RC 22261 (99194) November 28, 2001
Computer Science

IBM Research Report

Efficient Implementation of Java Interfaces: invokeinterface Considered Harmless

Bowen Alpern, Anthony Cocchi, Stephen J. Fink, David P. Grove, Derek Lieber
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

Efficient Implementation of Java Interfaces:

invokeinterface Considered Harmless

Bowen Alpern Anthony Cocchi Stephen Fink David Grove
Derek Lieber
IBM T.J. Watson Research Center
{alpern@watson, tony@watson, sjfink@us, groved@us, derek@watson}.ibm.com

Abstract

Single superclass inheritance enables simple and efficient table-driven virtual method dispatch. However, virtual method table dispatch does not handle multiple inheritance and interfaces. This complication has led to a widespread misimpression that interface method dispatch is inherently inefficient. This paper argues that with proper implementation techniques, interface method calls need *not* introduce significant performance degradation as compared to virtual method calls.

We present an efficient interface method dispatch mechanism, associating a fixed-sized *interface method table* (IMT) with each class. Interface method signatures hash to an IMT slot, with any hashing collisions handled by custom-generated conflict resolution stubs. The dispatch mechanism is efficient in both time and space. Furthermore, with static analysis and online profile data, an optimizing compiler can inline the dominant target(s) of any frequently executed interface call.

Micro-benchmark results demonstrate that the expected cost of an interface method call dispatched via an IMT is comparable to the cost of a virtual method call. Experimental evaluation of the techniques on a suite of larger applications demonstrates that, even for applications that make only moderate use of interface methods, these techniques can significantly impact bottom line performance.

1 Introduction

Multiple inheritance adds power, expressiveness, and perhaps complexity and controversy to an object-oriented programming model. Whether multiple inheritance simplifies or complicates the programming model remains a matter of debate. The designers of Java opted to avoid potential problems by providing only a limited form of multiple inheritance, with the `interface` construct. Java allows only single superclass inheritance; a class can inherit method implementations from at most one direct superclass. However, a class may *implement* any number of interfaces. Each class must explicitly provide implementations of the method signatures declared by its interfaces.

Single inheritance enables simple and efficient virtual method dispatch using *virtual method tables (VMTs)*. However, a single VMT cannot support interface method dispatch due to potential multiple inheritance. This has led to a widespread impression that interface method dispatch in Java is inherently inefficient. A naïve interface dispatch mechanism can indeed introduce tremendous overhead. Vallee-Rai reported, for example, that the Kaffe JIT Compiler `invokeinterface` bytecode costs approximately 50 times an `invokevirtual` [42]. The initial implementation of interface invocation in the Jalapeño JVM performed similarly poorly (see section 6 below).

This paper demonstrates techniques to implement Java interface method calls efficiently in both time and space. These techniques have been implemented in the Jalapeño virtual machine. Ex-

perimental results demonstrate that these techniques introduce negligible overhead for interface methods, compared to the cost of virtual methods.

The paper identifies three sources of potential inefficiency with interface methods: dynamic type checks, method dispatch, and inhibition of compiler optimizations. Section 2 reviews the semantics of Java interfaces, elucidating the dynamic type check requirements imposed by the Java virtual machine specification. Section 3 reviews the Jalapeño virtual machine, including its mechanism for quickly determining if a class implements an interface.

Interface method dispatch overhead represents the second potential source of inefficiency. Section 4 presents Jalapeño's scheme for interface method dispatch. The JVM associates a small, fixed-sized *interface method table* (IMT) with each class. The system hashes each interface method signature to an IMT slot, with hash collisions handled by custom-generated conflict resolution stubs. In the usual case (no collision), the runtime cost of a call through an interface is almost identical to a virtual method call. An IMT collision adds a little additional overhead, on the same order as a method prologue/epilogue sequence.

Barriers to compiler optimization represent the third potential source of inefficiency from interfaces; interface calls might hamper inlining or force extra run-time tests to guard inlined method bodies. Section 5 describes Jalapeño's mechanisms for inlining interface calls. Jalapeño's adaptive optimization system uses the same criteria for inlining interface method calls as it does for virtual calls. Moreover, the compiler usually employs the same run-time checks to guard both types of inlined method bodies. This section also shows how an optimizing compiler can often eliminate the dynamic type check imposed by the `invokeinterface` bytecode.

Section 6 presents experimental results evaluating performance with these techniques. Microbenchmark results verify that IMT-based interface method dispatch is not significantly more expensive than virtual method dispatch. Results with a suite of larger applications demonstrate the efficacy of the various techniques, and illustrate that direct and indirect costs of interface invocation can significantly impact overall performance.

Section 7 discusses other schemes used to dispatch methods efficiently in the presence of multiple inheritance or dynamic typing, and presents possible future improvements to Jalapeño's implementation.

2 Java Interfaces

The Java interface construct provides a limited form of multiple inheritance [20]. A Java interface is a type whose members are all either `abstract` methods or constants. A proper Java class may `implement` zero or more interfaces, while it `extends` exactly one class, `Object` notwithstanding. Additionally, an interface can extend other interfaces.

