance RPE Performance Results vs. Guarded Inlining**

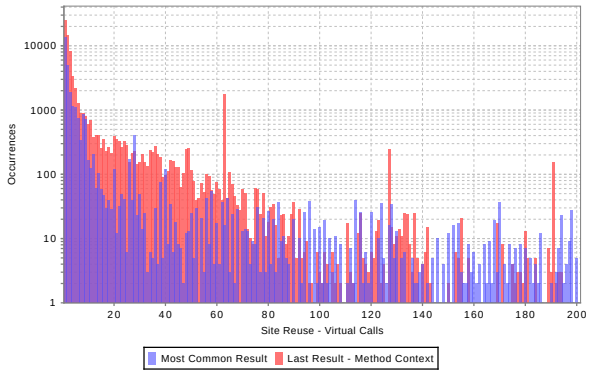
Appendix B
Polymorphism Query Profiles

The graphs presented in this appendix contrast the distributions of site-oriented and subject-oriented redundant polymorphism results, as described in §3.3.2. Summary measures for each of the various distributions are given in tables 3.3, 3.4, and 3.5.

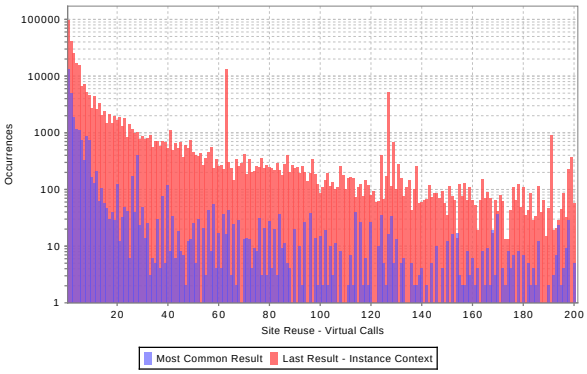
Each set of figures given here illustrates the run-lengths of redundant results seen in localized contexts relative to those measured against the *best* site-oriented result or *global* subject-oriented scope. The three diagrams on the left of each page highlight the intraprocedural redundancy (in red), while those on the right side show redundancy relative to the currently identifiable instance context. In each set of pairs (Call Target Redundancy, Type Test Redundancy, and Multiple Object Queries) the blue distributions represent the same *best* or *global* values, although some may appear different due to adjustments in the logarithmic scale. As mentioned in the discussion of §3.3.2, examples where the red counts exceed the blue counts are taken as indicative of opportunities for eliminating polymorphism results relative to the particular context considered. However, it is assumed that only a combination of high reuse counts for both site-oriented and subject-oriented measures will ultimately support effective redundancy elimination.

The graphs for compress and mpegaudio from the SPECjvm98 suite are omitted since neither benchmark exhibits significant variety in their uses of polymorphism. In both cases the distribution graphs are mostly degenerate and offer no useful insights.

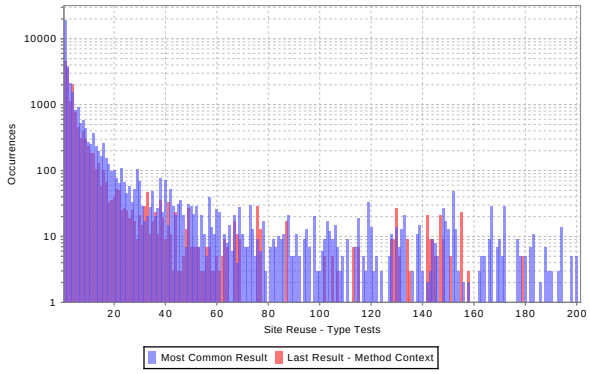
The final set of graphs given in figure B.16 only show the results for the contexts targeted by the synthetic L1 and L3 programs. In the case of the L3 program, it is worth noting that very few type tests are performed. This is by design since the program is intended to focus on dispatching in particular. Also, the subject reuse chart for L3 appears to show no significant redundancy, however this is because the primary subjects in the program are actually reused millions of times in a row (see appendix C) and thus effectively appear well outside the scope of the graph.



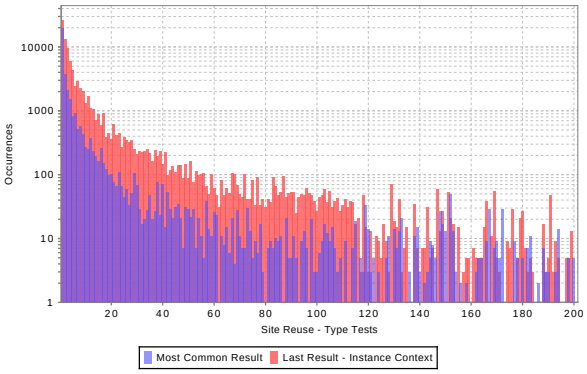
(a) Call Target Redundancy



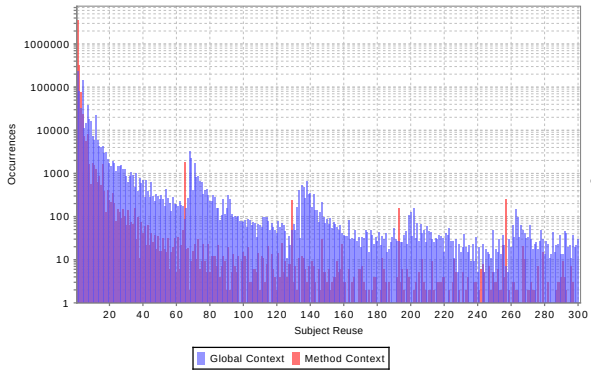
(b) Call Target Redundancy



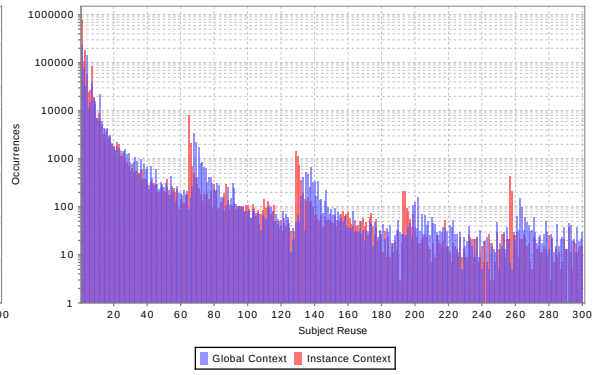
(c) Type Test Redundancy



(d) Type Test Redundancy

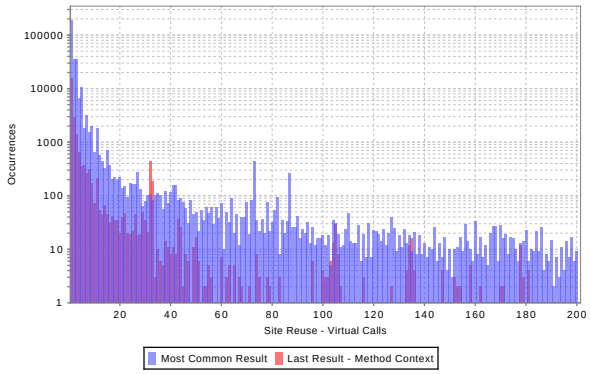


(e) Multiple Object Queries

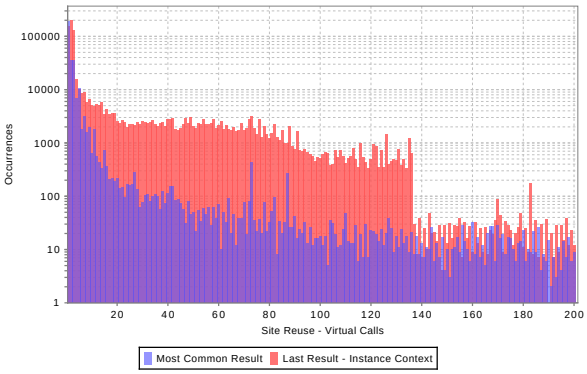


(f) Multiple Object Queries

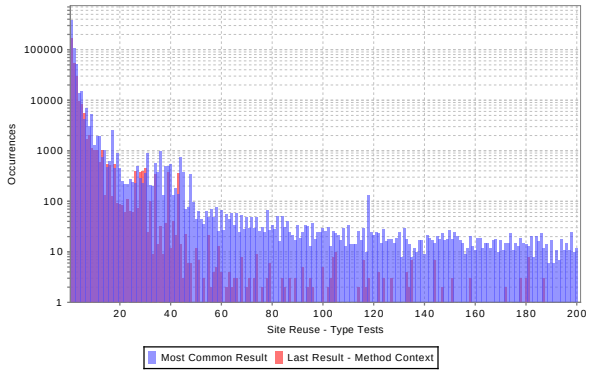
FIGURE B.1: **Polymorphism Query Redundancy for** antlr



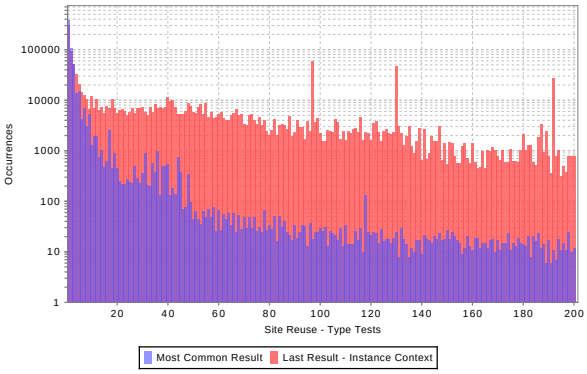
(a) Call Target Redundancy



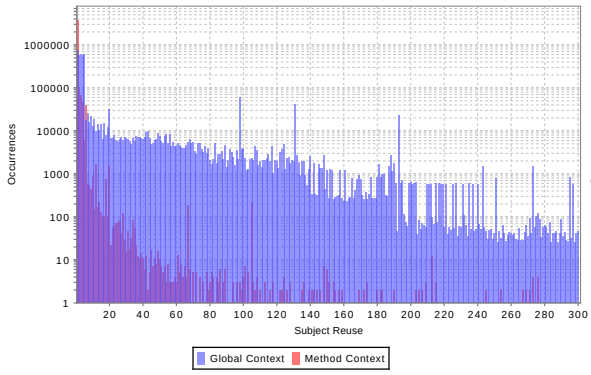
(b) Call Target Redundancy



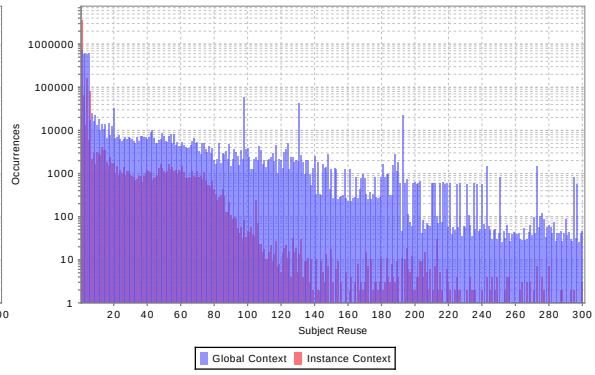
(c) Type Test Redundancy



(d) Type Test Redundancy

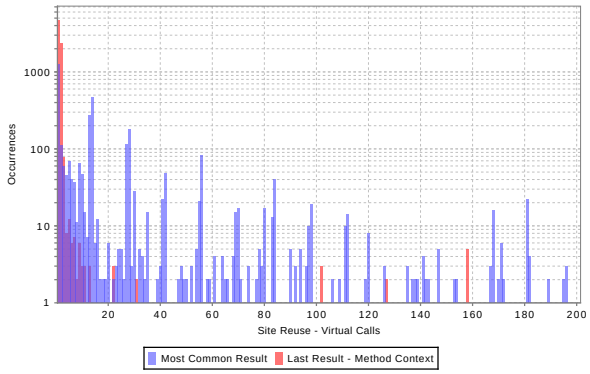


(e) Multiple Object Queries

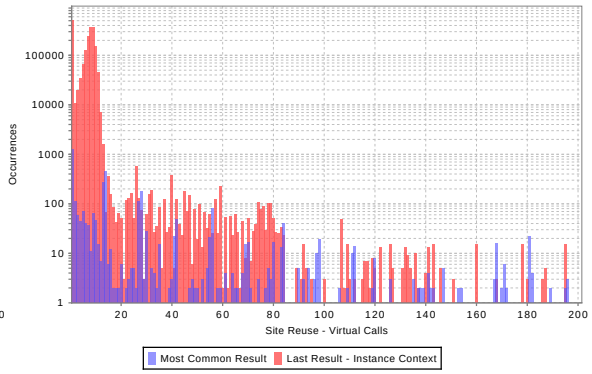


(f) Multiple Object Queries

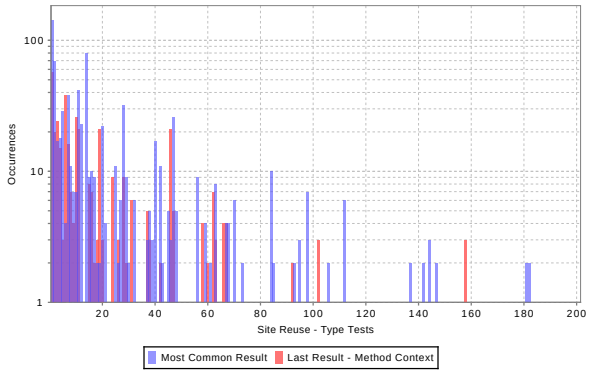
FIGURE B.2: **Polymorphism Query Redundancy for** bloat



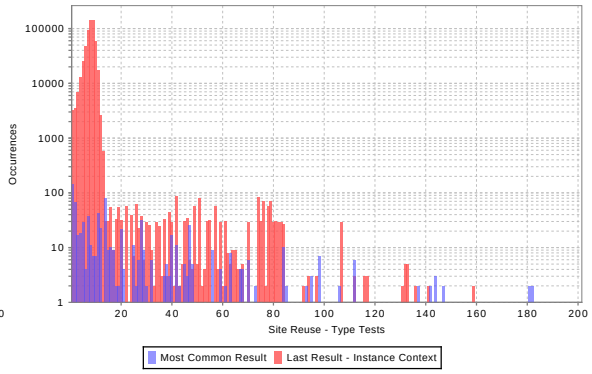
(a) Call Target Redundancy



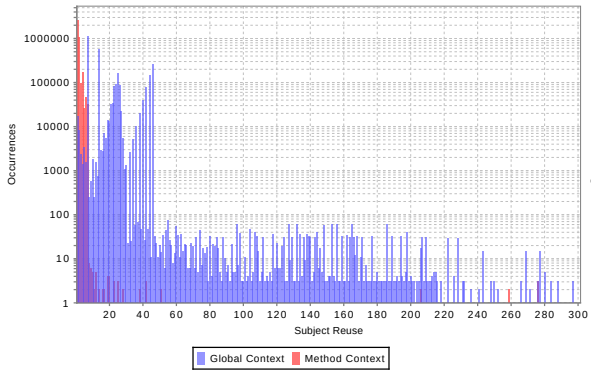
(b) Call Target Redundancy



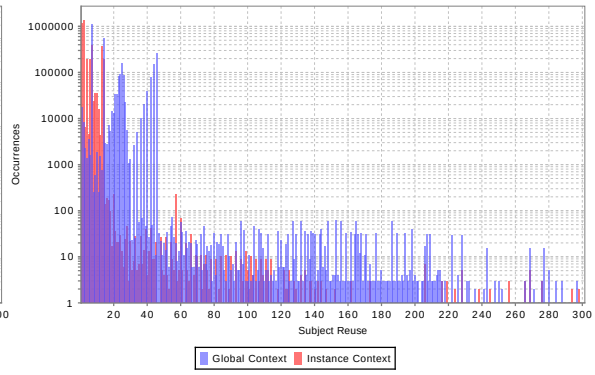
(c) Type Test Redundancy



(d) Type Test Redundancy

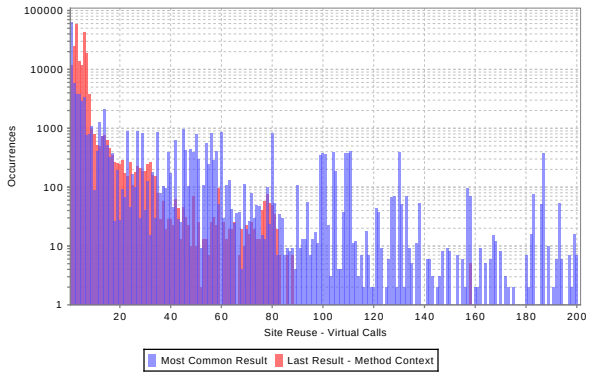


(e) Multiple Object Queries

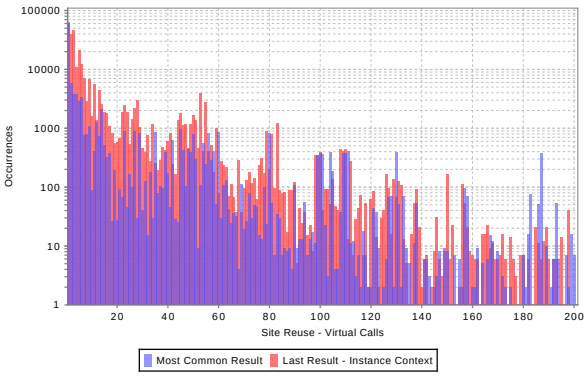


(f) Multiple Object Queries

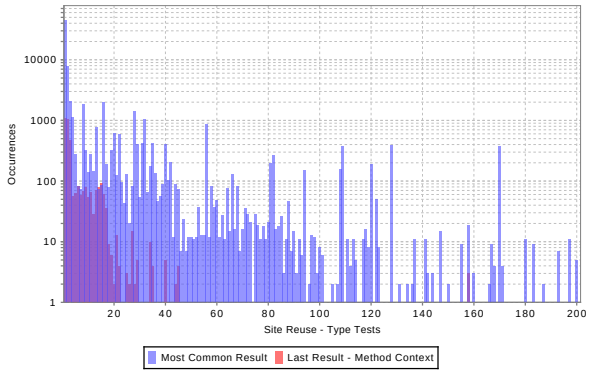
FIGURE B.3: **Polymorphism Query Redundancy for** chart



