

# Inlining of Virtual Methods

David Detlefs and Ole Agesen

Sun Microsystems Laboratories\*  
1 Network Drive  
Burlington, MA 01803-0902, USA  
{david.detlefs,ole.agesen}@sun.com

**Abstract.** We discuss aspects of inlining of virtual method invocations. First, we introduce a new *method test* to guard inlinings of such invocations, with a different set of tradeoffs from the class-equality tests proposed previously in the literature. Second, we consider the problem of inlining virtual methods directly, with no guarding test, in *dynamic* languages such as Self or the Java<sup>TM</sup> programming language, whose semantics prohibit a static identification of the complete set of modules that comprise a program. In non-dynamic languages, a whole-program analysis might prove the correctness of a direct virtual inlining. In dynamic languages, however, such analyses can be invalidated by later class loading, and must therefore be treated as assumptions whose later violation must cause recompilation. In the past, such systems have required an *on-stack replacement* mechanism to update currently-executing invocations of methods containing invalidated inlinings. This paper presents analyses that allow some virtual calls to be inlined directly, while ensuring that invocations in progress may complete safely even if class loading invalidates the inlining for future invocations. This provides the benefits of direct inlining without the need for on-stack replacement, which can be complicated and require space-consuming data structures.

## 1 Introduction

One of the most important jobs of a programming language is to provide good semantic abstraction boundaries. One of the most important jobs of a programming language *implementation* is to remove these abstraction boundaries to the extent necessary to permit efficient execution. Methods, one of the distinguishing characteristics of object-oriented languages, are a very important abstraction mechanism. By encapsulating functionality in a method, a programmer leaves open the possibility of reimplementing the method, adding new functionality without requiring changes at call sites. For example, consider a `Point` class

---

\* Sun, Sun Microsystems, Java, and HotJava are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

whose Cartesian coordinates are given in publicly accessible `x` and `y` fields. In some application it is found desirable to obtain the polar coordinates of points. In extending `Point` to accommodate this requirement, we decide to cache the result for efficiency, and only recompute it when the `x` or `y` coordinate changes. Since the fields are public, this requires some editing everywhere `x` or `y` are modified. If, on the other hand, the fields had been private, and public uses had been mediated by `get` and `set` methods, then the recomputation of the polar form would be a simple local change to the `Point` class.

At present, many programmers recognize the wisdom of such observations, but avoid the use of the extra abstraction. For immature programmers the reason may be to avoid some typing, but even mature programmers may avoid abstraction because they are (legitimately) wary of extra cost. Thus, if we are to allow programmers to write in the most modular and robust style without incurring a performance penalty, we must ensure that simple methods execute as fast as if the extra abstraction layer were not present. In practice, this dictates that such methods be inlined.

The above remarks apply generally to most programming languages. Object-oriented languages, such as Simula [3], C++ [26], Modula-3 [22], Smalltalk [14], Eiffel [21], Trellis [24], CLOS [13], and the Java<sup>TM</sup> programming language [15], to name just a few, complicate inlining, because methods are usually *virtual*. Virtual methods are defined in one class, but may be *overridden* in subclasses of that class. The method actually invoked at a call site depends on the dynamic type of the *receiver* object.

Inlining of virtual methods is difficult because a given call site may invoke several different actual methods over the course of a program execution. Thus, it may be impossible to uniquely identify code to inline. However, in many programs some virtual call sites actually execute only one method, i.e., are *monomorphic* rather than *polymorphic*. Some call sites are provably monomorphic; others are “almost monomorphic,” in that several methods might be executed, but one is executed much more frequently than the others.

In the latter situation, a strategy that has been used previously in the literature is to select code to be inlined, then generate a test to guard the inlined code to ensure that it is correct for the dynamic type of the current receiver. If the test fails, the normal virtual call mechanism is used. The guard test is usually a *class test*, verifying that the class of the receiver object matches a particular class. One contribution of this paper is to introduce an alternative test, a *method test* that compares the address of the method inlined with the address of the virtual method to be executed. This incurs the overhead of an extra load, but is more robust, and in some ways more suited to dynamic compilation.

Most analyses that prove a call site monomorphic require interprocedural techniques. Such analyses work best for languages where the full set of components that comprise a program is known statically. Languages with more dynamic semantics make such analyses more difficult. In the Java programming language, for example, `Class.forName` finds and loads a class with a dynamically com-

puted name. Use of this facility complicates interprocedural analyses, since new classes and methods may be added at runtime.

This liability can turn into something of an asset. In an environment with dynamic compilation, we have the freedom to make assumptions based on the code that has been executed so far, and recompile when these assumptions are violated. Thus, if some method invocation `o.m2()` in a calling method `m1` can be bound to a single implementation among the classes loaded at the time `m1` is compiled, we might choose to inline this invocation *directly* (i.e., without a guard test), recording the fact that the compilation of `m1` depends on this assumption about `m2`. If some class that overrides `m2` is loaded later, and an instance of that class could become the receiver `o` in the invocation `o.m2()`, then we must recompile the caller `m1`.

There is a fly in the ointment here: what if an invocation of `m1` is being executed when the assumption about `m2` is violated? Concretely, consider `m1` of the form:

```
void m1() {
    while (true) {
        O o = getSomeO();
        o.m2();    // Call site is inlined.
    }
}
```

In the worst case, `getSomeO` could query the user for the name of a new subclass of `O`, load that class, and create and return an instance. Such a class may of course override `m2`, invalidating the inlining of `m2`.