Most JVM implementations provide virtual method dispatch off a table. Each class has a *virtual method table* (VMT) which holds a reference to the implementation of each method declared by the class. When a class is loaded, the JVM assigns each virtual method in the class a unique offset in the virtual method table. Methods inherited from a superclass retain the unique offset assigned by the superclass. So, each new class that extends a superclass inherits the superclass's virtual method table and offsets. If a class overrides an inherited method, it simply overwrites the entry in the virtual method table at the original method's offset. The net result is that given a method `foo()` of class `A`, a reference to a suitable `foo()` method resides at the same offset in the virtual method table of class `A` and any of its subclasses.

Interfaces and multiple inheritance preclude this dispatch mechanism. Suppose `foo()` is an abstract method of an interface `I`. If classes `A` and `B` both implement `I`, then `A` and `B` must each have a suitable `foo()` method. However, in general, `A.foo()` and `B.foo()` will not map to the same VMT offset.

The interface method dispatch bytecode, `invokeinterface`, entails a greater runtime verifica-

tion burden than does its virtual counterpart, `invokevirtual`. The first time the latter bytecode is executed, it may force the specified class to be loaded (with all of the potential for raising exceptions that this may entail). Thereafter, the JVM verifier guarantees that receiver object for the virtual method will have a suitable method at the appropriate slot in its VMT. However, the verifier allows an `invokeinterface` call to an object of a class that does not actually implement the interface. Should this happen, the JVM must throw an `IncompatibleClassChangeError`.[1] Notice however, that after a class has been found to successfully implement an interface, the class will always implement the interface.

Any implementation of interface dispatch in Java should not compromise other optimizations enabled by Java's simple object model. For example, Jalapeño exploits a two-word object header for fast synchronization, hash codes, and garbage collection. For this reason, some multiple inheritance dispatch mechanisms employed for statically typed languages with more complex object models (notably C++ [38]), do not solve the interface problem for Java.

3 The Jalapeño JVM

Jalapeño [1] is a research Java virtual machine targeting server applications. It is written in Java [2]. In addition to providing a high-level strongly-typed development environment, this design decision allows techniques, such as those described in this paper, to apply not only to application code, but also to the JVM itself, including its compilers, thread scheduler, garbage collector, and adaptive optimization system.

Jalapeño employs a compile-only strategy; it compiles all methods to native code before they execute. The *baseline* compiler produces poor quality code, but generates it quickly. The *optimizing* compiler provides several levels of optimization. All optimizations levels include linear scan register allocation [35] and BURS-based instruction selection. The lowest optimization level (level 0) consists mainly of a set of on-the-fly optimizations performed during IR generation. Optimization level 1 augments level 0 with aggressive inlining (driven by both static heuristics and online profile information) and a number of other local and global flow-insensitive optimizations. Optimization level 2 augments level 1 with a suite of global SSA-based optimizations.

Jalapeño's adaptive optimization system [6] maintains statistical samples of the dynamic call graph. Using this information it can schedule frequently called and/or computationally intensive methods for recompilation at an appropriate level of optimization. This information is also used to inform inlining decisions.

Jalapeño supports a variety of configurations. For simplicity of exposition this paper assumes the following configuration. The JVM runs on a PowerPC-based SMP running the AIX operating system. It uses a parallel, nongenerational copying garbage collector. The optimizing compiler statically compiles the methods of system classes (at optimization level 2), as part of Jalapeño's boot image. The baseline compiler initially compiles each application method just before it executes for the first time. The adaptive optimization system causes hot methods to get recompiled for improved performance.

Objects in Jalapeño each have a two word header. The first header word points to a *Type Information Block* (TIB) for the type of the object. The TIB consists of an array of objects. Its first entry holds a reference to an object that describes the type. The TIB contains the type's VMT. It also contains a fixed-size *Interface Method Table* (IMT), described in the next section.

[1] This error is specified as a *Runtime Exception* in the second edition of the virtual machine specification [32], but not in the first edition [31]. A source-to-bytecode compiler would refuse to compile such a program, but if one file were modified after an initial compilation, subsequent compilation of the file could create the offending class files (hence the name of the error). Something similar could happen with `invokevirtual`, but, in that case, the incompatibility could be detected when the class was loaded. Since the interfaces a class implements are *not* loaded with the class itself, and since interfaces can extend other interfaces, the fact that a class does, or does not, implement an interface cannot be determined until the first time an instance of the class is tested against an interface.

```
    //    t0  contains the address of the receiving object ("this" parameter)
    //    s1  contains the interface method signature id ("hidden" parameter)
    //    LR  contains the return address in the caller
    //
    L     s0, tibOffset(t0); // s0 := TIB of the receiver
    CMPI  s1, id1;           // compare hidden parameter to id of first method
    BNE   l1;                // if not equal, skip
    L     r0, offset1, s0    // load VMT entry for first method into register 0
    MTCTR r0                 // move this address to the count register
    BCTR                     // branch to it (preserving contents of the LR)
l1: CMPI  s1, id2;           // compare hidden parameter to id of second method
    BNE   l2;                // if not equal, skip
    L     r0, offset2, s0    // load VMT entry for second method into reg. 0
    MTCTR r0                 // move this address to the count register
    BCTR                     // branch to it (preserve contents of the LR)
l2: TI    31, 31, 0xFFFF     // trap if neither matches (this shouldn't happen)
```

Figure 1: A conflict resolution stub with two entries.