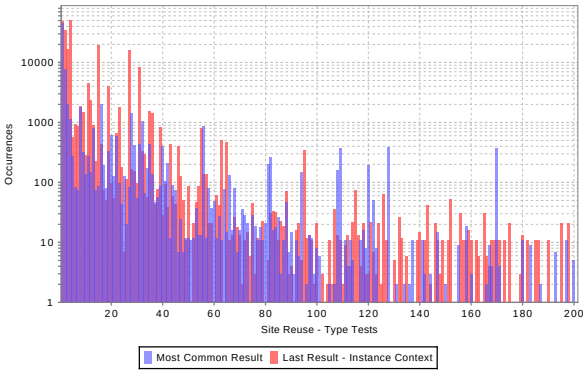
(a) Call Target Redundancy



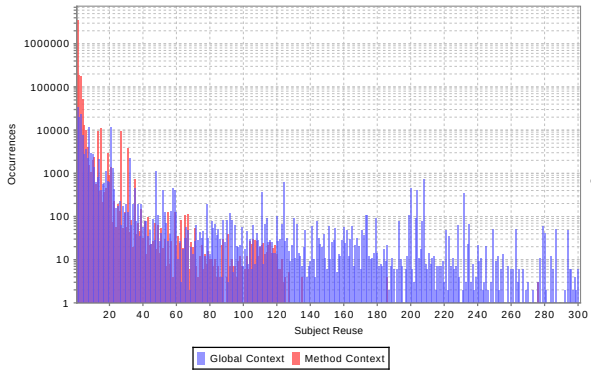
(b) Call Target Redundancy



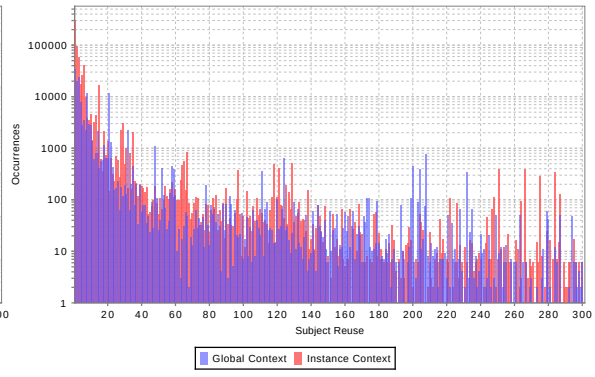
(c) Type Test Redundancy



(d) Type Test Redundancy

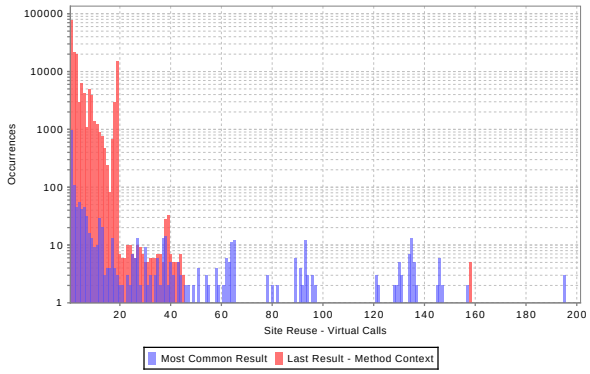


(e) Multiple Object Queries

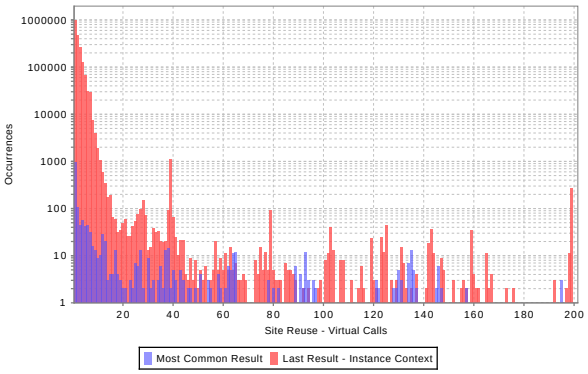


(f) Multiple Object Queries

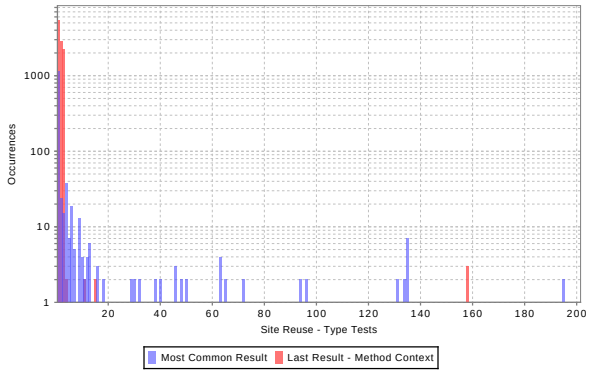
FIGURE B.4: **Polymorphism Query Redundancy for** fop



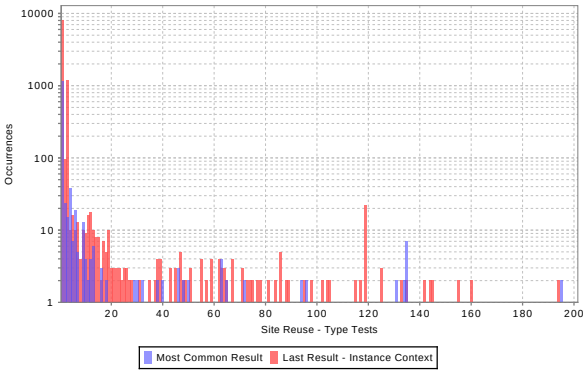
(a) Call Target Redundancy



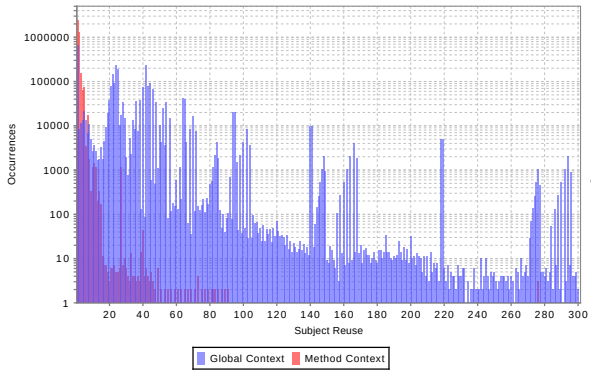
(b) Call Target Redundancy



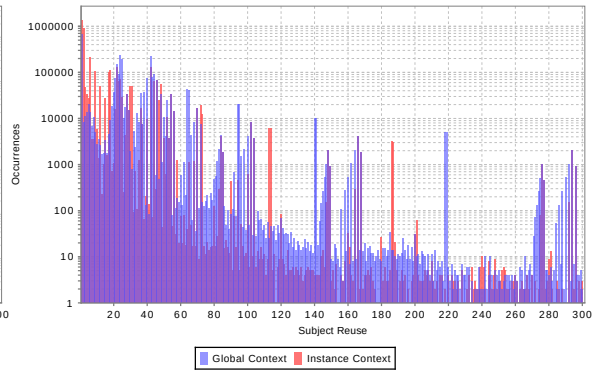
(c) Type Test Redundancy



(d) Type Test Redundancy

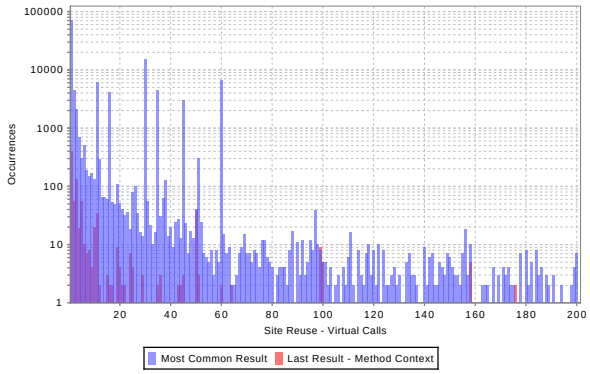


(e) Multiple Object Queries

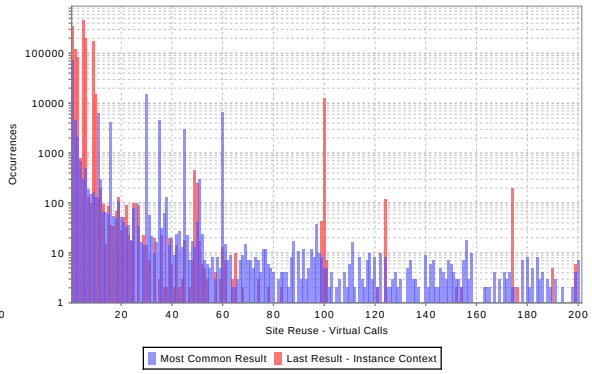


(f) Multiple Object Queries

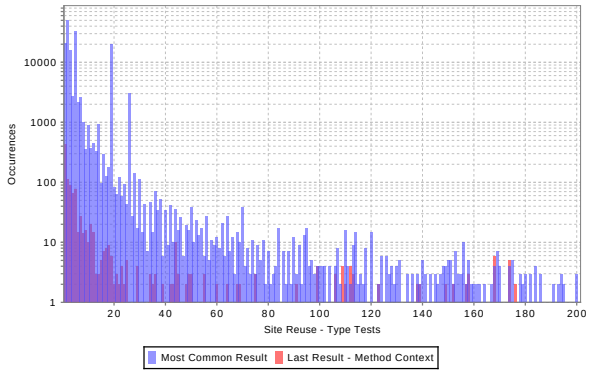
FIGURE B.5: **Polymorphism Query Redundancy for** hsqldb



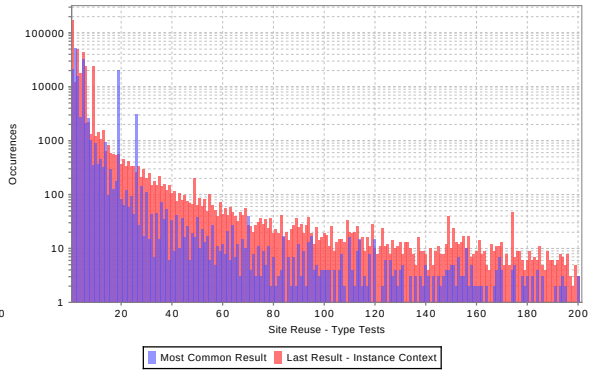
(a) Call Target Redundancy



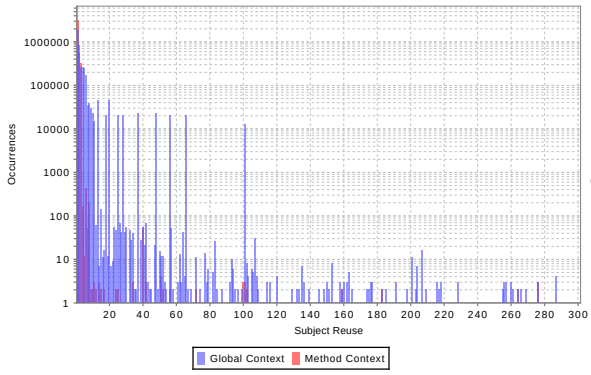
(b) Call Target Redundancy



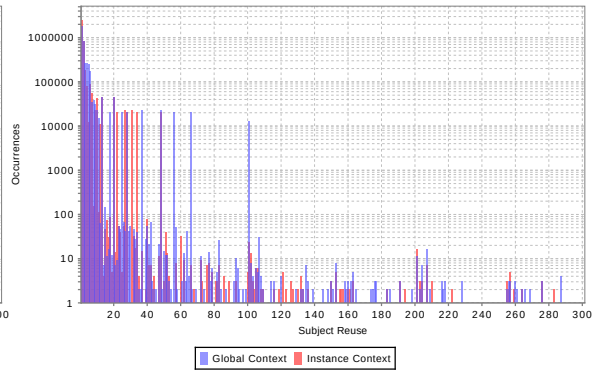
(c) Type Test Redundancy



(d) Type Test Redundancy

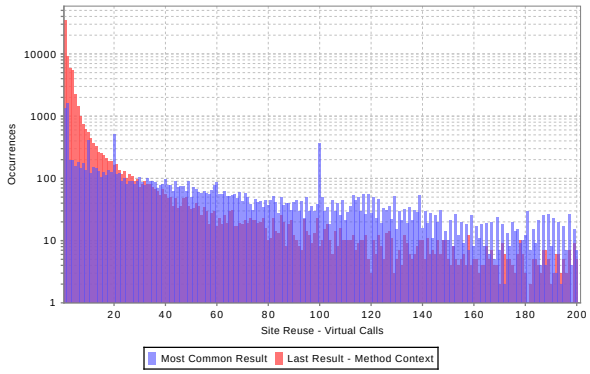


(e) Multiple Object Queries

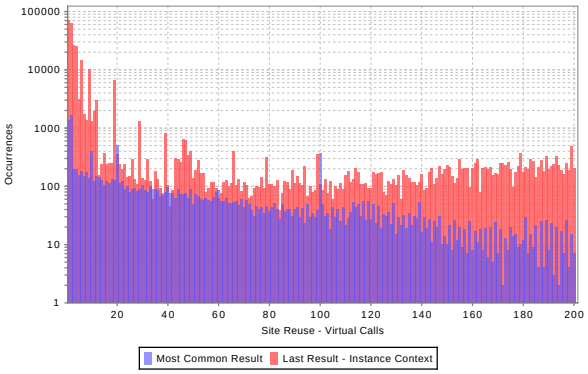


(f) Multiple Object Queries

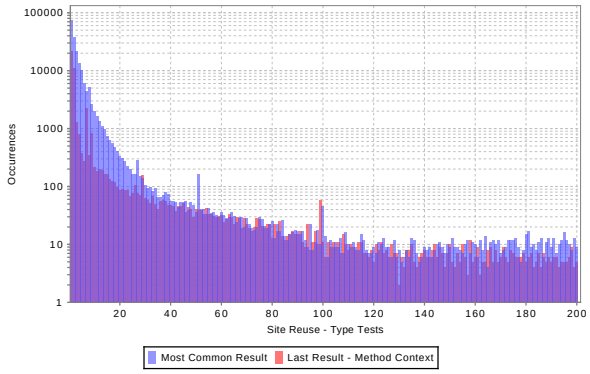
FIGURE B.6: **Polymorphism Query Redundancy for** jython