In the Self system, which did much pioneering work in the field of dynamic compilation [16], this complication is dealt with by a mechanism called *on-stack replacement* [18]. In Self, there are *deoptimization points* within each method, at which the *source state* of the method, the state of the method's variables as defined by the interpretation of the source code, can be recovered from the *machine state* maintained by the compiled code. When a compilation assumption is violated, any method currently in progress must be at such a deoptimization point (or, in a multi-threaded system, reach one within bounded time). The Self system then recovers the source state, recompiles the method without the violated assumption, and also computes from the source state the corresponding machine state at the deoptimization point for the new compilation (which may, of course, have completely different machine state variables.) The new state replaces the old state for the method "on the stack."

There are several reasons to be cautious about such a system. In the Self implementation, the compiler produces voluminous data structures to enable deoptimization.<sup>1</sup> Deoptimization points also introduce constraints on code motion: for example, if a deoptimization point separates two source code writes,

<sup>1</sup> Though, in fairness, deoptimization was also used to enable debugging of optimized code, and was made possible more frequently than would be required for the purposes of this paper.

then they can't be reordered by a code scheduler. Finally, veterans of the Self project recount how much of the complexity of the system is related to deoptimization and reoptimization.

This brings us, finally, to the second contribution of this paper. We present a property of receiver expressions called *preexistence*. Virtual method invocations whose receivers have this property can be inlined directly, and no on-stack replacement mechanism is needed to handle invocations in progress when the assumptions on which the direct inlinings depend are broken. We present two static analysis techniques that can prove preexistence in a significant number of cases. We present measurements indicating the efficacy and costs of these analyses, and the speedups they obtain in an actual Java virtual machine implementation.

## 2 Related Work

Several techniques have been used to implement virtual calls. The original “Blue Book” implementation of Smalltalk [14] used a hash table mechanism to look up method names at runtime. Object-oriented languages with more constrained type systems were also developed; the type constraints allow more efficient virtual method invocation, using what are called *vtables* in C++ [26]. With this technique, each method is statically assigned an index that is constant across all classes that implement the method. A virtual call jumps indirectly to the code address in the vtable of its receiver object at the index corresponding to the invoked method.

Smalltalk systems pioneered dynamic compilation techniques [8], increasing the efficiency of execution. Thus, the relative cost of method lookup increased, so *inline cache* techniques were developed to speed up calls. An inline cache records the class of the last receiver object observed at the call site, and jumps directly to the implementation of the method for that class. A method prologue validates that the dynamic type of the receiver matches the expected type; if this test fails, a slower method lookup forwards the call and rebinds the call site cache to the class of the current receiver. Hölzle presents a generalized *polymorphic* form of inline caches [17]. It turns out that on some modern architectures the high costs of indirect calls make inline cache techniques attractive even for languages for which vtables could be used [10].

Several systems perform analyses to statically bind virtual calls, allowing them to be inlined or implemented with fast direct calls. Dean et al. use a *class hierarchy analysis* to statically bind virtual calls [7] in the Vortex system [6]. Class hierarchy analysis is a fairly inexpensive process, that, given a complete program, determines when the static type of a receiver implies that an invoked method has only a single implementation in the set of classes used in the program. More expensive *type flow* analyses attempt to tighten the static type constraint on the receiver object, ideally to a set small enough to determine the invoked method. Chambers et al. present such an analysis [1]. Fernandez [12] describes a link-time optimization system, and Diwan et al. [9] describe WPO, a “Whole Program Optimizer.” Both systems are for Modula-3 programs and perform both

of kinds of analyses. In addition, several languages, such as Trellis, Dylan [11], and the Java programming language, have linguistic mechanisms that allow the programmer to declare a class *sealed*, that is, ineligible for further subclassing. This enables static binding.

The Self system continues the Smalltalk research path of run-time compilation, adding more aggressive optimization, including extensive method inlining [16], whose correctness is often maintained by the previously mentioned on-stack replacement mechanism. Dean's thesis [4] describes several inlining techniques that use class-based guard tests to ensure correctness. These techniques include efficient subclass tests that offer some of the same benefits as the method test of the present paper. Also, Dean defines the concept of the set of classes sharing the same implementation of a method, but uses this concept only in static analyses, not dynamically to implement a guard test. Vitek, Krall, and Horspool [27] describe data structures that support time- and space-efficient implementations of subtype tests. Some of these techniques require potentially expensive recompilation when new classes are added to the system.

The current work uses contributions from these research directions. The system in which we did our experiments speeds virtual calls with inline caches. The work on different kinds of inlining guards extends the previous work on inlining in dynamic systems. The work on direct inlining of virtuals combines a new static analysis with dynamic recompilation in a novel way.

Another line of related work which the current paper does not explore is *specialization*. The idea is that methods of a class are cloned, so that if method `foo`, invoked on some object of class `C`, calls method `bar` on the same object, it may call the clone of method `bar` for `C` directly, rather than indirectly, and may possibly inline the call. In a variation, a compiler may introduce type tests early in a method, creating two versions of downstream code, one of which assumes the test. Plevyak and Chien [23] describe a whole-program analysis that employs specialization to improve the precision of type-flow analysis. This enables the computation of a precise global control-flow graph from which most method calls can be statically bound. Chambers and Ungar [2] describe the use of *customization* (i.e., specialization on the type of the receiver object) in an early version of Self. Another example of specialization is described by Dean et al. [5]. Specialization is an interesting area of research, but one where a number of competing concerns (execution speed vs. code size, for example) must be balanced, and we did not explore it.

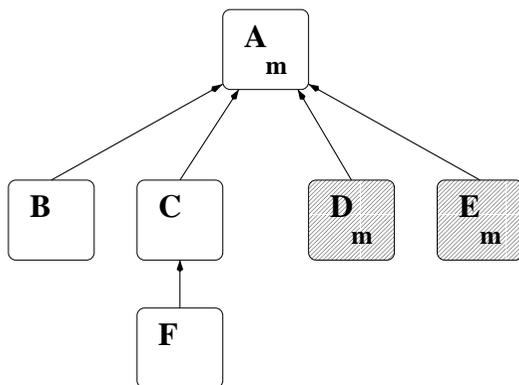
### 3 Method Tests and Class Tests.