An additional three slots in the TIB are used to speed dynamic type checking [3]. As discussed in the previous section, the test that the class of an object implements an interface contributes to the overhead of using interface methods. One of these three TIB slots points to a data structure — an array of bytes called an *Implements Trits Vector* (ITV) — that allows Jalapeño to answer just such questions quickly. Each interface is assigned unique integer index into the ITV. Consider the ITV for a class C, supposing interface I has been assigned ITV index n. The value of C's ITV entry at index n caches the result of a test that C implements I. This ITV entry holds 0 if C is known to *not* implement the I, 1 if C *does* implements I, and 2 if the test has not yet been made.

As there is no *a priori* bound on the number of interfaces that a JVM may encounter during its execution, the JVM must have the ability to grow the ITVs. To this end, the implementation logically partitions the ITV into two sections. The first section does not require an array bounds check, while the second section requires a check in case the ITV in question needs to be extended. Those interfaces with indices less than the initial size of all ITVs never require a bounds check.

In any event, the first test that a class implements an interface is moderately expensive. However, subsequent tests for the same class and interface obtain the cached result of the first test from the class's ITV fairly cheaply.

4 Efficent Interface Dispatch

Jalapeño dispatches virtual methods efficiently, because the VMT offset of the target method is available as a compile-time constant when the `invokevirtual` bytecode is compiled. Jalapeño devotes a fixed-sized portion of the TIB, called the *interface method table* (IMT), to allow interface method dispatch with similar efficiency.

Every Java method has a *signature* — its name, the types of its parameters, and its return type (if any). All methods that implement a particular interface method signature can be reached through the same slot in their class's IMT. Every interface method signature is assigned a unique *id*. Each id is hashed into an *IMT offset*. The system assigns interface method ids sequentially as new interface method signatures are discovered, either by loading interfaces or by compiling references to interface methods. In the current implementation, the system maps ids directly to IMT slots, modulo the size of the IMT, a fixed constant.

Ideally, the interface dispatch sequence would proceed as follows. At compile-time, the compiler

would identify the IMT offset corresponding to the callee's signature. At run-time, the system would receive the contents of this IMT offset for the receiver object's class. This IMT entry would hold a reference to the executable code for the callee, and the system would branch to this executable code (leaving the caller's return address in the `LR` register of the PowerPC's CPU).

For this ideal scheme to work without refinement, the IMT would have to be large enough to accommodate each interface method signature having a distinct IMT offset. Even though the mechanism requires only one IMT per class, the prohibitive space overhead renders this solution unacceptable.

Therefore, Jalapeño instead hashes each interface method signature id to an IMT slot, tolerating an occasional collision to obtain small, fixed-size IMTs. Consequently, some classes may have two, or more, interface methods with the same IMT offset. Such interface methods are said to *conflict* or *collide*. Jalapeño must discriminate between such methods at runtime.

Conflicts are handled as follows. Before Jalapeño jumps to the code referenced by an IMT entry, it loads the interface method signature into a *hidden parameter*. This *hidden parameter* will be used disambiguate interface method in case of IMT conflict. Where there is no conflict (that is, the IMT entry points directly to the target executable code), the hidden parameter is ignored.

When two or more methods collide in the IMT, the IMT slot points to a custom *conflict resolution stub* (see figure 1). This code successively compares the interface method signature id passed in hidden parameter to each id of the signatures of the interface method that shares this slot. When the stub finds a match, it loads the VMT offset for the appropriate method, and transfers control to the code referenced at this offset.

Conflict stub code generation must proceed carefully, due to the restricted context in which the conflict resolution stub must execute. Before calling the stub, the calling sequence has already stored the caller's return address and call parameters in the locations dictated by the calling convention. The conflict resolution stub must respect the calling convention register conventions, and thus may use only a small number of registers without introducing save/restore overhead.

Figure 1 shows a conflict resolution stub for an IMT slot with two possible target methods. The processor's link register contains the return address in the calling method. The non-volatile registers cannot be used until the callee saves them. The volatile registers cannot be used because they may contain parameters to the method being called. Only Jalapeño's three PowerPC scratch (`s0`, `s1`, and `r0`) are readily available for use by the stub.[2] Although the details of the stub are highly architecture dependent, similar ideas apply on other platforms.[3]

It remains to explain how the virtual machine populates IMTs. Ideally, Jalapeño would create the IMT for a class when the class is loaded. Unfortunately, since a class's interfaces are not loaded with the class, it does not know at class load time which of the public virtual methods *are* interface methods. It could conservatively assume that all such methods are interface methods, but this would lead to excessive false IMT conflicts.