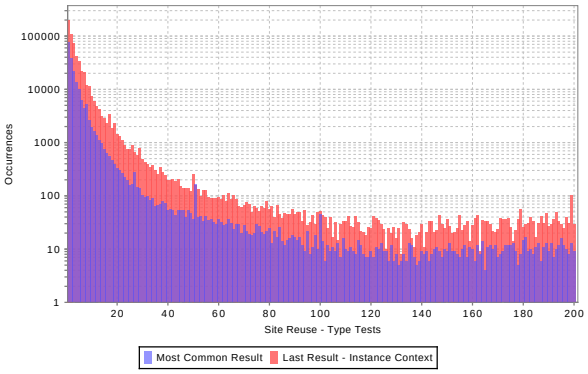
(a) Call Target Redundancy



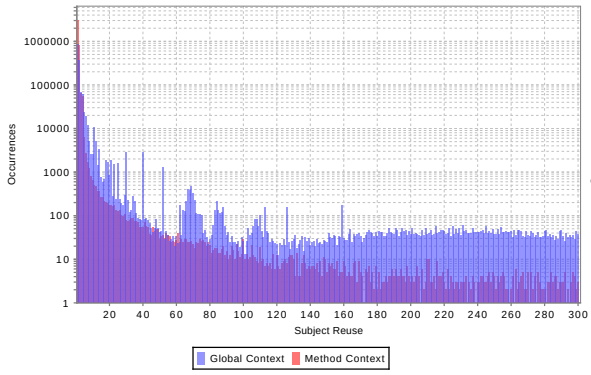
(b) Call Target Redundancy



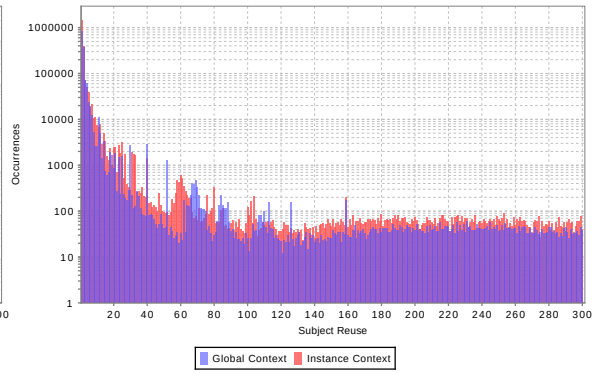
(c) Type Test Redundancy



(d) Type Test Redundancy

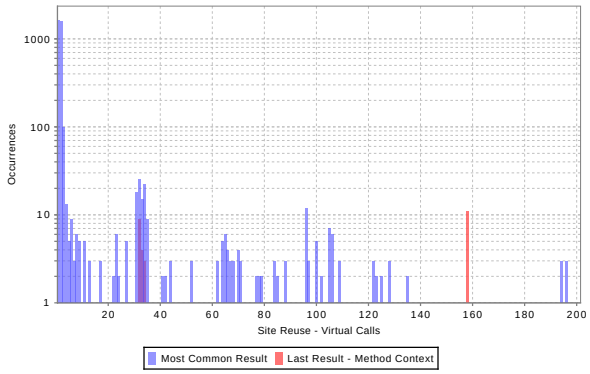


(e) Multiple Object Queries

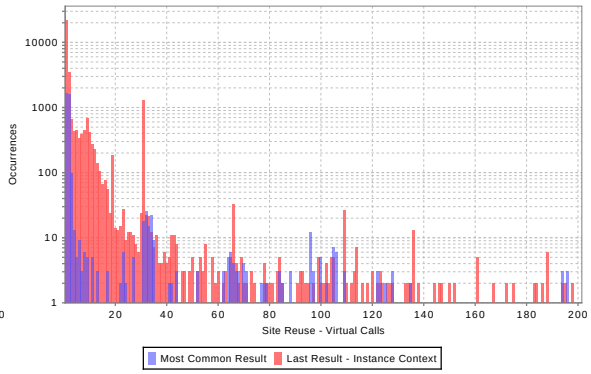


(f) Multiple Object Queries

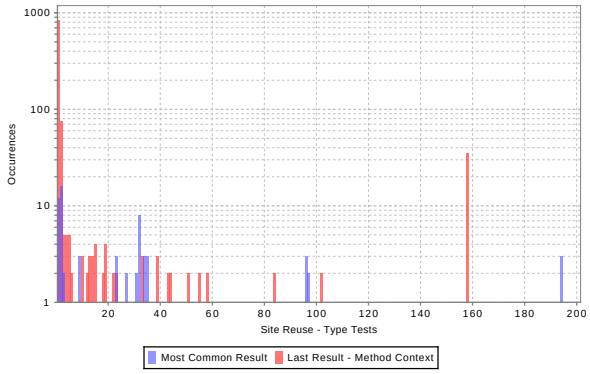
FIGURE B.7: **Polymorphism Query Redundancy for** luindex



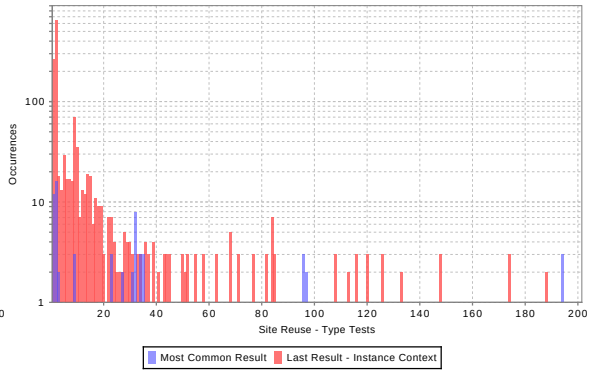
(a) Call Target Redundancy



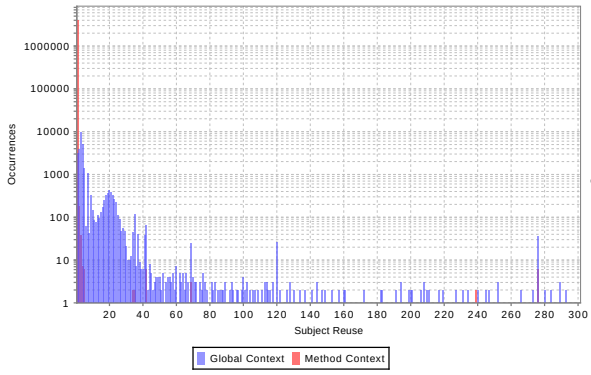
(b) Call Target Redundancy



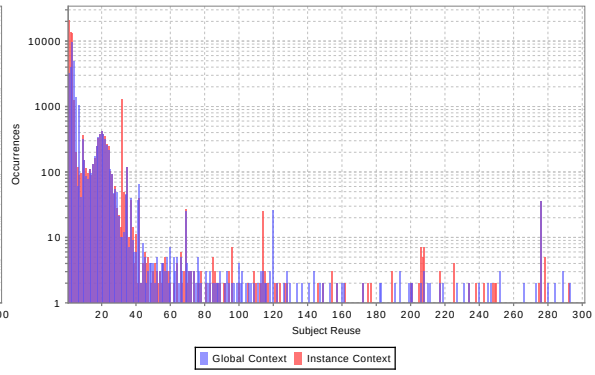
(c) Type Test Redundancy



(d) Type Test Redundancy

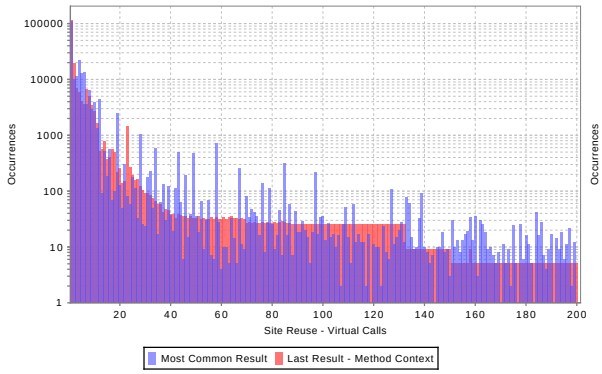


(e) Multiple Object Queries

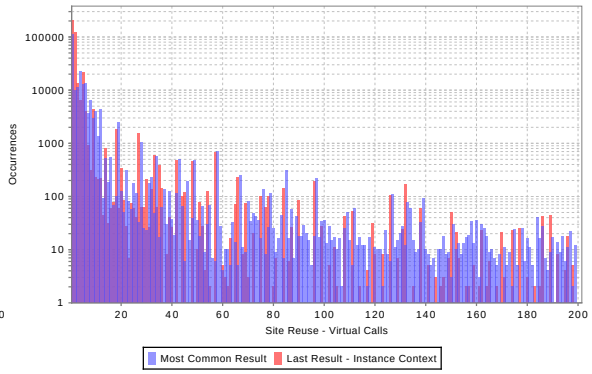


(f) Multiple Object Queries

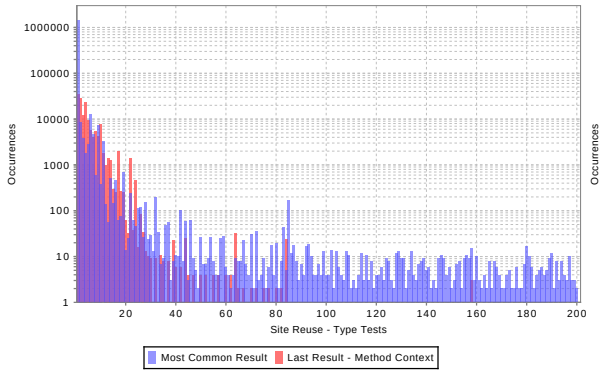
FIGURE B.8: **Polymorphism Query Redundancy for** lusearch



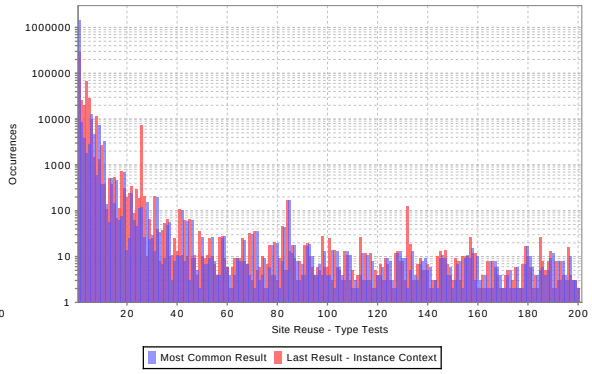
(a) Call Target Redundancy



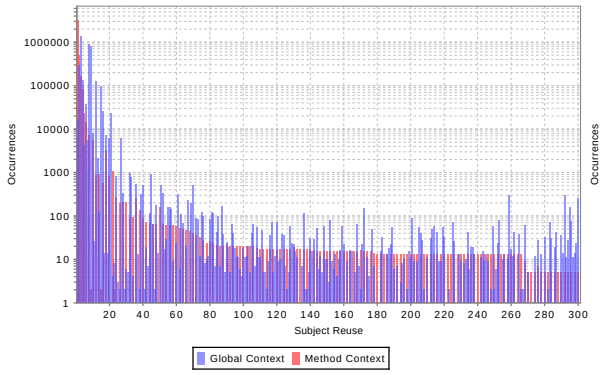
(b) Call Target Redundancy



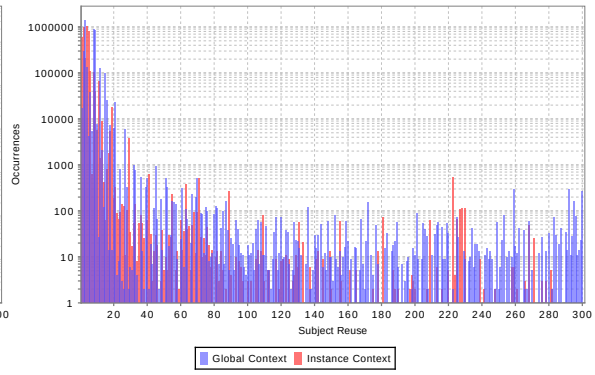
(c) Type Test Redundancy



(d) Type Test Redundancy

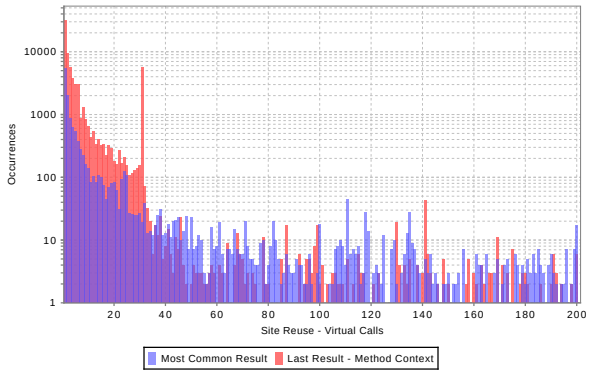


(e) Multiple Object Queries

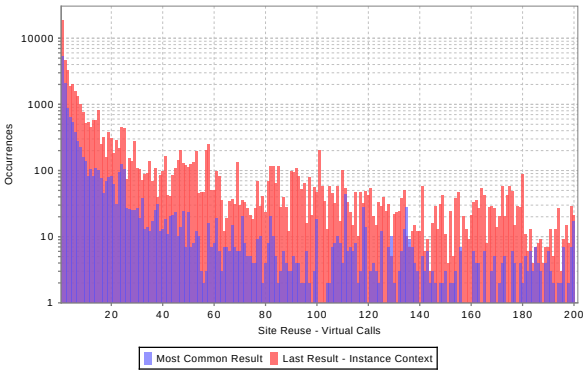


(f) Multiple Object Queries

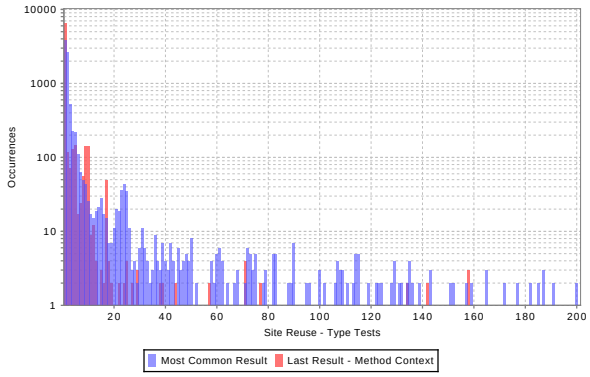
FIGURE B.9: **Polymorphism Query Redundancy for** pmd



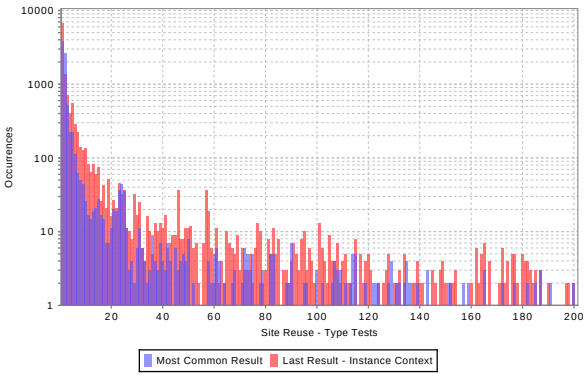
(a) Call Target Redundancy



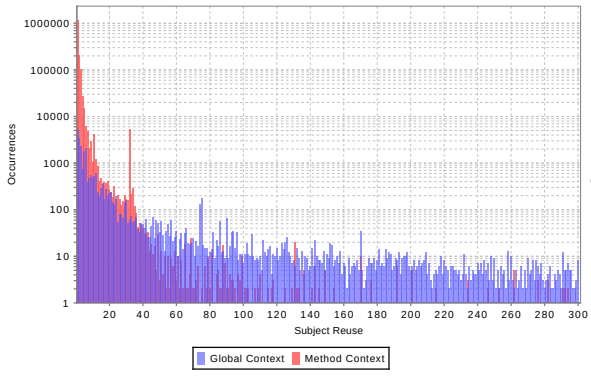
(b) Call Target Redundancy



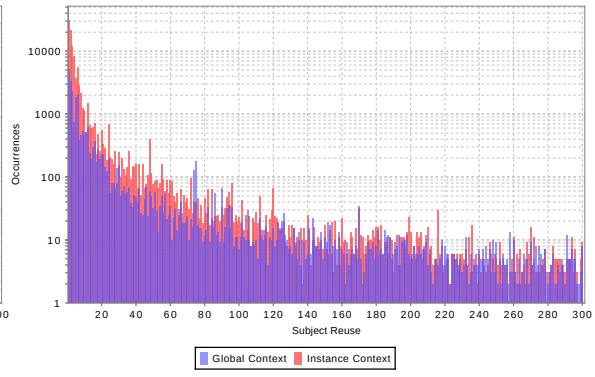
(c) Type Test Redundancy



(d) Type Test Redundancy

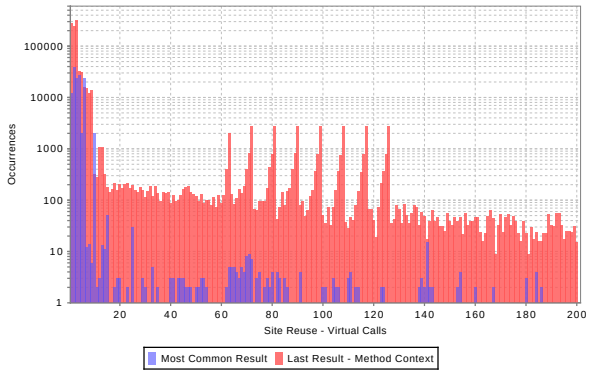


(e) Multiple Object Queries

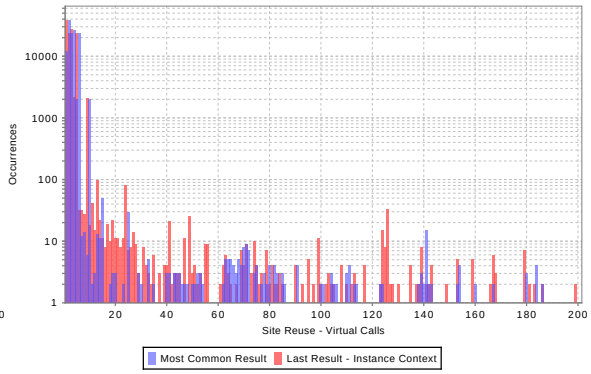


(f) Multiple Object Queries

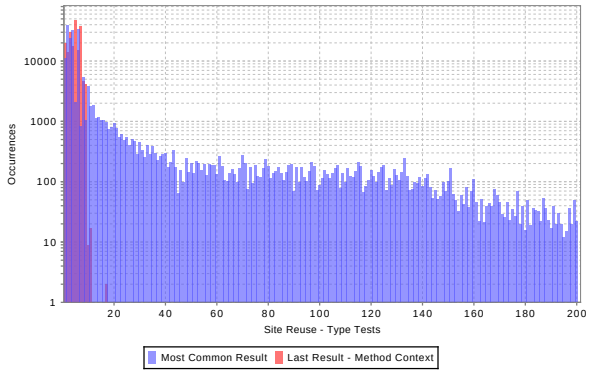
FIGURE B.10: **Polymorphism Query Redundancy for** xalan