As mentioned in the introduction, most systems that use virtual inlining have guarded the inlined code with a class test verifying that the receiver has a particular class. In pseudo-machine code, the code generated at such an inlined call site could be:

```

r0 := <receiver object>
r1 := load(r0 + <offset-of-class-in-object>)
if (r1 == <address-of-expected-class>) {
  <method inlining>
} else {
  r2 := load(r1 + <offset-of-method-in-class>);
  call r2
}

```



**Fig. 1.** An inheritance hierarchy

The basic problem with the class test is that it sometimes forces the non-inlined path to be taken unnecessarily. Assume we have the classes shown in figure 1. Class A defines a method *m*, classes B and C each extend A without overriding *m*, class F extends C still without overriding *m*, and classes D and E extend A and *do* override *m*. Consider a call site that invokes the method *m* on a receiver whose static type is A. We might wish to inline such an invocation. The inlined code for A.*m* is appropriate when the dynamic type of the receiver is any of A, B, C, or F, but incorrect if the dynamic type is D or E. A single class test will cover only one of the permitted classes A, B, C or F. One could convert the test into a disjunction of these possibilities, but only at some cost in speed and code density.

To solve this problem, we have invented an alternate test, called the *method test*, to guard inlined virtual methods. The class test imposes the reasonable requirement that each object contain a pointer to its class information. The method test imposes a further assumption, that the class information includes a *vtable*. A call site inlined with a method test guard will look like the following:

```

r0 := <receiver object>
r1 := load(r0 + <offset-of-class-in-object>)
r2 := load(r1 + <offset-of-method-in-class>)
if (r2 == <address-of-inlined-method>) {
    <method inlining>
} else {
    call r2
}

```

The method test obtains a pointer to the class information from the object, and then loads, from the class' vtable, the address of the code that would be invoked by an ordinary virtual call. The test compares this code address with the address of the method that was inlined. If they match, it is obviously permissible to execute the inlining. A method test can be used to good effect in the situation described previously. Suppose we decide to inline `A.m` because only classes `A`, `B`, and `C`, which share a single implementation of method `m`, are loaded when the caller is compiled. We can guard the inlining with a method test that covers all three receiver classes. If the overriding classes `D` and `E` are loaded later, the method test correctly chooses not to execute the inlined code for these receiver classes. Further, if the non-overriding class `F` is loaded later, the inlining *will* be executed for this receiver class. It is hard to see how this could be accomplished using class tests. Thus, the method test has two advantages over class tests: an *efficiency* advantage, in cases where one method test succinctly represents the results of several class tests, and a *robustness* advantage, in cases where a method test generated when only some of the classes in the program have been loaded stays effective as more classes are loaded.

The method test has another advantage in the particular case of the Java platform. It is a perhaps seldom-recognized property of the Java virtual machine's bytecode instruction set that a method invocation names the *definition* of the method that is invoked; the static type of the receiver expression, which may be more specific than the class that defines the method, is not apparent in the bytecodes. Consider the following situation:

```

class A {    void m() { ... }; }

class B extends A {                // Overrides m.
    void m() { ... };
}

class C extends A { ... }         // Does not override m

class X {
    void y(C c) { ... c.m(); ... }
}

```

The bytecodes of the method `X.y` indicate only that the invocation `c.m()` invokes the method `A.m`; the more specific type `C` of the receiver can be recovered through

an abstract interpretation similar to that performed in the bytecode verification process [19]. While it may be argued that excellent compilation of JVM bytecodes will have to recover this information, some JIT compilers will want to optimize speed of compilation over quality of code produced, and this analysis will impose some cost.

The missing type information is relevant to the comparison of method tests and class tests because it makes implementation of a useful class test problematical. In the situation above, use of a class test to guard an inlining of `c.m()` would be worse than useless unless we knew the most precise static type of the receiver: a test against `A` would always fail! In systems that do not implement a type-recovery pass, including the system we use, the method test may be used profitably where the class test cannot.

The imprecise type information may also lead to missed inlining opportunities. In the example above, if the compiler knows only that the receiver of `m` is an `A`, it would conclude that two methods might be invoked, and possibly reject the invocation `c.m()` as a candidate for inlining. However, if the precise static type of the receiver were known, the compiler would know that the invocation `c.m()` always invokes `A`'s definition of `m`, and would consider the call site a better candidate for inlining.

The method test has one obvious disadvantage when compared to the class test: an extra dependent load on the fast path. Our colleague Alex Garthwaite notes that instruction scheduling might ameliorate this problem. If several arguments to a method are being computed and moved into argument registers, the two loads required by the method test can often be separated.

### 3.1 Method and Class Test Measurements

We now present some measurements of the efficacy of virtual inlining in our system. Our experiments were performed in a version of the Sun Java<sup>®</sup>2 SDK for the Solaris<sup>™</sup> operating system. This environment includes a JIT compiler that performs inlining. Table 1 shows the programs for which we report measurements. For each benchmark, we report lines of source code (including comments and other non-code, excluding class libraries), and give a brief description. The first seven benchmarks constitute the SPECjvm98 suite [25], and the next three benchmarks were candidates considered for inclusion in SPECjvm98, but were not chosen for the final suite. The last two benchmarks are derived from real applications. The VolanoMark benchmark (**volano** in the tables) consists of a version of Volano LLC's VolanoChat internet chat server [20], and a client program that provides a simulated workload. We ran the client and server on the same machine, and added together measurements for both programs. The **portBOB** benchmark was created by IBM to predict the performance of object databases written in the Java language. We modified this program to make it run deterministically.