Instead Jalapeño builds the IMTs incrementally as the program runs. When the virtual machine discovers that a class implements an interface, it adds that interface's methods to the class's IMT. If this process reveals an IMT conflict, the system dynamically generates and/or extends the appropriate conflict resolution stubs. Since the virtual machine must always perform the relevant type check before the interface method dispatch (either at runtime or at compile time, as discussed in the next section), the IMT will always contain the required methods by the time of invocation.

[2]These registers are used by Jalapeño method prologues (and epilogues) to allocate (free) the stack frame for the called method. They are also used as temporary registers between method calls. Register 0 (`r0`) is of limited utility since many PowerPC instructions treat what would be a reference to it as a literal 0.

[3]Jalapeño's register conventions for Intel's IA32 architecture do not provide a free scratch register for the hidden parameter, so the interface method signature id is passed by storing it at a prearranged location in thread-specific memory.

5 Inlining Interface Invocations

The previous section described an efficient scheme for interface method dispatch. However, interfaces might also inhibit compiler optimizations, method inlining in particular. This section discusses optimizations performed by the Jalapeño optimizing compiler to inline interface calls and further reduce the costs of dynamic dispatching.

```
interface I { public void foo(); }

class B implements I {
  void foo() {...}
}

class C extends B {
}

class A {
  void bar(I i, I i2) {
    if (I instanceof B) {
      i.foo();
    } else {
      i2.foo();
    }
    I i3 = (I) new B();
    i3.foo();
  }
}
```

Figure 2: Some example interface usage patterns.

Devirtualization is a well-known technique that converts a virtual dispatch to a statically-bound (direct) call when the target of the dispatch can be uniquely determined at compile time. Similarly, the Jalapeño optimizing compiler performs *virtualization*, reducing an interface invocation to a virtual method call.

Consider, for example, the code for method `A.bar()` in Figure 2. This code can be transformed as follows:

1. The compiler can virtualize the call to `i.foo()`, since it can determine that at that program point, `i` must be a sub-class of `B`. Therefore, `i.foo` can be dispatched as a virtual method call to `B.foo()`.

2. The compiler cannot virtualize the call to `i2.foo()`, lacking any conclusive information on the type of `i2`.

3. The compiler can first virtualize and then even devirtualize the call to `i3.foo`, since type analysis [27, 10] determines that `i3` can only contain objects with concrete type B.

The optimizing compiler can inline devirtualized method calls, virtual calls, and interface invocations. The compiler can inline a devirtualized call directly, since analysis has revealed the exact target. To inline selected potential targets of a virtual call, compilers can perform various forms of guarded inlining. The compiler can decide which targets to speculatively inline at a call site using

static heuristics [14, 9], profile information [24, 21], and/or static examination of the program's class hierarchy [8, 12]. Jalapeño uses both class tests and method tests [13] to perform guarded inlining of virtual calls based on both class hierarchy analysis and on-line profile information.[4]

In addition to determining which calls are legal to inline, the compiler must identify a set of call sites as attractive candidates to inline. Jalapeño's optimizing compiler uses a mix of static heuristics and on-line profile information to make these decisions. The static heuristics identify candidates based on size estimates of the caller and callee, and data-flow properties known at the call site. Furthermore, the static heuristics elect to perform a guarded inline of a virtual or interface call only if analysis of the current class hierarchy of the program reveals that there is only one possible target for the call. Note that the adaptive optimization system generally does not optimize a method until it becomes a hot spot in the program's execution. We expect most dynamic class loading that affects such a call site to happen before Jalapeño optimizes and speculatively inlines it.

These static heuristics will not identify many of the most common interface methods as inline candidates. The most common interfaces (e.g. `Serializable`, `Enumeration`, etc.) have many different implementations. In general, a compiler would have to resort to context-sensitive interprocedural analysis to virtualize or devirtualize call sites for methods of these interfaces. Such analysis usually costs too much for a JIT or run-time compiler, which must normally rely on less expensive, solely intraprocedural analysis. As a result, a JIT will likely fail to statically determine one target for many interface calls.

Jalapeño's adaptive optimization system solves this problem using on-line profile-directed inlining to identify candidates to be inlined with guards at hot call sites. Normally, the method test guards inlined interface methods, just as it guards inlined virtual methods.[5]

Adaptive inlining naturally tends to minimize the overhead of conflict resolution stub execution. If a particular conflict resolution stub executes frequently, the adaptive optimization system will tend to flag at least one target method as "hot". The adaptive system heuristics would then likely inline that method into hot call sites. This optimization will reduce the frequency of conflict resolution stub execution.[6] If the heuristics do not inline a hot target method, deeming it too big to inline, then its execution cost likely dominates overhead imposed by the conflict resolution stub.