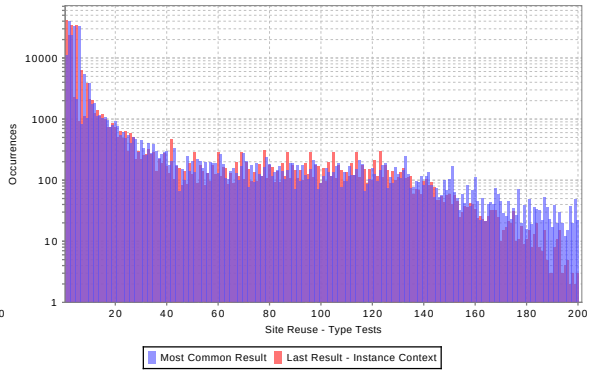
(a) Call Target Redundancy



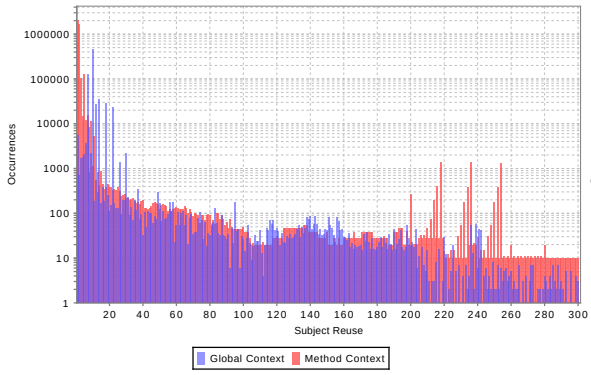
(b) Call Target Redundancy



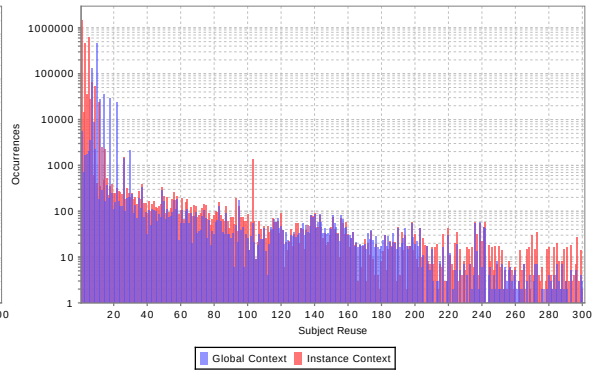
(c) Type Test Redundancy



(d) Type Test Redundancy

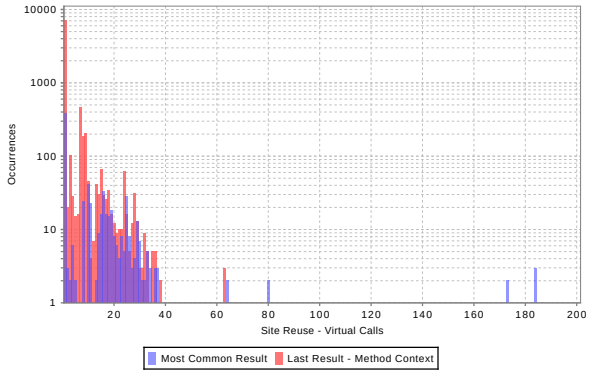


(e) Multiple Object Queries

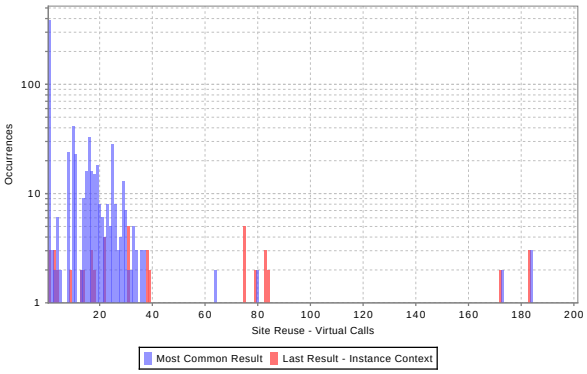


(f) Multiple Object Queries

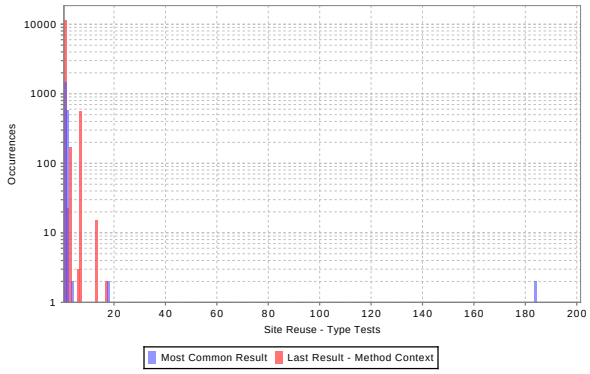
FIGURE B.11: **Polymorphism Query Redundancy for jess**



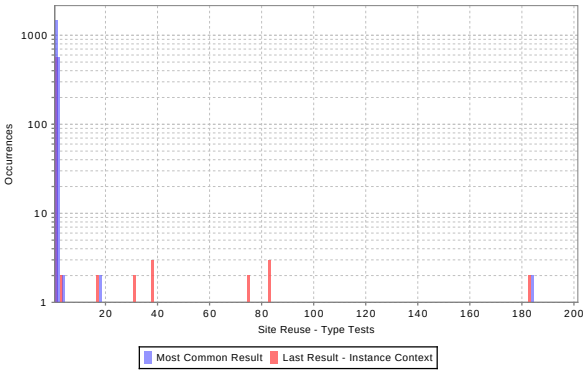
(a) Call Target Redundancy



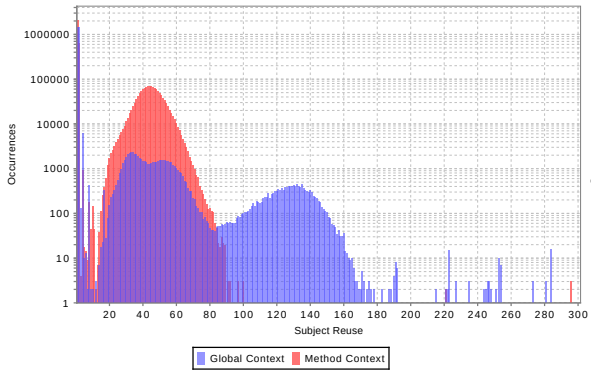
(b) Call Target Redundancy



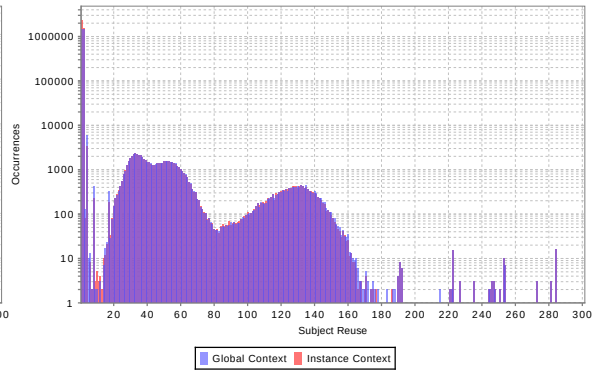
(c) Type Test Redundancy



(d) Type Test Redundancy

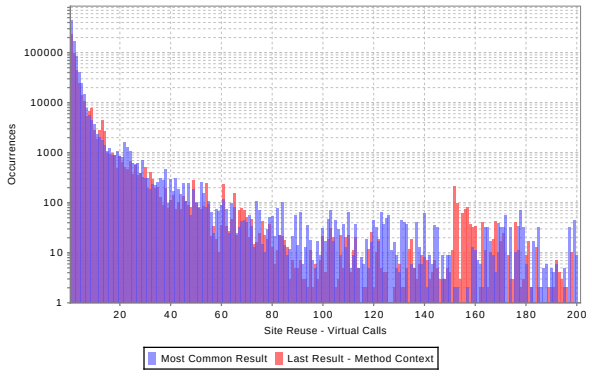


(e) Multiple Object Queries

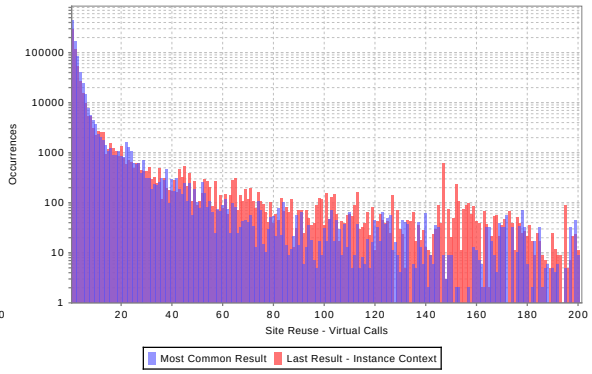


(f) Multiple Object Queries

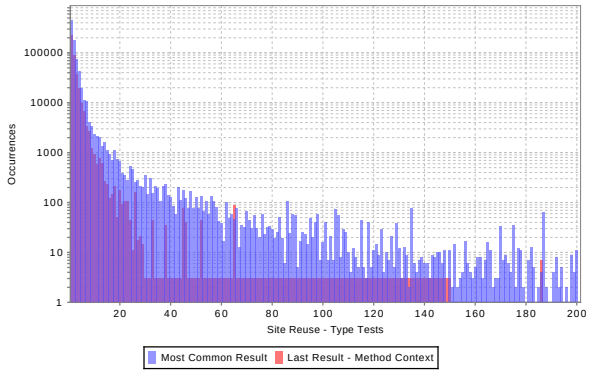
FIGURE B.12: **Polymorphism Query Redundancy for** db



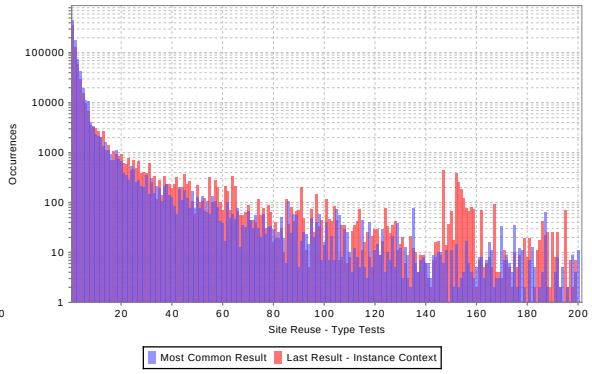
(a) Call Target Redundancy



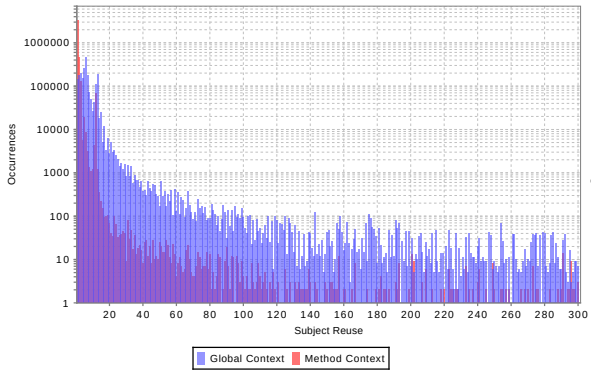
(b) Call Target Redundancy



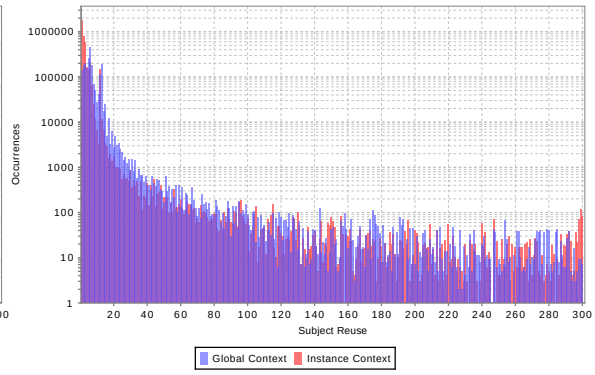
(c) Type Test Redundancy



(d) Type Test Redundancy

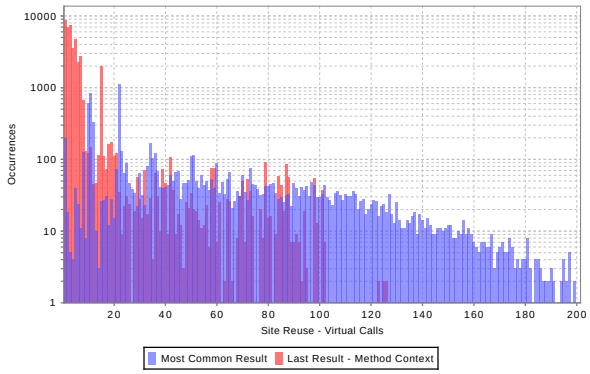


(e) Multiple Object Queries

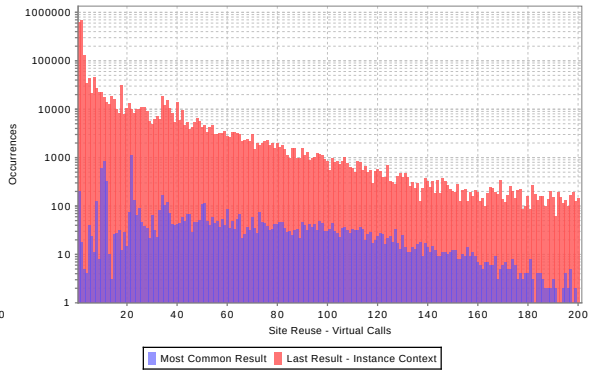


(f) Multiple Object Queries

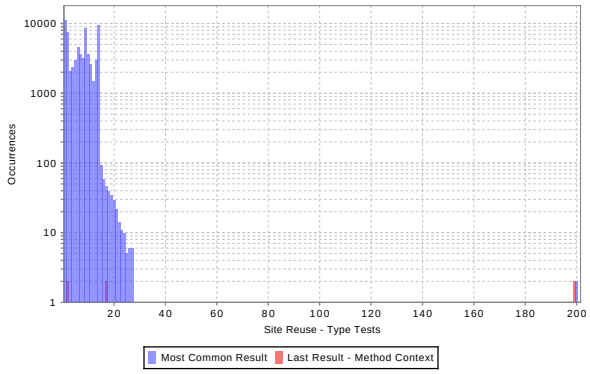
FIGURE B.13: **Polymorphism Query Redundancy for** javac