The measurements are performed on a Sun Ultra<sup>™</sup> 1 Creator 3D desktop, which has a 167 MHz UltraSPARC<sup>™</sup> processor and 512 megabytes of memory.

benchmark	lines of code	description
<b>mtrt</b>	3799	Multi-threaded image rendering
<b>jess</b>	10579	Version of NASA's CLIPS expert system shell
<b>db</b>	1028	Search and modify a database
<b>jack</b>	8194	Parser generator generating itself
<b>mpegaudio</b>	n/a	Decompress audio file
<b>javac</b>	25211	Source to bytecode compiler
<b>compress</b>	927	LZW compression and decompression
<b>richards</b>	3637	Threads running five versions of OS simulator
<b>tsgp</b>	894	Genetic program for traveling salesman problem
<b>si</b>	1707	Interpreter for a simple language
<b>volano</b>	13811	Internet chat server
<b>portBOB</b>	n/a	Transaction processing benchmark

**Table 1.** Descriptions of Benchmark Programs.

The compiler only inlines methods that have straight-line control flow, contain no exception handlers or `try...finally` constructs, and are shorter than some maximum length (15 bytecode instructions). Recursive inlining halves the length limit to bound code expansion. Inlining of virtual method invocations further requires the call site to be monomorphic with respect to the classes loaded at the time of compilation. We currently do not use on- or off-line profile information to determine when one method dominates at a polymorphic call site. We also do not inline invocations of methods of interface types, though this would be a straightforward extension.

We measured three system configurations for each program. The first, our baseline system, does no inlining of virtual methods. It uses an inline cache scheme to speed up virtual invocations, so this configuration is no straw man. The second configuration uses the method test. The third *hybrid* configuration combines method tests and class tests: a virtual invocation `o.m()` is inlined with a class test when the defining class of `m` is a *leaf class*, one with no subclasses (at the time of compilation), and with a method test otherwise. If no subclasses of the defining class are loaded later, the class test should be as accurate as a method test and somewhat cheaper. However, if subclasses are loaded later, the test may become inaccurate. This hybrid configuration gives us some efficiency/accuracy comparison of class and method tests.

For each benchmark program, table 2 details the number of virtual call sites inlined, the fraction that were inlined with a class test in hybrid mode, the number of executions of all inlined call sites, and the number of invocations that took the non-inline path using the pure method test and using the hybrid strategy. The benchmarks are ordered by number of executions of inlined virtual call sites.

We can make several remarks about the data in table 2. First, the number of inlined virtual sites varies considerably, by more than an order of magnitude, from program to program, but the number of executions of those sites varies

benchmark	inlined virtual call sites	hybrid class-test %	executions <i>millions</i>	method-test non-inline execs	hybrid non-inline execs
<b>mtrt</b>	781	30.9	235	0	0
<b>richards</b>	244	41.8	232	0	0
<b>portBOB</b>	359	39.0	18.7	0	2
<b>tsgp</b>	58	51.7	13.2	0	0
<b>jess</b>	211	46.9	8.8	0	0
<b>si</b>	63	54.0	6.3	0	0
<b>db</b>	69	37.7	5.8	0	0
<b>jack</b>	148	27.0	4.3	0	0
<b>mpegaudio</b>	73	31.5	3.2	0	0
<b>volano</b>	329	53.8	2.0	0	6
<b>javac</b>	315	19.0	1.8	412	412
<b>compress</b>	55	47.3	0.001	0	0

**Table 2.** Comparison of class and method tests for virtual inlining.

much more, by six orders of magnitude. The importance of the efficiency of virtual inlining tests is magnified by this second number, so we expect to see larger effects for the programs at the top of this table. Second, the fraction of call sites at which the class test is used in hybrid mode varies, averaging a little less than half of inlined call sites for these programs. Third, both kinds of tests are very accurate for these programs. The hybrid test failed a handful of times in **portBOB** and **volano**; the method test failed only in **javac**, and the hybrid test failed identically for that program.

Table 3 gives the performance results for the three configurations. For each benchmark and configuration we give user time and instruction count. The former is measured by the operating system, and the latter by accessing on-chip performance counters. For each of these measurements, we give the percentage difference with respect to no virtual inlining for the two virtual inlining modes. Finally, we also measure JIT compilation time and resulting code size. We omit compilation time for **volano**, since the server and client processes were running concurrently, distorting this measurement. The user time measurements should be treated with some skepticism, since they can vary significantly from run to run because of random factors such as instruction cache placement. Instruction counts are usually, if not always, a good predictor of elapsed time, and have the virtue of being highly repeatable, so that small differences indicate real effects.

The most important conclusion to draw from table 3 is that virtual inlining with either flavor of test is sometimes quite effective, and in no case decreases performance. Again, the benchmarks are ordered by frequency of execution of inlined virtual call sites. As one might expect, the performance improvement correlates well with this measure. Virtual inlining adds some compilation cost in most cases, but nothing we consider extreme. Similarly, virtual inlining adds a small increment in compiled code footprint. Since the previous measurements

indicated high accuracy for both tests, one would expect the hybrid test, which is less expensive when the class test is used, to perform better. This expectation is borne out by the measurements, but the magnitude of the difference is fairly small.