As discussed in section 2, both guarded interface invocation and the normal interface dispatch scheme require a dynamic type check. The compiler can reduce the overhead of this type check in two ways. First, if the compiler can use type analysis to staticly verify that the receiver implements the target interface, the runtime check can be eliminated. Secondly, the optimizing compiler represents dynamic type checks as binary operators in the low-level intermediate representation used to drive code motion and redundancy elimination. If the compiler can identify multiple type checks of the same object against a particular interface, it can remove the redundant checks. Partial redundancy elimination can, for example, hoist the loop-invariant type checks from the loop in figure 3. Similarly, the compiler will optimize redundant loads in the dispatch sequence; the TIB base pointer load for

[4] If class hierarchy analysis determines that a non-devirtualized call site can currently only invoke a single target method (but the callee method is not declared to be final), then it could be inlined without a guard by relying on invalidation mechanisms such as on-stack-replacement [23] or code patching [26] to undo the inlining if a future class loading event invalidates it. Neither of these recovery mechanisms have been implemented in Jalapeño. However, Jalapeño does use pre-existence [13] as a partial substitute for a full-fledged invalidation mechanism.

[5] In exceedingly rare cases, the compiler may speculatively inline a method from a class that cannot be proven, at compile time, to implement the target interface (for exmaple, figure 4). In this case, the compiler must insert a dynamic type check to ensure that the receiver implements the interface before executing the inlined body. Although Jalapeño implements this additional inlining guard, we have never actually seen this scenario occur outside of a regression test that was written specifically to provoke it.

[6] As a side effect, the profile-directed inlining algorithm has a slight tendency to further reduce the number of interface dispatches that go through conflict resolution stubs. Profile-directed inlining chooses the "hottest" call edges in the program as candidates for inlining. The adaptive optimization system identifies "hot" call edges via time-based statistical sampling [6]. If a particular interface invocation happens to frequently resort to a conflict resolution stub, the time spent in the conflict resolution stub gets attributed to the sample point in the prologue of the callee. As a result, the overhead of the conflict resolution stub makes the call edge appear "hotter", and thus a more profitable target for inlining.

Type of Invoke	Trivial Callee	Simple Callee
virtual	9.22	22.33
Fast Interface Implementation		
inteface with 1 method (no IMT conflict)	10.22	22.38
inteface with 100 methods (3 element stub)	20.31	34.42
Naïve Interface Implementation		
inteface with 1 method	162.9	176.0
inteface with 100 methods	964.1	974.4

Table 1: Cost, in clock cycles, of round-trip interface dispatch in Jalapeño, under the original naïve implementation and the fast implementation described in this paper.

object `e` in the figure can also be hoisted from the loop.

```
Enumeration e = getEnumeration();
while (e.hasMoreElements()) {
  use(e.next());
}
```

Figure 3: A common interface idiom requiring a single dynamic type check.

The virtual machine specification generates some fringe cases that the compiler must handle correctly. For example, in Figure 4, the compiler might successfully virtualize an interface call, but still fail to eliminate the dynamic type check for the dispatch. Suppose the compiler analyzes `Test.test()`, with only intraprocedural information and inlining. Further suppose the compiler inlines `createI` into `test()`, but doesn't choose to inline `createA()` (perhaps because it is too big). The compiler can virtualize the call to `foo()`, since type propagation determines that `i` is a subclass of `A`. However, it cannot remove the dynamic type check, since `A` doesn't implement `I`.[7]

6 Experimental Results

This section empirically assesses the effectiveness of IMT-based dispatch scheme presented in Section 4 and the optimizing compiler techniques discussed in Section 5. Section 6.1 presents micro-benchmark results that focus on the direct costs of virtual and interface dispatch in Jalapeño. Next, Section 6.2 describes our suite of larger benchmarks and presents data on the dynamic frequency of interface invocation in each program. Finally, Section 6.3 presents the bottom line performance impact of Jalapeño's interface handling techniques. The performance results reported below were obtained on an IBM F50 Model 7025 with three GB of main memory and with two 333MHz PPC604e processors running AIX v4.3.

6.1 Micro-benchmarks

Several micro-benchmarks compare the direct costs of interface and virtual dispatching in Jalapeño. The core of each micro-benchmark consists of a loop that in each iteration performs a method invocation 20 times. The loop executes 1,000,000 times, and the total wall clock time spent executing

[7]Generating the bytecodes for this example would be a non-trivial undertaking. As written, a source-to-bytecode compiler would object to the program of Figure 4. However, if class `A` were altered to implement `I`, all three classes and the interface would compile together. Class `A` could then be reverted to its original text and recompiled on its own. Thus, giving rise to the possibility of an `IncompatibleClassChangeError`.

```
class A {
  public int foo() { ... }
}

class B extends A implements I {}

interface I {
  public int foo();
}

class Test {
  int test() {
    I i = createI();
    return i.foo();
  }
  I createI() { return createA(); }
  A createA() { ... return new B(); }
}
```

Figure 4: An anomalous example; after inlining `createI`, the compiler can virtualize `i.foo()`, but cannot remove the dynamic type check to ensure that the object referred to by `i` actually implements `I`.

the loop is reported. Thus, these results include the cost of the method body, and so provide an upper bound on the cost of interface dispatch. Method inlining was disabled for these experiments.