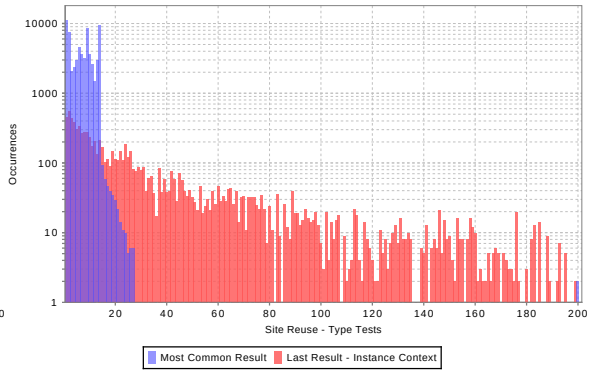
(a) Call Target Redundancy



(b) Call Target Redundancy

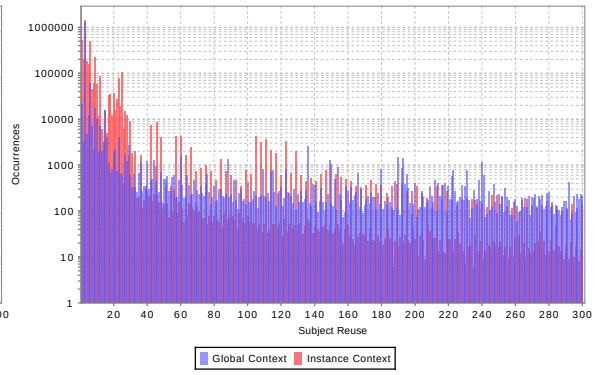


(c) Type Test Redundancy



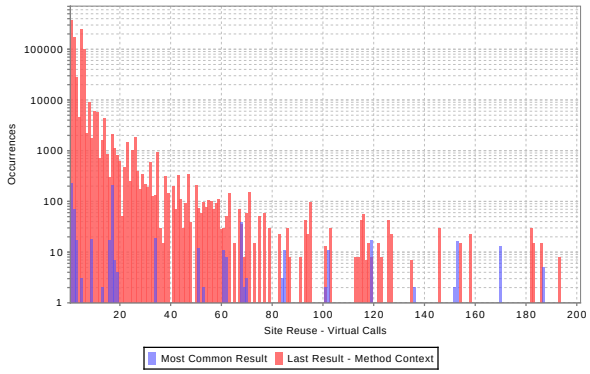
(d) Type Test Redundancy

(e) Multiple Object Queries

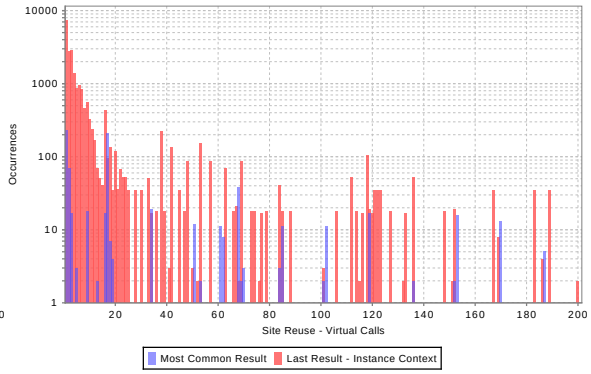


(f) Multiple Object Queries

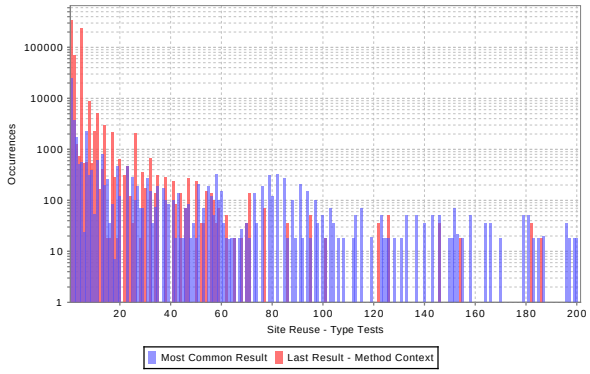
FIGURE B.14: **Polymorphism Query Redundancy for** mtrt



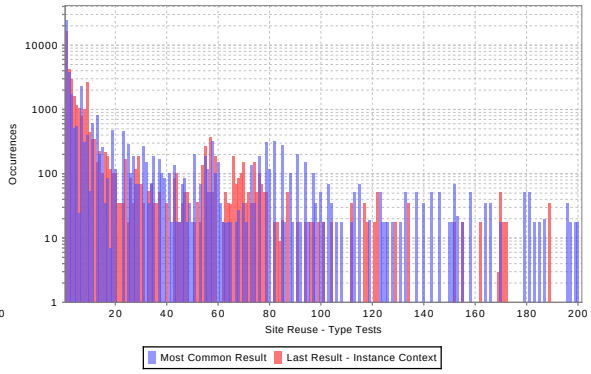
(a) Call Target Redundancy



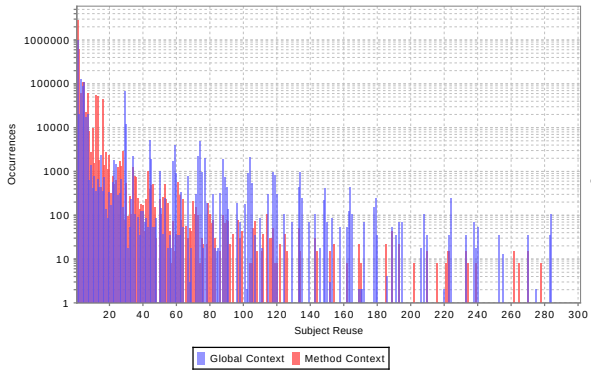
(b) Call Target Redundancy



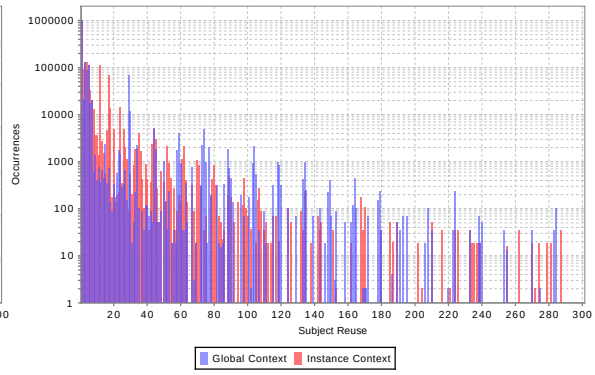
(c) Type Test Redundancy



(d) Type Test Redundancy

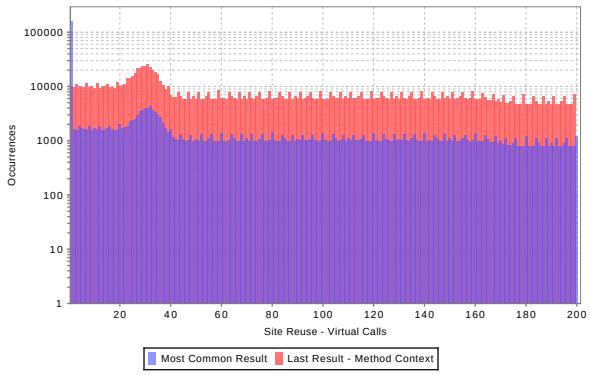


(e) Multiple Object Queries

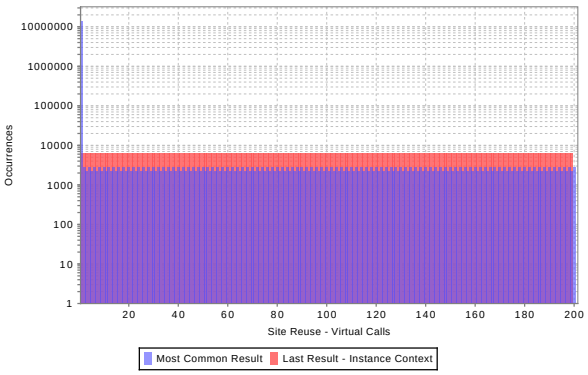


(f) Multiple Object Queries

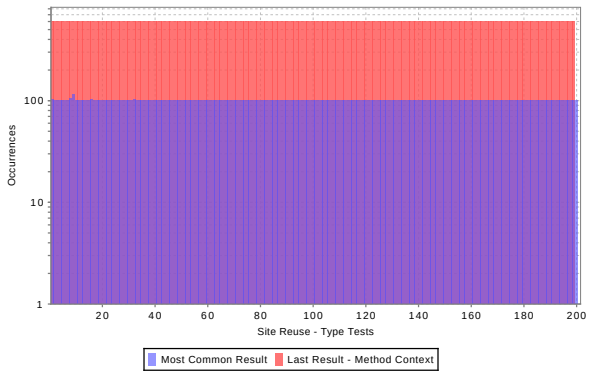
FIGURE B.15: **Polymorphism Query Redundancy for** jack



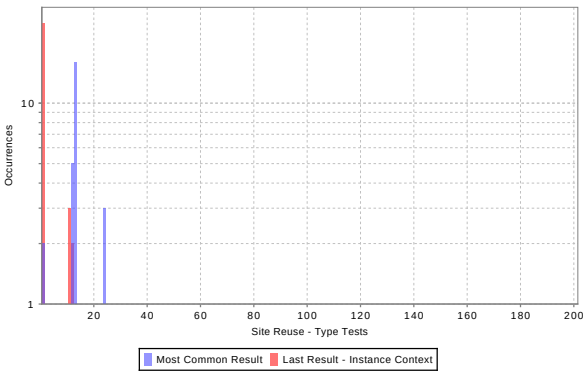
(a) L1 Call Target Redundancy



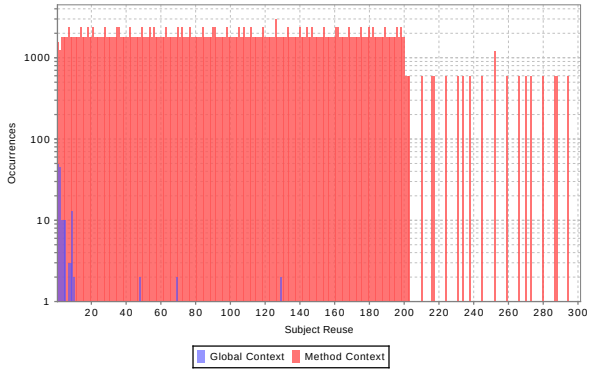
(b) L3 Call Target Redundancy



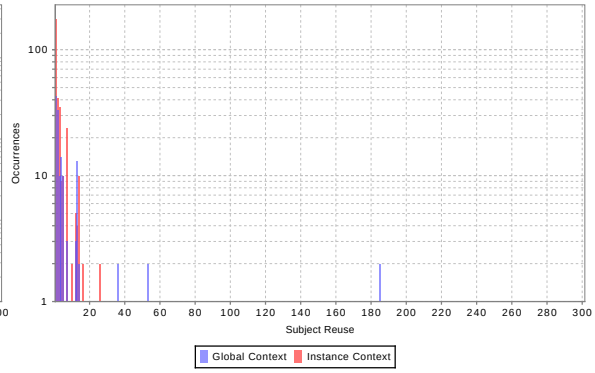
(c) L1 Type Test Redundancy



(d) L3 Type Test Redundancy



(e) L1 Multiple Object Queries



(f) L3 Multiple Object Queries

FIGURE B.16: **Polymorphism Query Redundancy for Benchmarks L1 (left) and L3 (right)**

Appendix C
Polymorphism Benchmarks

The following sections provide an overview of the L1 and L3 benchmarks used to highlight the kinds of polymorphism redundancy described in chapters 4 and 5. The intent of these programs is to focus on the load induced by virtual dispatch and type test operations. Thus, the functionality of the methods that are ultimately invoked involves little more than incrementing various counters. It is the cost incurred in reaching these targets that forms the substance of these programs.

Both programs share a common driver for a single *benchmark iteration* which is shown in figure C.1. Depending on the form of redundancy targeted—either intraprocedural or instance-lifetime—this driver is passed different pattern and subject configurations as shown in figures C.4 (for L1) and C.7 (for L3). Each pattern performs a different sequence of idiomatic applications of polymorphism operations to a given subject. The subjects each have different concrete types, and are passed to the patterns in a rotated order so that each pattern application resolves dispatches and tests differently, thus creating a challenge for site-oriented guarded inlining. The number of loops each pattern applies to a subject is ramped from 1 to `maxLoops` (200 by default), and repeated `strength` times at each loop count (100 by default). Hence, each pattern/subject combination is stressed a total of $\sum_{l=1}^{200} 100 \times l = 2,010,000$ times per benchmark iteration. As noted in §3.2.1, the benchmarks are typically run for 31 iterations, thus yielding a total of $62,310,000$ sequences of subject applications.

```
static void run(Pattern[] patterns, Subject[] subjects, int maxLoops, int strength) {
  long startTime = System.currentTimeMillis();
  for ( int loops = 1; loops <= maxLoops; loops++ )
    for ( int i = 1; i <= strength; i++ )
      for ( Subject subject : subjects )
        for ( Pattern pattern : patterns )
          pattern.execute(subject, loops);

  long time = System.currentTimeMillis() - startTime;
  summary(time);
}
```

Figure C.1: **Polymorphism Benchmark - Main Iteration**

C.1 Intraprocedural Polymorphism Redundancy: The L1 Program

To cover a variety of intraprocedural use cases, the L1 program uses six different polymorphism subjects and five different dispatching patterns. The code for the execution patterns is given in figure C.2, while the implementations of the subjects are outlined in figure C.3. The specific inputs used by the main benchmark driver are shown in figure C.4.

```
class Iterator implements Pattern {
  public void execute(
    Subject o, int loops)
  {
    while ( loops-- > 0 ) {
      while ( o.hasNext() ) {
        Subject r = o.next();
        r.action_01();
        r.action_04();
      }
      o.reset();
    }
  }
}

class Director implements Pattern {
  public void execute(
    Subject o, int loops)
  {
    while ( loops-- > 0 ) {
      o.action_01();
      o.action_02();
      o.action_03();
      o.action_04();
      o.action_03();
      o.action_02();
      o.action_01();
    }
  }
}
```

```
class Branch01 implements Pattern {
  public void execute(Subject o, int loops) {
    long bits = randomLong();
    while ( loops-- > 0 ) {
      if ( (bits & 0x1) == 0x1 )
        o.action_01();
      else
        o.action_03();
      bits >>= 1;
    }
  }
}

class Branch02 implements Pattern {
  public void execute(Subject o, int loops) {
    long bits = randomLong();
    o.action_02(); //: resolve o.a2
    o.action_04(); //: resolve o.a4
    while ( loops-- > 0 ) {
      if ( (bits & 0x1) == 0x1 )
        o.action_02();
      else
        o.action_04();
      bits >>= 1;
    }
  }
}

class CastGuard implements Pattern {
  public void execute(Subject o, int loops) {
    while ( loops-- > 0 ) {
      if ( o instanceof ASubject ) {
        ((ASubject) o).typed();
      }
    }
  }
}
```

FIGURE C.2: **Polymorphism Patterns**

The first pattern is an obvious implementation of the ITERATOR design pattern. The `Director` is intended to be a generalization of design patterns such as BUILDER, MEDIATOR, STRATEGY, and others which compose a linear sequence of actions over a subject. The `Branch01` and `Branch02` patterns present conditional control flow paths, requiring speculation to optimize the dispatch redundancy in the case of `Branch01`. Note that in both of the branching patterns the source of "random" bits used to alternate branches is exhausted after 64 loops, thus inducing a skew in the observed frequency of applied actions. The `CastGuard` pattern presents a typical test-and-cast idiom. This sequence often appears in patterns such as INTERPRETER and VISITOR, although in such patterns the tests are not usually repeated over the course of a single invocation. The `CastGuard` pattern is included to ensure that explicit type tests are reflected in the results. It is worth noting though that, because the `Subject` parameter is defined using an interface type, all of the patterns entail at least one implied type test in the form of a `MUST_IMPLEMENT` check.

```
public interface Subject {
  boolean hasNext(); //: Iterator
  Subject next();    //: Iterator
  void reset();      //: Iterator
  void action_01();  //: Director
  void action_02();  //: Director
  void action_03();  //: Director
  void action_04();  //: Director
}

abstract class ASubject implements Subject {
  // sub-classes define Subject methods
  public final void typed() { ... }
}
```

```
class S1 extends ASubject {
  public boolean hasNext() { ... }
  public Subject next()    { ... }
  public void reset()      { ... }
  public void action_01()  { ... }
  public void action_02()  { ... }
  public void action_03()  { ... }
  public void action_04()  { ... }
}

class S2 extends ASubject {
  // different implementations...
}

// similar for S3, S4, S5, S6...
```

FIGURE C.3: L1 **Polymorphism Subjects**

```
Subject[] subjects = {
  new S1(), new S2(), new S3(), new S4(), new S5(), new S6()
};

Pattern[] patterns = {
  new Iterator(), new Director(), new CastGuard(), new Branch01(), new Branch02()
};
```

FIGURE C.4: L1 **Input Configuration**

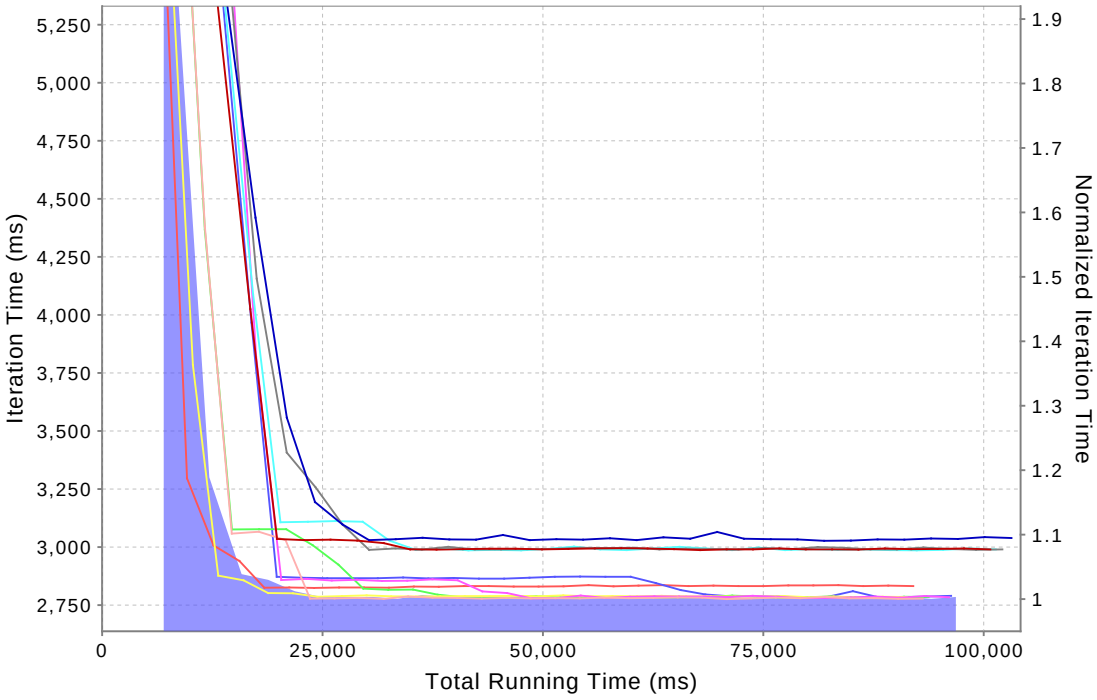


FIGURE C.5: **Performance Variation of** L1

The structure of the patterns and the variety of subjects used in the L1 program is sufficient to frustrate Jikes RVM's use of method test and class test guarded inlining strategies (see §2.4.2). In practice, what typically occurs is that only some concrete subject methods are inlined, and never in a consistent or predictable fashion. This behavior is revealed by examining the code produced by Jikes RVM's optimizing compiler for the `execute` methods of the patterns. Its effects are also apparent when the performance results of several executions are plotted together, as in figure C.5.[1] While the performance scores are very stable within an execution once the adaptive optimization system has identified and optimized the pattern code, some runs simply do not reach the best possible performance level. This is precisely because the optimizing compiler is confused about which methods to inline.