benchmark	virtual inlining	user time (sec)	percent diff.	instructions (millions)	percent diff.	compilation (ms)	code size (KBytes)
<b>mtrt</b>	none	52.8		5731		631	352
	method	43.4	-17.8	4595	-19.8	797	384
	hybrid	41.8	-20.8	4439	-22.5	749	382
<b>richards</b>	none	92.2		9645		846	377
	method	72.0	-21.9	9829	-8.4	688	371
	hybrid	69.6	-24.5	8697	-9.8	837	370
<b>portBOB</b>	none	76.5		6156		899	510
	method	76.9	0.5	6122	-0.6	946	515
	hybrid	76.6	0.1	6093	-1.0	930	514
<b>tsgp</b>	none	209.2		29408		434	244
	method	208.4	-0.4	29305	-0.3	432	246
	hybrid	208.1	-0.5	29279	-0.4	435	246
<b>jess</b>	none	39.1		4072		758	410
	method	36.4	-6.9	4034	-0.9	769	416
	hybrid	36.3	-7.2	4031	-1.0	768	416
<b>si</b>	none	93.0		9805		510	290
	method	90.1	-3.1	9776	-0.3	518	291
	hybrid	89.5	-3.8	9764	-0.4	512	290
<b>db</b>	none	144.8		8475		449	263
	method	142.3	-1.7	8473	0.0	453	264
	hybrid	141.8	-2.1	8473	0.0	454	264
<b>jack</b>	none	40.9		4161		793	470
	method	41.5	1.5	4141	-0.5	800	476
	hybrid	41.4	1.2	4141	-0.5	799	476
<b>mpegaudio</b>	none	100.7		13155		647	332
	method	106.3	5.6	13146	-0.1	633	333
	hybrid	108.6	7.8	13146	-0.1	632	333
<b>volano</b>	none	296.7		16861		n/a	756
	method	295.5	-0.7	16861	0.0	n/a	758
	hybrid	297.2	0.2	16849	-0.1	n/a	757
<b>javac</b>	none	70.5		8143		1424	793
	method	72.4	2.7	8129	-0.2	1453	815
	hybrid	72.0	2.1	8128	-0.2	1456	814
<b>compress</b>	none	81.4		10268		448	258
	method	77.4	-4.9	10267	0.0	450	258
	hybrid	77.5	-4.8	10268	0.0	450	258

**Table 3.** Performance comparison of class and method tests for virtual inlining

A question is raised by these measurements: if the hybrid test performs better than the method test, is the “method” part actually responsible for any benefit? That is, could we get the same performance improvements from a purely class-test-based system? We can run the system in two class test configurations to help answer this. In the first, *class-def*, we inline all virtual sites with a class test on the class that defines the method. This test will fail when the receiver class is a subclass of the defining class. In the second, *class-leaf*, we inline the virtual site with a class test only if the defining class has no subclasses (at the time of compilation). This test will fail less often, but fewer sites will be inlined.

We run these configurations on four of the benchmarks: **mtrt**, **richards**, and **portBOB**, because they are most affected by virtual inlining, and **javac**, which had some failed tests. Table 4 is similar to table 2. It records the number of call sites inlined, the number of executions of these call sites, and the percentage of failed tests for each configuration. We see that quite a substantial fraction of class tests fail when the class-def configuration is used, and many fewer sites are inlined when class-leaf is used. For concurrent programs such as **richards**, behavior can vary from run to run; for example, the number of inlined call sites differs slightly from the number in table 2.

benchmark	inlining mode	inlined virtual call sites	executions <i>millions</i>	failed tests
<b>mtrt</b>	class-def	781	243	27.1 %
	class-leaf	241	76.9	0.0%
<b>richards</b>	class-def	250	232	73.7%
	class-leaf	102	64.9	0.0%
<b>portBOB</b>	class-def	359	18.7	76.5%
	class-leaf	140	4.4	0.0%
<b>javac</b>	class-def	315	1.9	54.6%
	class-leaf	60	0.3	0.0%

**Table 4.** Two class tests for virtual inlining.

Table 5 is similar to table 3, and compares the performance of these two class test configurations with the baseline of no virtual inlining at all. For **mtrt**, the class-def configuration does almost as well as the method or hybrid tests (see table 3), but the class-leaf configuration does substantially worse. For **richards**, both configurations do poorly; in fact, class-def, because of the high failure rate of the test, does worse (in instruction counts) than no inlining at all.

These measurements show that class tests can obtain a significant fraction of the potential performance increase from inlining virtuals for some programs (like **mtrt**), but for others (like **richards**), the inaccuracy of such tests makes at least some method tests necessary to get significant benefit.

It might seem that the data we have presented argues unequivocally for the hybrid test. However, that is not so clear: whenever a call site is inlined with a class test, future class loading may render the test inaccurate in cases where the method test would continue to execute the inlined code.

## 4 Possible Benefit of Direct Virtual Inlining

The measurements in the last section indicate that inlining of selected virtual calls with a method test guard can have significant benefits for some programs. They also suggest an experiment. For all programs except **javac**, the method

benchmark	virtual inlining	user time ( <i>sec</i> )	percent diff.	instructions ( <i>millions</i> )	percent diff.
<b>mtrt</b>	none	52.6		5626	
	class-def	43.9	-16.5	4752	-15.5
	class-leaf	50.0	-4.9	5169	-8.1
<b>richards</b>	none	93.7		9644	
	class-def	88.3	-5.8	9741	1.0
	class-leaf	89.5	-4.5	9184	-4.8
<b>portBOB</b>	none	76.7		6173	
	class-def	75.6	-1.4	6161	-0.2
	class-leaf	75.0	-2.2	6107	-1.1
<b>javac</b>	none	70.9		8142	
	class-def	72.3	2.0	8149	0.1
	class-leaf	71.1	0.3	8146	0.0

**Table 5.** Performance comparison of class tests for virtual inlining

test never failed, so we altered the compiler to directly inline virtual calls. While incorrect in general, this altered compiler produces code that executes correctly for these programs. This experiment determines an upper bound on how much further efficiency can be gained by schemes that inline virtual invocations directly. Table 6 shows the results of this experiment, for the four benchmarks out of the set where direct inlining results in a noticeable improvement. We ran each benchmark twice, once using the method test, and once inlining virtual calls directly. For each run, we show user time, instruction count, and percentage differences between the two runs.