The micro-benchmarks exercise three categories of invocation: virtual method invocation, interface invocation where the interface has only one method, and interface invocation where the interface has many (100) methods. The final category illuminates the costs of conflict resolution stubs.[8]

The micro-benchmarks call one of two target methods. The first target (*Trival Callee*) simply returns the integer constant 1. For this trival method, Jalapeño's optimizing compiler applies leaf method optimizations which avoid the normal method prologue and epilogue sequences. In fact, the generated code for the callee method contains only two machine instructions. The second example (*Simple Callee*) invokes a slightly more complex target method. This callee method conditionally either returns 1 or invokes another method, based on the value of a static field. The benchmark sets value of this static field at runtime such that the method always returns 1; however, the compiler cannot statically fold the branch and does not apply leaf method optimizations.

Table 1 presents the results of these experiments. In the most typical case (IMT with no conflict dispatching to a non-trivial callee), the difference between a virtual or interface dispatch is insignificant. Recall that the interface dispatch requires one extra register move immediate compared to virtual dispatch, in order to set up the hidden parameter. With the non-trivial (Simple) callee, the superscalar hardware overlaps the extra move with other operations, resulting in zero observed overhead.

With the trivial callee the hardware cannot overlap the extra register move with other work (having no other work to do). In this case, we observe a 10% overhead for IMT-based interface dispatching.

When the interface size forces a three-element conflict resolution stub, the dispatch overhead increases by 54%. The extra cost of the three-element conflict resolution stub is roughly the same as the cost of the prologue/epilogue sequence.

[8]The IMT size used here is 29 slots. It is possible that a slightly larger IMT would actually use marginally less total space, since each conflict resolution stub consumes at least 15 words of memory.

Benchmark	Description	Classes	Methods	Bytecodes
`compress`	Lempel-Ziv compression algorithm	48	489	19,480
`jess`	Java expert shell system	176	1101	35,316
`db`	Memory-resident database exercises	41	510	20,495
`javac`	JDK 1.0.2 Java compiler	176	1496	56,282
`mpegaudio`	Decompression of audio files	85	712	51,308
`mtrt`	Two-thread raytracing algorithm	62	629	24,435
`jack`	Java parser generator	86	743	36,253
`SPECjbb2000`	simulated transaction processing [41]	132	1778	73,608
`opt-compiler`	Jalapeño optimizing compiler	414	5030	139,004
`HyperJ`	Hyper-J [34, 25] composition tool	421	5003	136,957
`DOMCount`	Xerces v1.2.3 [39] XML parser	142	1880	88,134

Table 2: Benchmark characteristics. For each benchmark, the Table gives the number of classes loaded, the number of methods compiled at runtime, and the number of bytecodes compiled at runtime. The statistics include both application code and library code loaded at runtime. The first seven rows comprise the suite of SPECjvm98 benchmarks.

To provide context for the experimental results on the larger applications, Table 1 also reports the results of running the same micro-benchmarks using Jalapeño's prior interface dispatch implementation. The previous implementation used a naïve (non-caching) implementation of the JVM specification: for every interface dispatch, the system scanned the receiver's class object to find the matching virtual method. As these measurements indicate, this naïve implementation adds substantial overhead compared to virtual dispatch, and scales poorly when the receiver class defines a large number of virtual methods.

6.2 Application Characteristics

Table 2 describes the application benchmark suite, comprising the SPECjvm98 [40] benchmarks and several larger codes. Improving interface invocation will only help an application with non-negligible overhead due to interfaces (Amdahl's law [4]). Therefore, let us begin by trying to quantify the importance of interface method invocation in each benchmark. In this experiment, the system does not employ the techniques described in this paper; in particular, the compiler does not virtualize, devirtualize, or inline interface invocations. However, it may inline and devirtualize static, special, and virtual calls.

Using instrumentation capability in Jalapeño's adaptive optimization system, the optimizing compiler inserted counters into the generated code to count and categorize the dynamic non-inlined invocations during benchmark execution.[9] All macro-benchmark run in a testing harness that executes the benchmark ten times, printing and clearing the counters at the start of each run.

Figure 5 shows the rate of non-inlined invocations per second on the tenth run for each of the four different `invoke` bytecodes. [10] Based on this data, we expect can little benefit on `mpegaudio` and absolutely no benefit on `compress` and `mtrt`. In fact, both `compress` and `mtrt` make exactly one interface method invocation per iteration. Results for these two benchmarks will not be reported hereafter. The potential for improvement on `db`, `opt-compiler`, `HyperJ` and `DOMCount` appears significant; calls to interface methods represent a substantial portion of their non-inlined invocations.

[9] Only invocations in methods that are executed frequently enough to be selected for optimizing recompilation will be counted. However, the calling behavior of infrequently executed methods should not substantially impact performance.

[10] The bar for `compress` appears invisible because after inlining, `compress` makes only 163 method calls per second.