C.2 Object-Specific Polymorphism Redundancy: The L3 Program

The L3 benchmark is designed to focus specifically on calls that are relayed from one object to another. Thus the subjects for runs of L3 are constructed by creating various wrappers over the simple subject types used by L1. The wrapper implementations, which emulate the structure of the DECORATOR pattern, are given in figure C.6. As shown in figure C.7, these wrappers are assembled to form 9 primary subjects which together contain 18 distinct inner/outer couplings. Since there is no need for intraprocedural variety, the subjects are stressed using just the `Director` application pattern in this case. This particular pattern invokes 4 different methods on each subject, leading to a potential of 72 (4×18) distinct opportunities for specialization. The observed specialization counts given in table 5.1 confirm that each of these methods are identified and specialized.

The effects of poor guarded inlining choices are even more pronounced when comparing the performance of multiple L3 runs. As figure C.8 indicates, the iteration scores typically stabilize early within each execution, yet the best scores across executions may differ by as much as 60%.

```
abstract class L3_Subject extends ASubject {
  protected final Subject s;
  L3_Subject(Subject inner) { s = inner; }

  // Don't use these for L3:
  public final boolean hasNext() { return false; }
  public final Subject next()    { return null;  }
  public final void    reset()   { }
}
class W1 extends L3_Subject {
    W1(Subject s) { super(s); }

    // forward calls to inner delegate:
    public void action_01() { if ( s != null ) s.action_01(); }
    public void action_02() { if ( s != null ) s.action_02(); }
    public void action_03() { if ( s != null ) s.action_03(); }
    public void action_04() { if ( s != null ) s.action_04(); }
}
```

FIGURE C.6: L3 **Polymorphism Subjects**

[1] See §A.1 for more detail on the construction of this plot, as well as the one shown in figure C.8.

```
Subject s1 = new S1();
Subject s2 = new S2();
Subject s3 = new S3();

Subject[] subjects = {
  new W1(new W1(s1)), new W2(new W1(s2)), new W3(new W1(s3)),
  new W1(new W2(s1)), new W2(new W2(s2)), new W3(new W2(s3)),
  new W1(new W3(s1)), new W2(new W3(s2)), new W3(new W3(s3)),
};

Pattern[] patterns = { new Director() };
```

FIGURE C.7: L3 **Input Configuration**

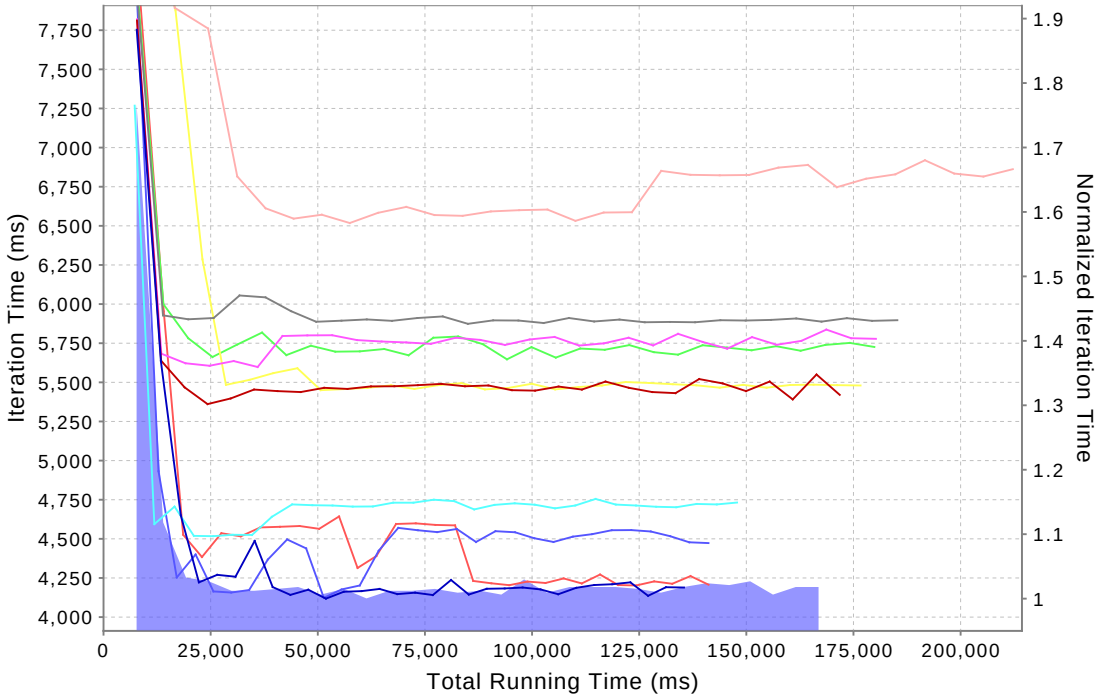


FIGURE C.8: **Performance Variation of** L3

Bibliography

Abadi, M. and Cardelli, L. (1996). *A Theory of Objects*. Monographs in Computer Science. New York, New York, USA: Springer-Verlag.

Agerbo, E. and Cornils, A. (1998). How to preserve the benefits of design patterns. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 134–143. ACM Press.

Agosta, G., Reghizzi, S. C., Palumbo, P., and Sykora, M. (2006, Apr.). Selective compilation via fast code analysis and bytecode tracing. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pp. 906–911. ACM Press.

Alpern, B., Attanasio, C. R., Barton, J. J., Burke, M. G., Cheng, P., Choi, J.-D., et al. (2000). The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 211–238. Renamed the Jikes RVM (Research Virtual Machine) in 2001.

Alpern, B., Augart, S., Blackburn, S. M., Butrico, M., Cocchi, A., Cheng, P., et al. (2005). The Jikes research virtual machine project: Building an open-source research community. *IBM Systems Journal*, 44(2), 339–417.

Alpern, B., Cocchi, A., Fink, S., and Grove, D. (2001, Oct.). Efficient implementation of Java interfaces: Invokeinterface considered harmless. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 108–124. ACM Press.

Alpern, B., Cocchi, A., and Grove, D. (2001, Apr.). Dynamic type checking in Jalapeño. In *Proceedings of the USENIX Java Virtual Machine Research and Technology Symposium (JVM)*. USENIX Association.

Alpern, B., Wegman, M. N., and Zadeck, F. K. (1988). Detecting equality of variables in programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 1–11. ACM Press.

Arnold, M., Fink, S., Grove, D., Hind, M., and Sweeney, P. F. (2004, Nov.). Architecture and policy for adaptive optimization in virtual machines. Technical Report RC23429, Yorktown Heights, New York, USA: IBM Research Division.

Arnold, M., Fink, S. J., Grove, D., Hind, M., and Sweeney, P. F. (2005, Feb.). A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2), 449–466.

Arnold, M. and Grove, D. (2005). Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pp. 51–62. IEEE Computer Society Press.

Arnold, M., Hind, M., and Ryder, B. G. (2000, Aug.). An empirical study of selective optimization. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Lecture Notes in Computer Science, vol. 2017, pp. 49–67. Springer.

Bacon, D. F., Fink, S. J., and Grove, D. (2002, Jun.). Space- and time-efficient implementation of the Java object model. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, vol. 2374, pp. 13–27. Springer.

Bacon, D. F. and Sweeney, P. F. (1996). Fast static analysis of C++ virtual function calls. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 324–341. ACM Press.

Ball, T. and Larus, J. R. (1993). Branch prediction for free. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 300–313. ACM Press.

Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., et al. (2006, Oct.). The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 169–190. ACM Press.

Bodík, R., Gupta, R., and Sarkar, V. (2000). ABCD: Eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 321–333. ACM Press.

Bodík, R., Gupta, R., and Soffa, M. L. (1997). Interprocedural conditional branch elimination. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 146–158. ACM Press.

Bodík, R., Gupta, R., and Soffa, M. L. (1999, May). Load-reuse analysis: Design and evaluation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 64–76. ACM Press.

Bodík, R., Gupta, R., and Soffa, M. L. (2004, Apr.). Retrospective: Complete removal of redundant expressions. *ACM SIGPLAN Notices*, 39(4), 596–611. Original paper from PLDI 1998.

Booch, G. (1982). Object-oriented design. *ACM SIGAda Ada Letters*, 1(3), 64–76.

Booch, G., Maksimchuk, R. A., Engle, M. W., Young, B. J., Conallen, J., and Houston, K. A. (2007). *Object-Oriented Analysis and Design with Applications*, Third ed. Object Technology Series. Upper Saddle River, New Jersey, USA: Addison-Wesley. First Edition: 1991.

Bourne, J. R. (1992). *Object-Oriented Engineering: Building Engineering Systems with Smalltalk-80*. Homewood, Illinois, USA: Irwin.

Brown, R. H. F. and Horspool, R. N. (2006, Jul.). Object-specific redundancy elimination techniques. In *Proceedings of the Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS)*. Springer.

Brown, R. H. F. and Horspool, R. N. (2009a, Dec.). Local redundant polymorphism query elimination. Unpublished report. University of Victoria. Available from http://webhome.cs.uvic.ca/~rhodesb/research/LocalRPE.pdf

Brown, R. H. F. and Horspool, R. N. (2009b, Oct.). Selective, speculative partial redundancy elimination. Unpublished report. University of Victoria. Available from http://webhome.cs.uvic.ca/~rhodesb/research/SSPRE.pdf

Budd, T. A. (1998). Chapter 1: Object-Oriented Programming, In *Handbook of Programming Languages, Volume I: Object-Oriented Programming Languages*, pp. 3–15. USA: Macmillian Technical Publishing.

Budgen, D. (2003). *Software Design*, Second ed. New York, New York, USA: Addison-Wesley.

Büttner, F., Radfelder, O., Lindow, A., and Gogolla, M. (2004). Digging into the visitor pattern. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, pp. 135–141. Knowledge Systems Institute.

Cai, Q. and Xue, J. (2003, Mar.). Optimal and efficient speculation-based partial redundancy elimination. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pp. 91–102. IEEE Computer Society Press.

Calder, B. and Grunwald, D. (1994). Reducing indirect function call overhead in C++ programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 397–408. ACM Press.

Calder, B., Grunwald, D., Lindsay, D., Martin, J., Mozer, M., and Zorn, B. (1995). Corpus-based static branch prediction. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 79–92. ACM Press.

Cardelli, L. and Wegner, P. (1985, Dec.). On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4), 471–523.

Cargill, T. A. (1993). The case against multiple inheritance in C++. In *The Evolution of C++: Language Design in the Marketplace of Ideas*, pp. 101–109. Cambridge, Massachusetts, USA: MIT Press.

Carr, S. and Kennedy, K. (1994). Scalar replacement in the presence of conditional control flow. *Software: Practice and Experience*, 24(1), 51–77.

Chambers, C., Harrison, B., and Vlissides, J. (2000). A debate on language and tool support for design patterns. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 277–289. ACM Press.

Chambers, C., Ungar, D., and Lee, E. (1989). An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 49–70. ACM Press.

Cheadle, A. M., Field, A. J., and Nystrom-Persson, J. (2008, Mar.). A method specialisation and virtualised execution environment for Java. In *Proceedings of the ACM SIGPLAN-SIGOPS International Conference on Virtual Execution Environments (VEE)*, pp. 51–60. ACM.

Coad, P. (1992). Object-oriented patterns. *Communications of the ACM*, 35(9), 152–159.

Cocke, J. (1970). Global common subexpression elimination. In *Proceedings of the a Symposium on Compiler Optimization*, pp. 20–24. ACM Press.

Cohoon, J. P. and Davidson, J. W. (2006). *Java 5.0 Program Design: An Introduction to Programming and Object-Oriented Design*. Boston, Massachusetts, USA: McGraw-Hill.

Cook, W. R. (2009). On understanding data abstraction, revisited. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 557–572. ACM Press.

Cooper, J. W. (1997). *Principles of Object-Oriented Programming in Java 1.1*. North Carolina, USA: Ventana Communications Group.

Cooper, K. D. and Xu, L. (2002, Jun.). An efficient static analysis algorithm to detect redundant memory operations. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance (MSP)*, pp. 97–107. ACM Press.

Cunei, A. and Vitek, J. (2005). PolyD: A flexible dispatching framework. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 487–503. ACM Press.

The DaCapo Benchmark Suite [Web site]. (2009). DaCapo Research Group. Visited 26-04-2010. http://dacapobench.org

Dean, J., Chambers, C., and Grove, D. (1995, Jun.). Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 93–102. ACM Press.

Dean, J., Grove, D., and Chambers, C. (1995, Aug.). Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, vol. 952, pp. 77–101. Springer.

DeRemer, F. and Kron, H. (1975). Programming-in-the large versus programming-in-the-small. In *Proceedings of the International Conference on Reliable Software*, pp. 114–121. ACM Press.

Detlefs, D. and Agesen, O. (1999, Jun.). Inlining of virtual methods. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, vol. 1628, pp. 258–278. Springer.

Deutsch, L. P. and Schiffman, A. M. (1984, Jan.). Efficient implementation of the Smalltalk-80 system. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 297–302. ACM Press.

Dolby, J. and Chien, A. (2000, Jun.). An automatic object inlining optimization and its evaluation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 345–357. ACM Press.

Driesen, K. (2001). *Efficient Polymorphic Calls*. Norwell, MA, USA: Kluwer Academic Publishers.

Driesen, K. and Hölzle, U. (1996, Oct.). The direct cost of virtual function calls in C++. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 306–323. ACM Press.

Dufour, B., Driesen, K., Hendren, L., and Verbrugge, C. (2003). Dynamic metrics for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 149–168. ACM Press.

Eeckhout, L., Georges, A., and Bosschere, K. D. (2003). How Java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 169–186. ACM Press.

Ernst, E. (2001). Family polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, vol. 2072, pp. 303–326. Springer.

Fink, S. J., Knobe, K., and Sarkar, V. (2000, Jun.). Unified analysis of array and object references in strongly typed languages. In *Proceedings of the International Static Analysis Symposium (SAS)*, Lecture Notes in Computer Science, vol. 1824, pp. 155–174. Springer.

Friedrich, M., Papajewski, H., Schröder-Preikschat, W., Spinczyk, O., and Spinczyk, U. (2000). Efficient object-oriented software with design patterns. In *Proceedings of the Generative and Component-Based Software Engineering*, Lecture Notes in Computer Science, vol. 1799, pp. 79–90. Springer.

Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M. R., et al. (2009, Jun.). Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 465–478. ACM Press.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Reading, Massachusetts, USA: Addison-Wesley.

Georges, A., Buytaert, D., and Eeckhout, L. (2007). Statistically rigorous Java performance evaluation. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 57–76. ACM Press.

Georges, A., Buytaert, D., Eeckhout, L., and Bosschere, K. D. (2004). Method-level phase behavior in Java workloads. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 270–287. ACM Press.

Gil, J. Y. and Lorenz, D. H. (1998, Mar.). Design patterns and language design. *IEEE Computer*, 31(3), 118–120.

Gil, J. Y. and Maman, I. (2005). Micro patterns in Java code. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 97–116. ACM Press.

Glew, N. and Palsberg, J. (2002, Jun.). Type-safe method inlining. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, vol. 2374, pp. 143–152. Springer.