benchmark	virtual inlining	user time ( <i>sec</i> )	percent diff.	instructions ( <i>millions</i> )	percent diff.
<b>mtrt</b>	method	43.7		4693	
	direct	27.5	-38.7	2280	-51.4
<b>richards</b>	method	71.4		8829	
	direct	62.1	-13.0	6838	-22.6
<b>portBOB</b>	method	76.0		6095	
	direct	73.1	-3.8	5944	-2.5
<b>jess</b>	method	36.5		4034	
	direct	37.0	1.4	3956	-1.9

**Table 6.** Potential benefit of direct virtual inlining

The comparison in table 6 shows that there are programs for which direct virtual inlining yields significantly better performance. Direct inlining both eliminates the overhead of a guard test and enables additional optimizations because

the inlined code can be assumed always executed, instead of just possibly executed.

## 5 Avoiding On-Stack Replacement: Preexistence

Section 4 presented an experiment that quantified the potential benefits of direct inlining of virtual calls, using a system known to be incorrect in general but safely applicable to the particular programs tested. In the next sections, we discuss techniques for safe direct inlining of virtual calls, without requiring an on-stack replacement mechanism.

Assume that we only inline virtual invocations directly when the called method has only a single implementation at the time of compilation of the caller. Assume further that we record a dependency of the caller on this *single-implementation assumption*, as is done in Self. It is easy to guarantee that any such assumptions made by a method are true on entry to the method—we simply recompile the method without performing the direct inlining, and thus without the assumption, before we allow the assumption to be violated. In the case of direct inlining, the associated single-implementation assumption is violated by the loading of a class that provides a second implementation of the inlined method. Thus, we augment the class loading process to detect violations of single-implementation assumptions, and to recompile the effected methods, or revert them to interpretation, before making the loaded class available.

The problem remains that even when a single-implementation assumption is satisfied on entry to a method, it could become invalid during the method's execution. This problem prompted the invention of on-stack replacement. Now we propose an alternative solution.

Consider a method `m1` that contains a method invocation `o.m2()`. We say that the receiver expression `o` is *preexisting in m1* when the object denoted by `o` was allocated before execution of the calling method `m1` began. (When `m1` is clear from context, we will simply say that `o` is preexisting.) Because `o` is allocated prior to the start of an execution of `m1`, the dynamic type of `o` is clearly in the set  $S$  of classes extant in the system at the start of `m1`. If a single-implementation assumption about `m2` is true over the classes in  $S$ , and the dynamic type of `o` is in  $S$ , then even as the total set of classes in the system grows, the type of the preexisting receiver `o` remains in the set  $S$  for the duration of the execution of `m1`. Thus, if `m2` has a single implementation in  $S$ , it can be inlined directly without a guard test.

If we directly inline virtual calls only when the receiver is preexisting, then when some class loading operation provides a second implementation of `m2` that necessitates a recompilation of `m1`, as described above, we can allow any in-progress invocations of the original compilation of `m1` to continue to execute the original code without risking incorrect behavior.

## 6 Proving Preexistence

The concept of preexistence is useful only if it is possible to prove that receiver expressions denote preexisting objects at a significant fraction of call sites, and do so at low cost (always an issue in dynamic compilation systems). Our experience shows that this is possible. We present two analyses for proving preexistence. The first is quite simple, and gets a surprisingly large fraction of the possible benefit. The second is more complicated, and adds little benefit in the programs we tried; still, it illustrates the generality of the approach, and it may be that other programs benefit significantly.

### 6.1 Invariant Argument Analysis

The first technique, *invariant argument analysis*, examines a method to identify input arguments to which no assignments are made. Clearly, such constant arguments refer to objects that were allocated before the method began executing, i.e., are preexisting. A slightly more ambitious analysis tracks the values of such arguments, rather than simply pattern matching on their textual occurrences, to prove that the receiver expression in

```
void m1(Foo f) {
    Foo f2;
    ...
    f2 = f;
    f2.m2();    // f2 is preexisting.
}
```

is preexisting.<sup>2</sup>

In many object-oriented languages, the receiver argument of a method is passed with a different syntax than is used for other arguments to the method, e.g., `o.m(...)` instead of `m(o, ...)`, and members of the current object may be accessed with a syntax different from that used to access members of other objects. All such syntactic sugars are irrelevant to this analysis.

### 6.2 Effectiveness of Invariant Argument Analysis

We augmented the compiler in our system to prove preexistence using invariant argument analysis. As before, a virtual call site with a single implementation may be inlined, but now, when the receiver is preexisting, the call is inlined directly. The compiler records a dependency linking the validity of the compiled code

<sup>2</sup> Note that this observation could be extended to the `clone` operation of the Java programming language: if we clone an object we can prove preexisting through other means, we obtain an object which, while *not* preexisting as defined in this paper, still has a dynamic type that is a member of the set of classes extant at the start of the calling method, which is what direct inlining requires.

to the assumption of single implementation of the called method, as discussed previously.