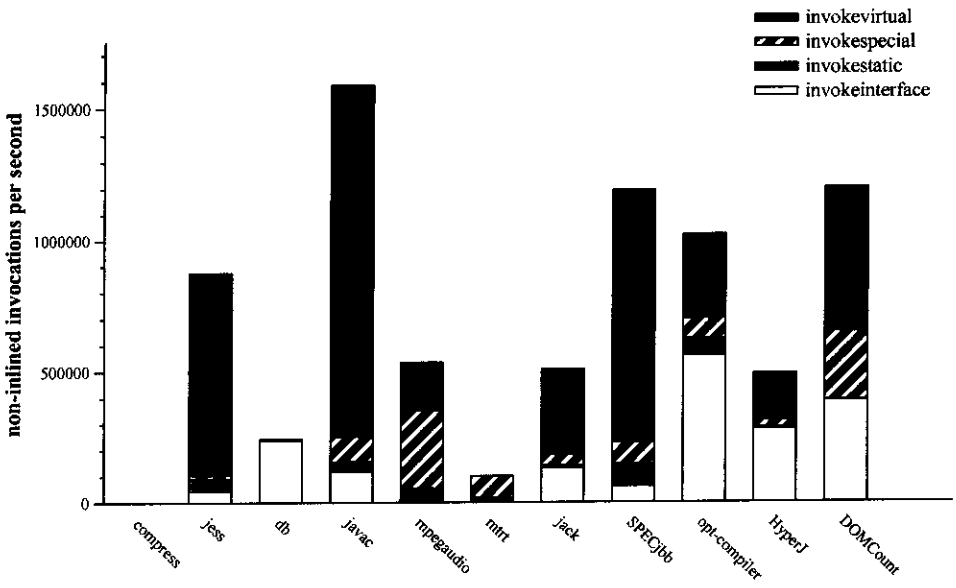

Figure 5: Dynamic rate of non-inlined invocations categorized by invoke bytecode. Optimizing interface invocation will only improve the performance of those applications with a significant rate of interface invocation (the bottom white portion of each bar).

6.3 Application Performance

This subsection reports the bottom-line performance impact of improving interface invocation on the application benchmarks using three Jalapeñovariants:

- *old*: This configuration uses the prior naïve implementation of interface invocation and does not apply any of the compile-time optimizations described in Section 5.
- *new*: This configuration uses the IMT-based implementation of interface invocation, but does not include the compile-time optimizations.
- *new+opt*: This configuration augments *new* by enabling the compile-time optimizations of interface invocations.

Comparing *old* to *new* demonstrates the importance of an efficient mechanism for dispatching interface calls. The difference between *new* and *new+opt* isolates the improvements made by virtualizing, devirtualizing, optimizing type checks, and/or inlining interface invocations.

Figure 6 depicts the performance improvement of *new* and *new+opt* over *old*. In general, the rate of interface invocation of Figure 5 correlates closely with the performance improvement due to IMT-based dispatching. Benchmarks with a significant rate of interface calls (100,000 invocations per second or greater) enjoyed performance improvements ranging from 5% on `javac` to 561% on HyperJ.

`HyperJ` shows tremendous improvement, disproportionate to its relative rate of interface invocation when compared to `opt-compiler` and `DOMCount`. This discrepancy results from the exceptionally poor performance of Jalapeño's old interface method dispatch to an object of a class that defines a large number of virtual methods. In `HyperJ`, some of the most frequently used interfaces define at least dozens of methods, with one case over one hundred. Although this also tends to cause collisions in the IMT, this cost pales when compared to the linear search through the receiver class's defined virtual methods done in the *old* configuration.