Gosling, J., Joy, B., and Steele, G. (1996). *The Java Language Specification*, First ed. Reading, Massachusetts, USA: Addison-Wesley. Available from http://java.sun.com/docs/books/jls/

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *The Java Language Specification*, Third ed. Reading, Massachusetts, USA: Addison-Wesley. Available from http://java.sun.com/docs/books/jls/

Grove, D., DeFouw, G., Dean, J., and Chambers, C. (1997). Call graph construction in object-oriented languages. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 108–124. ACM Press.

Gu, D., Verbrugge, C., and Gagnon, E. M. (2006, Jun.). Relative factors in performance analysis of Java virtual machines. In *Proceedings of the ACM SIGPLAN-SIGOPS International Conference on Virtual Execution Environments (VEE)*, pp. 111–121. ACM Press.

Gupta, M., Choi, J.-D., and Hind, M. (2000, Jun.). Optimizing java programs in the presence of exceptions. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, vol. 1850, pp. 422–446. Springer.

Gupta, R., Berson, D. A., and Fang, J. Z. (1998, May). Path profile guided partial redundancy elimination using speculation. In *Proceedings of the International Conference on Computer Languages*, pp. 230–239. IEEE Press.

Hannemann, J. and Kiczales, G. (2002). Design pattern implementation in Java and AspectJ. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 161–173. ACM Press.

Harris, T. L. (2000, Oct.). Dynamic adaptive pre-tenuring. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pp. 127–136. ACM Press.

Hejlsberg, A., Torgersen, M., Wiltamuth, S., and Golde, P. (2008). *The C# Programming Language*, Third ed. Microsoft .NET Development Series. Addison-Wesley Professional.

Herrmann, S., Hundt, C., , and Mosconi, M. (2007). ObjectTeams/Java language definition: Version 1.0. Technical Report 2007/03, Technical University Berlin.

Heuzeroth, D., Holl, T., Högström, G., and Löwe, W. (2003). Automatic design pattern detection. In *Proceedings of the IEEE International Workshop on Program Comprehension*, pp. 94–103. IEEE Computer Society Press.

Hölzle, U., Chambers, C., and Ungar, D. (1991, Jul.). Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, vol. 512, pp. 21–38. Springer.

Hölzle, U. and Ungar, D. (1994, Oct.). A third-generation SELF implementation: Reconciling responsiveness with performance. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 229–243. ACM Press.

Horspool, R. N. and Ho, H. C. (1997, Jun.). Partial redundancy elimination driven by a cost-benefit analysis. In *Proceedings of the Israeli Conference on Computer-Based Systems and Software Engineering (ICCSSE)*, pp. 111–118. IEEE Computer Society Press.

Horspool, R. N., Pereira, D. J., and Scholz, B. (2006, Sep.). Fast profile-based partial redundancy elimination. In *Proceedings of the Joint Modular Languages Conference (JMLC)*, Lecture Notes in Computer Science, vol. 4228, pp. 362–376. Springer.

Hosking, A. L., Nystrom, N., Whitlock, D., Cutts, Q., and Diwan, A. (2001). Partial redundancy elimination for access path expressions. *Software: Practice and Experience*, 31(6), 577–600.

Ingalls, D. H. H. (1978). The Smalltalk-76 programming system design and implementation. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 9–16. ACM Press.

Intel. (2009a). *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel. Available from http://www.intel.com/products/processor/manuals/index.htm

Intel. (2009b). *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*. Intel. Available from http://www.intel.com/products/processor/manuals/index.htm

Intel. (n.d.). *Intel Pentium 4 Processor Family* [Web page]. Visited 29-06-2009. http://www.intel.com/design/Pentium4/documentation.htm

Ishizaki, K., Kawahito, M., Yasue, T., Komatsu, H., and Nakatani, T. (2000). A study of devirtualization techniques for a Java just-in-time compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 294–310. ACM Press.

Jay, B. (2009). *Pattern Calculus: Computing with Functions and Structures*. Berlin, Germany: Springer.

Jikes RVM [Web site]. (2007). Jikes RVM Project. Visited 26-04-2010. http://jikesrvm.org

Joao, J. A., Mutlu, O., Kim, H., Agarwal, R., and Patt, Y. N. (2008, Mar.). Improving the performance of object-oriented languages with dynamic predication of indirect jumps. In *Proceedings of the ACM/IEEE International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 80–90. ACM Press.

John, L. K. (2005). Chapter 4: Aggregating Performance Metrics Over a Benchmark Suite, In *Performance Evaluation and Benchmarking*, pp. 47–58. Boca Raton, Florida, USA: CRC Press.

Johnson, R. A. (2007). *An Introduction to Java Programming and Object-Oriented Application Development*. Boston, Massachusetts, USA: Thomson Course Technology.

Jones, N. D. and Muchnick, S. S. (1976). Binding time optimization in programming languages: Some thoughts toward the design of an ideal language. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 77–94. ACM Press.

Jones, R. E. and Ryder, C. (2008). A study of Java object demographics. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pp. 121–130. ACM Press.

Kaczor, O., Guéhéneuc, Y.-G., and Hamel, S. (2006, Mar.). Efficient identification of design patterns with bit-vector algorithm. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 184–194. IEEE Computer Society Press.

Kam, J. B. and Ullman, J. D. (1977, Sep.). Monotone data flow analysis frameworks. *Acta Informatica*, 7(3), 305–317.

Kawahito, M., Komatsu, H., and Nakatani, T. (2004, Sep.). Partial redundancy elimination for access expressions by speculative code motion. *Software: Practice and Experience*, 34(11), 1065–1090.

Kennedy, K. (1972, Sep.). Safety of code motion. *International Journal of Computer Mathematics*, 3(2/3), 117–130.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., et al. (1997, Jun.). Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, vol. 1241, pp. 220–242. Springer.

Knoop, J., Rüthing, O., and Steffen, B. (1992). Lazy code motion. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 224–234. ACM Press.

Knoop, J., Rüthing, O., and Steffen, B. (1995, Jun.). The power of assignment motion. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 233–245. ACM Press.

Knoop, J., Rüthing, O., and Steffen, B. (2004). Retrospective: Lazy code motion. *ACM SIGPLAN Notices*, 39(4), 460–472. Original paper from PLDI 1992.

Krasner, G. E. and Pope, S. T. (1988). A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3), 26–49.

Krintz, C., Grove, D., Lieber, D., Sarkar, V., and Calder, B. (2001, Mar.). Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8), 717–738.

Larman, C. and Guthrie, R. (2000). *Java 2 Performance and Idiom Guide*. Upper Saddle River, New Jersey, USA: Prentice-Hall.

Lewis, J. and Chase, J. (2009). *Java Software Structures: Designing and Using Data Structures*, Third ed. New York, New York, USA: Addison-Wesley.

Lewis, J. and Loftus, W. (2008). *Java Software Solutions: Foundations of Program Design*, Sixth ed. Boston, Massachusetts, USA: Addison-Wesley.

Lhoták, O. and Hendren, L. (2003, Apr.). Scaling Java points-to analysis using Spark. In *Proceedings of the International Conference on Compiler Construction (CC)*, Lecture Notes in Computer Science, vol. 2622, pp. 153–169. Springer.

Lhoták, O. and Hendren, L. (2006, Mar.). Context-sensitive points-to analysis: is it worth it? In *Proceedings of the International Conference on Compiler Construction (CC)*, Lecture Notes in Computer Science, vol. 3923, pp. 47–64. Springer.

Li, T., John, L. K., Narayanan, V., Sivasubramaniam, A., Sabarinathan, J., and Murthy, A. (2000). Using complete system simulation to characterize SPECjvm98 benchmarks. In *Proceedings of the International Conference on Supercomputing (ICS)*, pp. 22–33. ACM Press.

Lin, J., Chen, T., Hsu, W.-C., Yew, P.-C., Ju, R. D.-C., Ngai, T.-F., et al. (2004). A compiler framework for speculative optimizations. *ACM Transactions on Architecture and Code Optimization*, 1(3), 247–271.

Lindholm, T. and Yellin, F. (1999). *The Java Virtual Machine Specification*, Second ed. Reading, Massachusetts, USA: Addison-Wesley. Available from http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

Liskov, B. (1987). Data abstraction and hierarchy. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 17–34. ACM Press.

Liskov, B. and Zilles, S. (1974, Apr.). Programming with abstract data types. *ACM SIGPLAN Notices*, 9(4), 50–59. Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages.

Liu, Y. A., Stoller, S. D., Gorbovitski, M., Rothamel, T., and Liu, Y. E. (2005). Incrementalization across object abstraction. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 473–486. ACM Press.

Lu, J. and Cooper, K. D. (1997). Register promotion in C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 308–319. ACM Press.

Madsen, O. L., Magnusson, B., and Møller-Pedersen, B. (1990, Oct.). Strong typing of object-oriented languages revisited. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 140–150. ACM Press.

Marion, S., Jones, R., and Ryder, C. (2007). Decrypting the Java gene pool: Predicting objects' lifetimes with micro-patterns. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pp. 67–78. ACM Press.

Marquez, A. (2004, Feb.). Efficient implementation of design patterns in Java programs. Technical Report TR-CS-04-03, Canberra, Australia: Department of Computer Science, Australian National University.

Maurer, P. M. (2004, Mar.). Metamorphic programming: Unconventional high performance. *IEEE Computer*, 37(3), 30–38.

Millstein, T. (2004). Practical predicate dispatch. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 345–364. ACM Press.

Morel, E. and Renvoise, C. (1979). Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2), 96–103.

Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. San Francisco, California, USA: Morgan Kaufmann.

Mueller, F. and Whalley, D. B. (1995). Avoiding conditional branches by code replication. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 56–66. ACM Press.

Murphy, B. R., Menon, V., Schneider, F. T., Shpeisman, T., and Adl-Tabatabai, A.-R. (2008). Fault-safe code motion for type-safe languages. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pp. 144–154. ACM Press.

Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F. (2009). Producing wrong data without doing anything obviously wrong! In *Proceedings of the ACM/IEEE International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 265–276. ACM Press.

Nguyen, P. H. and Xue, J. (2004). Strength reduction for loop-invariant types. In *Proceedings of the Australasian Conference on Computer Science*, pp. 213–222. Australian Computer Society, Inc.

Nygaard, K. and Dahl, O.-J. (1978). The development of the SIMULA languages. *ACM SIGPLAN Notices*, 13(8), 245–272. Proceedings of the ACM SIGPLAN History of Programming Languages Conference (HOPL).

Odaira, R. and Hiraki, K. (2005). Sentinel PRE: Hoisting beyond exception dependency with dynamic deoptimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pp. 328–338. IEEE Computer Society Press.

Odersky, M. and Zenger, M. (2005). Scalable component abstractions. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 41–57. ACM Press.

Pande, H. D. and Ryder, B. G. (1994, Apr.). Static type determination for C++. In *Proceedings of the USENIX C++ Technical Conference*, pp. 85–98. USENIX Association.

Parnas, D. L. (1972, May). A technique for software module specification with examples. *Communications of the ACM*, 15(5), 330–336.

Pereira, D. J. (2007, Dec.). *Isothermality: Making speculative optimizations affordable*. Ph.D. thesis, Department of Computer Science, University of Victoria, Victoria, British Columbia, Canada.

Pree, W. (1995). *Design Patterns for Object-Oriented Software Development*. Reading, Massachusetts, USA: Addison-Wesley.

Pugh, W. (1999, Jun.). Fixing the Java memory model. In *Proceedings of the ACM SIGPLAN Java Grande Conference*, pp. 89–98. ACM Press.

Qian, F. and Hendren, L. (2004, May). Towards dynamic interprocedural analysis in JVMs. In *Proceedings of the USENIX Java Virtual Machine Research and Technology Symposium (JVM)*, pp. 139–150. USENIX Association.

Qian, F., Hendren, L., and Verbrugge, C. (2002, Apr.). A comprehensive approach to array bounds check elimination for Java. In *Proceedings of the International Conference on Compiler Construction (CC)*, Lecture Notes in Computer Science, vol. 2304, pp. 325–342. Springer.

Qian, F. and Hendren, L. J. (2005, Apr.). A study of type analysis for speculative method inlining in a JIT environment. In *Proceedings of the International Conference on Compiler Construction (CC)*, Lecture Notes in Computer Science, vol. 3443, pp. 255–270. Springer.

Rajan, H. and J.Sullivan, K. (2009). Unifying aspect- and object-oriented design. *ACM Transactions on Software Engineering and Methodology*, 19(1), 1–41.

Reges, S. and Stepp, M. (2008). *Building Java Programs: A Back to Basics Approach*. Boston, Massachusetts, USA: Addison-Wesley.

Riehle, D. (2009). Design pattern density defined. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 469–480. ACM Press.

Rogers, I., Zhao, J., Kirkham, C., and Watson, I. (2008, Sep.). Constraint based optimization of stationary fields. In *Proceedings of the ACM SIGPLAN International Conference on Principles and Practice of Programming In Java (PPPJ)*, pp. 95–104. ACM Press.

Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1988). Global value numbers and redundant computations. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 12–27. ACM Press.

Ryder, B. G., Soffa, M. L., and Burnett, M. (2005). The impact of software engineering research on modern progamming languages. *ACM Transactions on Software Engineering and Methodology*, 14(4), 431–477.

Savitch, W. and Carrano, F. M. (2009). *Java: An Introduction to Problem Solving & Programming*, Fifth ed. Upper Saddle River, New Jersey, USA: Prentice-Hall.

Scholz, B., Horspool, N., and Knoop, J. (2004, Jun.). Optimizing for space and time usage with speculative partial redundancy elimination. In *Proceedings of the ACM SIGPLAN-SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, pp. 221–230. ACM Press.

Scholz, B., Mehofer, E., and Horspool, R. N. (2003, Aug.). Partial redundancy elimination with predication techniques. In *Proceedings of the International Conference on Parallel Processing (Euro-Par)*, Lecture Notes in Computer Science, vol. 2790, pp. 242–250. Springer.

Schultz, U. P., Lawall, J. L., and Consel, C. (2000, Sep.). Specialization patterns. In *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE)*, pp. 197–206. IEEE Computer Society Press.

Schultz, U. P., Lawall, J. L., and Consel, C. (2003, Jul.). Automatic program specialization for Java. *ACM Transactions on Programming Languages and Systems*, 25(4), 452–499.

Shankar, A., Sastry, S. S., Bodík, R., and Smith, J. E. (2005). Runtime specialization with optimistic heap analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 327–343. ACM Press.

Shivers, O. (1991, May). *Control-Flow Analysis of Higher-Order Languages*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

Shuf, Y., Serrano, M. J., Gupta, M., and Singh, J. P. (2001). Characterizing the memory behavior of Java workloads: A structured view and opportunities for optimizations. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 194–205. ACM Press.

Soot: a Java Optimization Framework [Web site]. (2008, Jun.). Sable Research Group, McGill University. Visited 26-04-2010. http://www.sable.mcgill.ca/soot/

SPECjvm98 Java Benchmarks [Web site]. (2008, Sep.). Standard Performance Evaluation Corp. Visited 26-04-2010. http://www.spec.org/jvm98/

Steimann, F., Siberski, W., and Kühne, T. (2003, Jun.). Towards the systematic use of interfaces in Java programming. In *Proceedings of the ACM SIGPLAN International Conference on Principles and Practice of Programming In Java (PPPJ)*, pp. 13–17. Computer Science Press, Inc.

Stroustrup, B. (1997). *The C++ Programming Language*, Third ed. Reading, Massachusetts, USA: Addison-Wesley.

Su, L. and Lipasti, M. H. (2006, Mar.). Dynamic class hierarchy mutation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pp. 98–110. Washington, D.C., USA: IEEE Computer Society Press.

Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H., and Nakatani, T. (2001). A dynamic optimization framework for a Java just-in-time compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 180–195. ACM Press.

Suganuma, T., Yasue, T., and Nakatani, T. (2006, Jan.). A region-based compilation technique for dynamic compilers. *ACM Transactions on Programming Languages and Systems*, 28(1), 134–174.

Sullo, G. C. (1994). *Object Engineering: Designing Large-Scale Object-Oriented Systems*. New York, New York, USA: John Wiley & Sons.

Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., et al. (2000, Oct.). Practical virtual method call resolution for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 264–280. ACM Press.

Sundaresan, V., Stoodley, M., and Ramarao, P. (2008). Removing redundancy via exception check motion. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pp. 134–143. ACM Press.

Taivalsaari, A. (1996, Sep.). On the notion of inheritance. *ACM Computing Surveys*, 28(3), 438–479.

Tip, F. and Palsberg, J. (2000). Scalable propagation-based call graph construction algorithms. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 281–293. ACM Press.

Tip, F. and Sweeney, P. F. (1997, Oct.). Class hierarchy specialization. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 271–285. ACM Press.