Table 7 shows the fraction of inlined virtual call sites at which invariant argument analysis proves the receiver to be preexisting, as well as the number of recompilations caused by dependency violations over the course of the run. Table 8 compares the performance of the system using this analysis with method-test virtual inlining. In both tables, we present results for the four benchmarks of table 6, the ones for which direct virtual inlining has a potential benefit. We also include **javac**, which, because it had method test failures, could not be included in table 6. Finally, table 7 includes a run of the HotJava<sup>TM</sup> browser accessing the ECOOP home page. Unlike the SPEC benchmarks, this is an interactive program, and compiles considerably more code. The interactive nature of this program complicates performance measurements, but we include it here to see if recompilation will be a more significant issue for interactive programs, which use the deeper class hierarchies common to user-interface toolkits.

benchmark	inlined virtual call sites	sites with preexisting receiver	percentage	recompilations
<b>mtrt</b>	784	324	41.3	0
<b>richards</b>	246	154	62.5	0
<b>portBOB</b>	359	166	46.2	0
<b>jess</b>	211	54	25.6	0
<b>javac</b>	323	218	67.5	1
<b>hotjava</b>	1548	723	46.7	40

**Table 7.** Effectiveness of invariant argument analysis, by call sites

benchmark	virtual inlining	user time (sec)	percent diff.	instructions (millions)	percent diff.	compilation (ms)	code size (KBytes)
<b>mtrt</b>	method	43.6		4697		893	383
	preexist	32.2	-26.1	3227	-31.3	684	364
<b>richards</b>	method	72.0		8829		795	371
	preexist	62.4	-13.3	7279	-17.6	619	362
<b>portBOB</b>	method	76.7		6115		983	515
	preexist	75.4	-1.7	5978	-2.2	920	505
<b>jess</b>	method	36.5		4034		810	417
	preexist	37.7	3.3	4010	-0.6	775	413
<b>javac</b>	method	72.2		8129		1461	815
	preexist	72.6	0.6	8114	-0.2	1469	801

**Table 8.** Performance of direct virtual inlining via invariant argument analysis

Table 7 shows that, for these programs, approximately half of virtual call sites are proven to have preexisting receivers by invariant argument analysis.

(The average over the complete benchmark set is 40.9%.) The **javac** benchmark is the only SPEC benchmark that performs recompilations, and performs just one. The HotJava browser performed 40 recompilations, but it compiled 2269 methods in total, so this is a relatively small fraction.

A comparison of table 6 and table 8 shows that a surprisingly large fraction of the potential performance gains from direct virtual inlining can be obtained by using invariant argument analysis to prove receiver object preexistence. In the case of **mtrt**, we obtained (roughly) a 30% speedup out of a possible 50%. Thus, with the caveat that we are extrapolating from a small number of benchmarks, it seems possible to obtain many the benefits of direct virtual inlining without making “closed-world” assumptions, and without implementing an on-stack replacement mechanism. Further, the dependency mechanism does not seem to trigger an excessive number of recompilations.

### 6.3 Immutable Field Analysis

To understand the second technique for proving preexistence, consider a virtual method invocation of the form  $o.f.m()$ . That is, the receiver expression is of the form  $o.f$ , where  $o$  is an expression of static type  $O$ , and  $f$  is a field of that class. Call a field  $f$  *immutable* if it is assigned to only in constructors. If  $f$  is proven immutable, and one has access to a fully-constructed instance of a class in which the field  $f$  appears, then one can assume that  $f$  will not change subsequently.<sup>3</sup> Consequently, preexistence of  $o.f$  follows from preexistence of  $o$  and immutability of  $f$ . (We assume that “ $o$  is preexisting” is taken to mean not only that  $o$  is allocated, but also that it is constructed—this is an easy consequence of the rules of the C++ and Java programming languages.)

In our implementation, we attempt to prove immutability only of private fields of classes, fields that cannot be accessed by methods of other classes. In this case, the analysis is easy and efficient: each method of the class that declares the private field is searched for assignments to that field; the field is immutable if such assignments are found only in constructors. (A refinement of this analysis identifies private methods that are called only from constructors, and treats field assignments in such methods as if they occur in constructors.)

It is easy to see why immutable field analysis for non-private fields would not be useful for proving preexistence. The proof that the preexistence of  $o.f$  follows from the preexistence of  $o$  and the immutability of  $f$  requires a *proof* of both of the latter properties, not an *assumption* that might later be violated. In systems that cannot make closed-world assumptions, even if one examines all the classes extant to determine that a non-private field  $f$  is immutable “so far,” a class might be loaded later that writes to  $f$ . Thus, even if an expression  $o$  is preexisting and a non-private field  $f$  is immutable “so far,” modifications to  $f$  by methods loaded later may make  $o.f$  denote a non-preexisting value.

It is sometimes suggested that the reflection API of the Java platform invalidates this analysis, by allowing classes to modify private fields of other classes.

<sup>3</sup> At least in languages in which constructors are executed only once on a given object.

We believe that the most current specification of the reflection API allows an implementation to prevent such modifications by enforcing access constraints. An implementation whose compiler uses immutable field analysis would have to implement reflection in this manner. The Java Native Interface, on the other hand, does allow such uncontrolled updating. Our view is that since native code can do *anything*, authors of native code are on their honor to respect access constraints. Even if native code respects privacy, the inability to analyze native code requires immutable field analysis to assume that native methods update all private fields of their class.

#### 6.4 Effectiveness of Immutable Field Analysis

We have implemented immutable field analysis, but will only summarize performance results. There are a noticeable number of call sites at which immutable field analysis can prove preexistence but invariant argument analysis cannot. The fraction of such call sites varies from 1.2% to 21.3% of all inlined virtual call sites, averaging 9.5%. However, this increase in the number of call sites directly inlined does not translate, at least for these benchmarks, into significantly better performance. The **mtrt** benchmark's instruction count decreases by 1.4%, and **portBOB**'s by 0.4%, but no other benchmark's performance changes appreciably. Compilation time does not increase noticeably from the cost of the analysis. Despite these somewhat disappointing end-to-end results, the increase in the fraction of call sites inlined causes us to believe that immutable field analysis may be important for some programs; if immutable field analysis enables direct inlining in a frequently-executed inner loop of some program, it could have a large effect on that program.

## 7 Conclusions and Future Work

There are two main contributions in this paper. First, we presented an alternative test for guarding inlining of virtual invocations. This method test is less efficient than a class test when there is only one possible receiver type, but more efficient when several possible receiver types can be covered in a single test. It may also be more robust than class tests, since a method test allows execution of inlined code even for receiver types that weren't loaded when the test was generated. Finally, method tests may be more convenient to generate in some systems, such as just-in-time compilers for Java virtual machines.

The second contribution allows the direct inlining of some virtual calls without an on-stack replacement mechanism. The scheme restricts direct inlining to call sites whose receivers have a preexistence property, and records single-implementation assumptions on which correctness of the inlining depends. Callers must be recompiled when these assumptions are invalidated, but currently-executing invocations of such callers may continue, by virtue of the preexistence restriction. We described two analyses, with different cost/precision tradeoffs, that prove preexistence.

We presented several measurements. Even though the base system against which comparisons were made included an inline cache mechanism to speed up virtual calls, inlining of virtuals with a method test guard offers significant improvements for programs that execute inlined virtual calls frequently. We determined an upper bound on the speedup available by inlining all eligible virtual calls directly. Our simplest analysis, invariant argument analysis, allowed sufficient direct inlining to realize a large fraction of this potential speedup. Immutable field analysis increased the fraction of call sites inlined directly, but did not improve the overall performance significantly for the programs measured.

In future work, we hope to eliminate some of the restrictions our compiler imposes on what may be inlined. The resulting increase in inlining opportunities may extend the benefits of virtual inlining to more programs. We also hope to investigate the application of preexistence to other assumptions that compilers may make to enable optimizations.

## Acknowledgments

We would like to thank Christine Flood and Steve Heller for careful readings of the paper. Craig Chambers and Jeff Dean helped clarify the relationship between their work and our's. Thanks also to the anonymous referees for helpful comments.

## References

1. Craig Chambers, David Grove, Greg DeFouw, and Jeffrey Dean. Call graph construction in object-oriented languages. *ACM SIGPLAN Notices*, 32(10):108–124, October 1997.
2. Craig Chambers and David Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In Bruce Knobe, editor, *Proceedings of the SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 146–160, Portland, OR, USA, June 1989. ACM Press.
3. Ole-Johan Dahl, Bjørn Myrhaug, and Kristen Nygaard. Simula common base language. Technical Report S-22, Norwegian Computing Center, Oslo, Norway, 1970.
4. Jeffrey Dean. *Whole-Program Optimization of Object-Oriented Languages*. PhD thesis, University of Washington, Seattle, Washington, 1996.
5. Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. *ACM SIGPLAN Notices*, 30(6):93–102, June 1995.
6. Jeffrey Dean, Greg DeFouw, David Grove, Vassily Livinov, and Craig Chambers. Vortex: an optimizing compiler for object-oriented languages. *ACM SIGPLAN Notices*, 31(10):83–100, October 1996.
7. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Walter G. Olthoff, editor, *Proceedings of the Ninth European Conference on Object-Oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, Århus, Denmark, 7–11 August 1995. Springer-Verlag.

8. L. Peter Deutsch and Allan Schiffman. Efficient implementation of a Smalltalk-80 system. In *Proceedings of the 11th Symposium on the Principles of Programming Languages*, pages 297–302, Salt Lake City, 1984. ACM SIGPLAN.
9. Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *Proceedings of the 1996 Conference on Object-Oriented Programs, Systems, Languages, and Applications*, pages 292–305. ACM SIGPLAN, October 1996.
10. Karel Driesen and Urs Hölzle. Accurate indirect branch prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, volume 26,3 of *ACM Computer Architecture News*, pages 167–178, New York, June 27–July 1 1998. ACM Press.
11. Neal Feinberg, Sonya E. Keene, Robert O. Mathews, and P. Tucker Withington. *The Dylan Programming Book*. Addison-Wesley Longman, Reading, Mass., 1997.
12. Mary F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. *ACM SIGPLAN Notices*, 30(6):103–115, June 1995.
13. Richard Gabriel, Jon White, and Daniel Bobrow. CLOS: Integrating object-oriented and functional programming. *CACM: Communications of the ACM*, 34, 1991.
14. Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.
15. James Gosling, Bill Joy, and Guy Steele. *The Java™ Language Specification*. The Java Series. Addison-Wesley, 1.0 edition, August 1996.
16. Urs Hölzle. Adaptive optimization for SELF: Reconciling high performance with exploratory programming. Ph.D. Thesis CS-TR-94-1520, Stanford University, Department of Computer Science, August 1994.
17. Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In P. America, editor, *Proceedings of the 1991 European Conference on Object-oriented Programming*, LNCS 512, pages 21–38, Geneva, Switzerland, July 1991. Springer-Verlag.
18. Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In Christopher W. Fraser, editor, *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 32–43, San Francisco, CA, June 1992. ACM Press.
19. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1996.
20. Volano LLC. Volanomark benchmark. <http://www.volano.com/benchmarks.html>, Mar. 1999.
21. Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
22. Greg Nelson, editor. *Systems Programming in Modula-3*. Prentice-Hall, Englewood Cliffs, NJ, 1991.
23. John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–324, October 1994.
24. Craig Schaffert, Topher Cooper, and Carrie Wilpolt. Trellis object-based environment language reference manual. DEC-TR 372, Digital Equipment Corp., Object-Based Systems Group, Hudson, Massachusetts, Nov. 1985.
25. SPEC. SPECjvm98 benchmarks. <http://www.spec.org/osg/jvm98>, August 1998.
26. Bjarne Stroustrup. *The C++ Programming Language: Second Edition*. Addison-Wesley, Reading, Massachusetts, 1991.
27. Jan Vitek, Nigel R. Horspool, and Andreas Krall. Efficient type inclusion tests. In *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages, and Applications*, Atlanta, GA, October 1997. ACM Press.