Tourwé, T. and De Meute, W. (1999, Mar.). Optimizing object-oriented languages through architectural transformations. In *Proceedings of the International Conference on Compiler Construction (CC)*, Lecture Notes in Computer Science, vol. 1575, pp. 244–258. Springer.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., and Halkidis, S. T. (2006, Nov.). Design pattern detection using similarity scoring. *IEEE Computer*, 32(11), 896–909.

Unkel, C. and Lam, M. S. (2008, Jan.). Automatic inference of stationary fields: a generalization of Java's final fields. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 183–195. ACM Press.

Vallée-Rai, R., Gagnon, E., Hendren, L., Lam, P., Pominville, P., and Sundaresan, V. (2000, Mar.). Optimizing Java bytecode using the Soot framework: Is it feasible? In *Proceedings of the International Conference on Compiler Construction (CC)*, Lecture Notes in Computer Science, vol. 1781, pp. 18–34. Springer.

VanDrunen, T. and Hosking, A. L. (2004a). Anticipation-based partial redundancy elimination for static single assignment form. *Software: Practice and Experience*, 34(15), 1413–1439.

VanDrunen, T. and Hosking, A. L. (2004b, Mar.). Value-based partial redundancy elimination. In *Proceedings of the International Conference on Compiler Construction (CC)*, Lecture Notes in Computer Science, vol. 2985, pp. 167–184. Springer.

Volanschi, E. N., Consel, C., Muller, G., and Cowan, C. (1997, Oct.). Declarative specialization of object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 286–300. ACM Press.

Wackerly, D. D., Mendenhall, III, W., and Scheaffer, R. L. (2002). *Mathematical Statistics with Applications*, 6th ed. Duxbury Advanced Series. Pacific Grove, California, USA: Duxbury.

Waldo, J. (1993). The case for multiple inheritance in C++. In *The Evolution of C++: Language Design in the Marketplace of Ideas*, pp. 111–120. Cambridge, Massachusetts, USA: MIT Press.

Whaley, J. (2001, Oct.). Partial method compilation using dynamic profile information. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 166–179. ACM Press.

Wirfs-Brock, R., Wilkerson, B., and Wiener, L. (1990). *Designing Object-Oriented Software*. Englewood Cliffs, New Jersey, USA: Prentice-Hall.

Wulf, W. and Shaw, M. (1973, Feb.). Global variable considered harmful. *ACM SIGPLAN Notices*, 8(2), 28–34.

Zendra, O., Colnet, D., and Collin, S. (1997). Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pp. 125–141. ACM Press.

Zendra, O. and Driesen, K. (2002). Stress-testing control structures for dynamic dispatch in Java. In *Proceedings of the USENIX Java Virtual Machine Research and Technology Symposium (JVM)*, pp. 105–118. USENIX Association.

Zimmer, W. (1995). Relationships between design patterns. In J. O. Coplien and D. C. Schmidt (Eds.), *Pattern Languages of Program Design*, pp. 345–364. New York, New York, USA: Addison-Wesley.

Glossary

adaptive optimization Also: *adaptive compilation* or *feedback-directed optimization* (FDO). Just-in-time compilation strategy that incorporates on-line statistics to optimize for a particular execution. Progressively recompiles active code using more aggressive techniques. Employed by Jikes RVM (see §3.1.1), Sun's HotSpot JVM, IBM's JDK VM, and the Self runtime.

anticipatable expression Computation that occurs later along some execution path (may-be-anticipatable) or all paths (must-be-anticipatable) from the current control flow location to the exit.

available expression Computation seen previously along some execution path (may-be-available) or all paths (must-be-available) from the entry point to the current control flow location.

basic block Sequence of low-level instructions that contains no internal branching. Upon entry, will definitely execute all instructions in the block unless an exception occurs. Typically ends in an unconditional branch, or a sequence of one or more conditional branches.

benchmark execution Run of one or more benchmark iterations within the same virtual machine instance. Iteration scores are expected to improve over the course of an execution when using adaptive optimization.

benchmark iteration A single repetition of the main processing loop of a benchmark. To discount start up behavior and allow for adaptive optimization, numerous iterations are typically run within an execution.

branch history table (BHT) Hardware component used to record the (binary) choices made at conditional branching instructions. Entries are used to make predictions regarding which instruction is likely to follow a given branch. Facilitates speculative pipelined execution.

branch target buffer (BTB) Hardware component used to record the target addresses of indirect branching instructions, usually subroutine calls. Entries are used to make predictions regarding which instruction is likely to follow a given branch. Facilitates speculative pipelined execution.

busy code motion (BCM) PRE algorithm that hoists expressions to the earliest point they are definitely anticipatable, but not yet available. Permits the removal of later computations.

class test Query used by guarded inlining to ensure an exact match against the predicted subject type. In Jikes RVM, implemented by comparing TIB references. See §2.4.2.

common subexpression elimination (CSE) Redundancy elimination that identifies equivalent computations (expressions) and substitutes later instances with results from the first occurrence. The simple version operates over basic blocks. The more general "global" form (GCSE) uses a forward *available expressions* flow analysis to merge results from multiple incoming paths. Does not eliminate *partial* redundancies.

customization Form of specialization where all inherited methods are duplicated and specialized for a subclass. The primary objective is to transform internal dispatches on `self` or `this` to bind directly to the (local) specialized method address. See §5.1.

dependent load A memory load operation where the address to load from is the result of a preceding instruction. Creates pipelining delays. Often found in the implementation of polymorphism queries.

devirtualization An optimization, based on static analyses of the type hierarchy and variable usage, that identifies virtual dispatch sites with only one potential target. Transforms the dispatch to bind directly to the inferred target with no virtual table query. Allows subsequent inlining of the target code without guarding. See §2.4.1.

dynamic binding Also: *late binding*. Method call where the exact target is resolved immediately preceding the call. The choice of target is usually made based on the run-time types of the parameter values. In most object-oriented languages, the *receiver* is often the sole determinate of the target.

guarded inlining Permits inlining in the face of multiple dispatch targets. Inlined code is preceded by a test against the expected type or target; entails a polymorphism query. If the test fails, branches to a classic dispatch sequence. See §2.4.2.

inclusion polymorphism Form of dynamic polymorphism based on membership within a type hierarchy: extending a class, or implementing an interface. Resolved through run-time queries and dynamic method binding. See §2.1.

indirect call Also: *indirect branch*. Branch to a subroutine where the target is a loaded or computed address. To avoid pipelining stalls, requires specu-

lative execution and branch prediction based on a branch target buffer.

inline caching Optimization for dynamic binding that substitutes an indirect call with a direct call preceded by a guarding test against the expected target. When the guard fails frequently the guard and direct call instructions are overwritten to match the most recently resolved target. See §2.4.2.

inner class Java construct for creating a surrogate object with direct access to the `private` fields and methods of another object. Often used to implement Iterators and other adapted object views. Contains an implicit, `final` reference to the outer object instance. See §5.1.1.

instance context Dynamic context that includes the state and all method actions associated with a particular object instance. Instance-redundant computations are those that occur more than once over the span of all method invocations performed on a particular object. The extent of an instance context is thus the lifetime of its associated object, from first to last use. See §2.5.1.

interface method table (IMT) Mechanism similar to a virtual function table, used for dispatching on interface types. Conceptually, a sparse table where method signatures map to a globally unique index. Table size is reduced in Jikes RVM through use of a hashing scheme. Methods with no hash collision require three dependent loads to resolve a dispatch target. On Intel x86, hash conflicts resolved by consulting a signature ID stored in thread-local data immediately preceding dispatch.

interprocedural context Dynamic context that spans a series of active method invocations within a single thread of execution. Invocations share a common call stack. Interprocedurally redundant computations are those that occur more than once relative to some portion of the stack that remains unchanged. See §2.5.1.

intraprocedural context Dynamic context that spans the invocation (prologue through epilogue) of a particular method. Context is defined by the value of parameters and local variables. Intraprocedurally redundant computations are those that occur more than once within the scope of a single method, possibly by revisiting the same site as part of a repetitive sequence. See §2.5.1.

Java virtual machine (JVM) Program to execute Java bytecode, according to the Java Virtual Machine Specification. May interpret bytecode or use just-in-time compilation to translate to native code. Provides runtime facilities such as threading, locks, and memory management.

just-in-time (JIT) compilation Optimization for portable instruction formats, such as Java bytecode, and other interpreted languages (e.g., JavaScript). Expressions or instructions are compiled by the runtime environment into a platform specific ("native") form as needed. Jikes RVM always performs JIT compilation of bytecode, whereas other virtual machines may begin by interpreting code, then later JIT compiling heavily used elements.

lazy code motion (LCM) PRE algorithm that hoists expressions to the latest point they are anticipatable, but not yet available, without inducing unnecessary redundancy. Permits the removal of later computations. Considered preferable to busy code motion since it minimizes register pressure.

lazy compilation Just-in-time compilation strategy that translates one method at a time, immediately preceding the code's first invocation. Expends no effort on code which is never actually executed.

load elimination Optimization that identifies and eliminates redundant load operations which access the same memory location and yield the same result. Often complicated by the potential for aliasing and concurrent memory updates. Generalizations: *scalar replacement*, or *register promotion*.

loop-invariant code motion (LICM) Approach to redundancy elimination that hoists computations with invariant results to precede looping regions. Transformation is *speculative* if targeted loop body may not execute. Relies on a backward *anticipatable expressions* flow analysis. Does not eliminate partial redundancies. See §4.2.

metaobject An object that represents information about the structure of a program. Typically used to hold and control access to shared data, and support reflective inspection and execution of code. In Java, the primary form of metaobject is the `Class` object associated with each distinct type.

method test Query used by guarded inlining to ensure an exact match against the predicted method target. Captures more cases than a class test, assuming inherited methods, but at the cost of an additional load. See §2.4.2.

module context Dynamic context similar to an *instance context*, but relative to a persistent entity such as a class or *metaobject*. Module-redundant computations are those that occur anywhere a module is accessible after its initialization. See §2.5.1.

on-stack replacement (OSR) Technique for updating the compiled implementation of a method in the midst of an active invocation. Needed to undo speculative optimizations, such as unguarded inlining, when optimistic assumptions become invalid.

overloading Form of *ad hoc polymorphism* wherein the same symbol or procedure name is used to represent several distinct actions. The precise meaning of an overloaded name is deduced by inspecting the types used as operands. All combinations must be known to the compiler in order to prevent ambiguity.

overriding Mechanism common in languages that support class-oriented *inclusion polymorphism*. Permits a subclass to replace or extend a method implementation inherited from a parent class. Clients that view instances through the parent type witness polymorphic behavior when invoking the overridden method.

parametric polymorphism Variant of *universal polymorphism* in which the definition of a type is constructed from an abstract "template" parametrized with certain component or input types. Implemented as *generics* in Java. See §2.1.

partial redundancy Computation that is unnecessary (redundant) only on some execution paths. Elimination of partial redundancy is achieved by splitting or duplicating paths, or hoisting computations to reveal definite redundancy.

partial redundancy elimination (PRE) Code transformation that removes *partial redundancy*. Many variations exist, but most are based on a combination of forward and backward flow analyses that determine which results are available and which are needed (anticipatable). In a conservative formulation, only computations that are definitely anticipatable are optimized.

patching Speculative approach to inlining. If only one target is available at compile time, inline with no guard. When more targets appear, overwrites the first inlined instruction with a branch to classic virtual dispatch or OSR trigger.

polymorphic inline caching Extension of *inlining caching* designed to accommodate several common polymorphic targets. Expected target tests are arranged in a sequence, with the most common targets appearing first.

polymorphic behavior Computing action where the symbolic representations of code elements (variables, methods, etc.) appears similar but yields differing results when executed, depending on the precise typing of the elements.

polymorphism query Inspection of an object's type information, at run time, to resolve a polymorphic attribute such as membership in a type family, or the address of a method implementation. In Java, a component of all *virtual dispatch*, *type test*, and *type cast* operations. See §2.3.

polymorphism subject Also: virtual dispatch *receiver*. Primary determinant of the result of a *polymorphism query*. See §2.3.

potentially excepting instruction (PEI) An instruction, such as division or memory access, that is expected to complete, but may trigger a low-level trap which leads to an exception. Typically presents a semantic barrier to code motion.

preexistence Speculative approach to inlining. If the subject was allocated at a time when only one target was available, inline with no guard. When more targets appear, recompile with classic virtual dispatch or other inline guard.

register pressure Effect wherein a series of computations needs to retain many intermediate results, complicating the mapping of variables onto hardware registers. Often requires at least some of the results be *spilled* onto the stack. Full costs typically not apparent until the final stages of compilation.

redundant polymorphism elimination (RPE) Approach to code optimization, presented in chapters 4 and 5, that aims to reuse the results of previous *polymorphism queries* applied to the same *subject* value. Opportunities for reuse are identified relative to either an *intraprocedural* or *instance-lifetime* contextual scope.

safe code motion Strictly: a code motion strategy that does not alter the effective outcome of a program for any potential inputs. In practice, a transformation that prevents the relocation of code to positions where it may execute when it would not have in the original program structure.

selective optimization Focused approach to optimization where execution statistics are used to determine which code is most active and thus most likely to yield a benefit from aggressive compilation techniques. Alternately: give infrequently executed code a superficial treatment, or ignore it completely.

site-oriented redundancy Statistical view of redundancy which focuses on the most common usage scenario seen at the site of a computation. Does not incorporate partial results from previous uses of the same operands, and thus incurs no additional cost to hold partial results in temporary variables.

specialization Optimization where alternate method implementations are compiled assuming constant values for certain inputs. When a dispatch site is identified as satisfying a particular set of constrained inputs the call is transformed to dispatch to the specialized target instead. Similar to inlining, but without the similar growth in total code size at the cost of retaining calls and their associated overhead.

speculative optimization Strategy in which computations are relocated, anticipating that they will be executed fewer times in the new location. Preserves the effective outcome of execution (*safety*), but may permit some computation that would not have occurred using the original code structure.

static analysis Technique that aims to infer information about a program purely based on an understanding of its operational semantics. Typically formulated in terms of a flow analysis over the control representation. Used to discover results or behaviors which either may occur on some execution paths, or must occur on all possible execution paths.

static context Context determined entirely by the code expression of a program. Assumes no knowledge of actual input values, and thus presents a view which consolidates all possible execution paths.

static single assignment (SSA) form Intermediate code representation wherein every variable is assigned at exactly one location. Inserts synthetic *phi* instructions to model assignments that occur on multiple control flow paths. Facilitates certain forms of redundancy elimination, notably constant folding and copy propagation.

static typing Commonly, a programming language feature wherein the types of values must be declared and conformance in assignment and usage is enforced by the compiler. Alternately: a typing system in which the definitions associated with a type are immutable during execution.

stationary field Object instance, or `static` data field with the property that all writes to the field precede all reads of the field, thus rendering the value effectively constant over all uses. A generalization of the restrictions imposed by Java's `final` modifier.

subclassing Mechanism to permit the sharing of code by inheriting the concrete field and method definitions of an existing class. Establishes a *subtyping* relation, but only manifests polymorphism when inherited behaviors are *overridden*. See §2.2.

subject-oriented redundancy Perspective on redundancy in which partial results from previous uses of the same operands (subjects) are retained and incorporated if available. Permits elimination of cross-site redundancy, but at the cost of additional temporary variables to hold potentially reusable results.

subtyping Relationship where one type is defined as providing all of the same declared capabilities as another type, allowing its implementors to be used in place of the latter (parent) type. Does not necessarily imply any shared implementation details, thus multiple overlapping parent definitions may be established without behavioral ambiguity. See §2.2.

trace Group of instructions or methods either presumed or observed to execute in sequence. Reflects a specific control flow path. Also: program *slice*.

type cast Polymorphism operation that asserts a definite inclusion relationship. In Java, if the cast subject is not equivalent to, or a subtype of the test type a `ClassCastException` is triggered. A cast applied to a `null` value always succeeds.

type information block (TIB) Extension of the *virtual function table* concept employed by Jikes RVM to hold the address of virtual method implementations, as well as links to the related IMT, superclass and implemented interface information, and a link to the metadata representation of the type. See §2.3 and, in particular, figure 2.1.

type test Polymorphism query that determines if a subject is a member of a particular type. Returns true if the subject's type is equivalent to, extends, or implements the test type. In Java, a component of `instanceof`, `checkcast`, and `invokeinterface` instructions. In the case of `instanceof`, a `null` subject is not considered a member of any type.

universal polymorphism Major class of polymorphism, including *parametric* and *inclusion* polymorphism, where the number of possible variants for a particular expression is unbounded. See §2.1.

virtual dispatch Late binding dispatch operation in which the run-time type of the dispatch *receiver* determines the target of the call. In Java, implemented using distinct, but similar mechanisms for both `invokevirtual` and `invokeinterface`.

virtual function table (vtable) Array of pointers containing the addresses of method implementations for a particular class; accessible from every instance of the class. Method signatures are associated with an index in the array that is unique over a class' single-inheritance hierarchy. Permits virtual dispatching to be implemented using two dependent loads with precomputed offsets.

Index