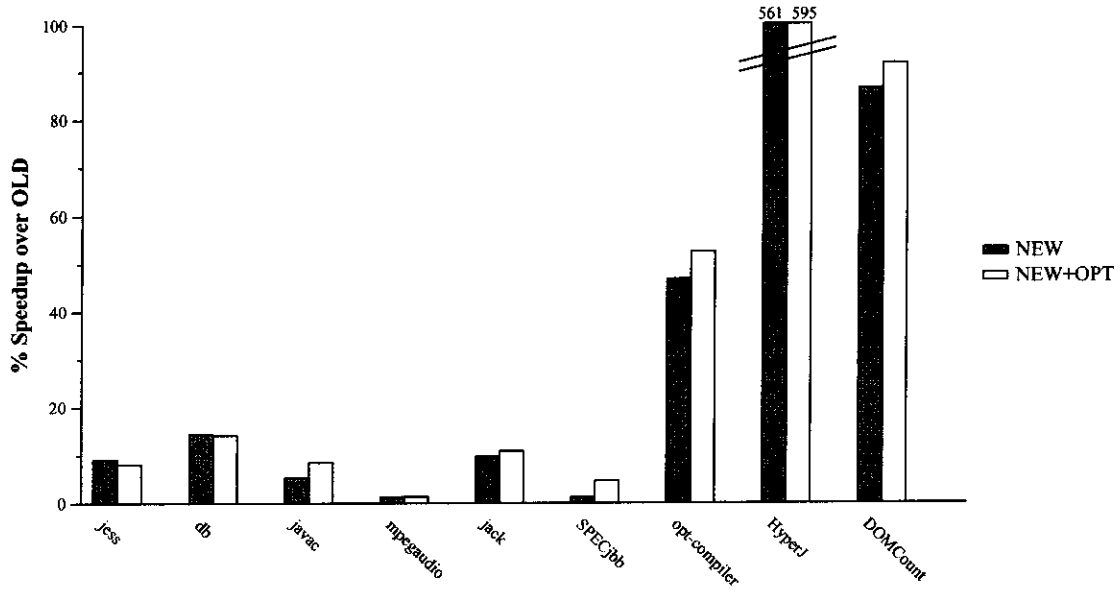

Figure 6: Percent performance improvement of *new* and *new+opt* over *old*. Note that the graph truncates the bars for `HyperJ`; the actual improvements were 561% and 595%.

In Figure 6, the additional gains from *new+opt* over *new* were less dramatic, ranging from a 1% degradation on `jess` and `db` to a 5% improvement on `HyperJ`. The larger programs saw gains consistently between 3% and 5%.[11]

Figure 7 provides more detail on the *new+opt* configuration, reporting the dynamic percentage of interface invocations handled by each dispatching mechanism. The system dispatches each interface method call by one of the following mechanisms:

- *Virtualized and inlined*: Based on the results of type analysis, the optimizing compiler virtualized the interface call. The compiler then inlined the virtual call. With the exception of a tiny fraction of the virtualized and inlined calls in `HyperJ`, the compiler consistently further devirtualized, and could omit the method test to guard the inlined method body.

- *Virtualized*: The optimizing compiler succeeded in virtualizing the call through type analysis, but was either unable or unwilling to inline it. The compiler might not inline a virtual call for any of number of reasons. Prime candidates include: a) the adaptive system compiled the caller method at its lowest optimization level, with all inlining disabled, b) the inlining heuristics deem the callee method too big to inline into the calling context given the call site's dynamic frequency, and c) the call graph profile identifies multiple possible targets at a dynamically polymorphic call site, but does not identify any of the receivers as a dominant target profitable to inline.

- *Static guarded inline*: The optimizing compiler failed to virtualize the call with type analysis. But, at the time the method was optimized, the set of loaded classes defined only one implementation of the interface method. The compiler, based exclusively on size heuristics (without profile information), speculatively inlined the single static target, guarded by a run-time check.

[11] It is worth noting that the performance of *new* is already fairly good for a research project; competitive with that of a state-of-the-art product JVM and JIT. For this benchmark suite, on average *new* is 8% faster than the IBM Developers Kit for AIX, Java Technology Edition, Version 1.3 with JIT version 3.6; relative performance on individual programs varies from -25% to +39%.

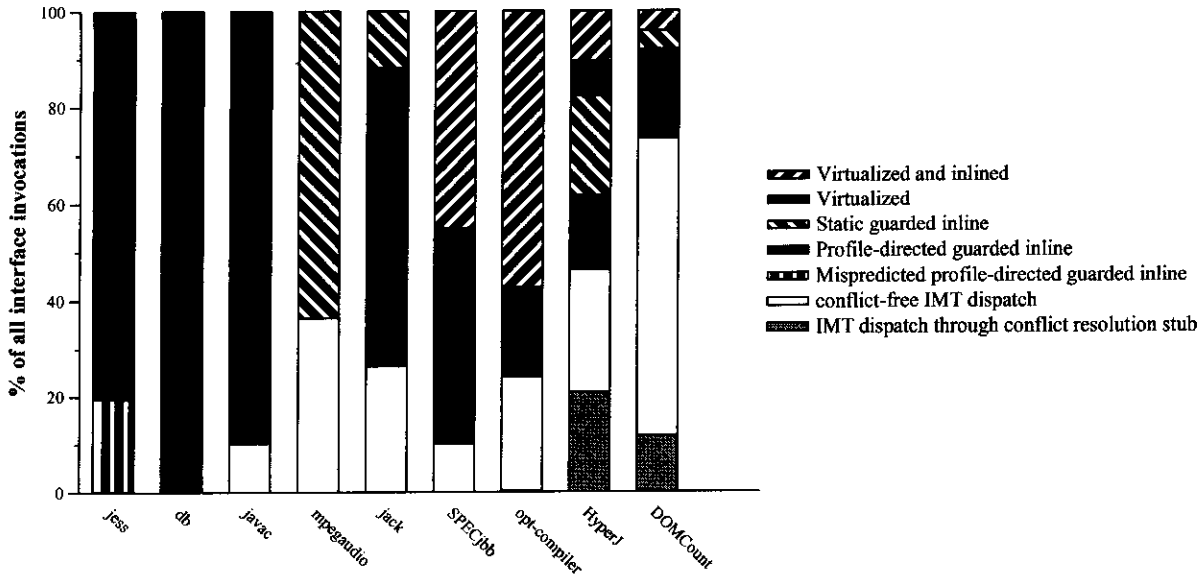

Figure 7: Dynamic percentage of interface invocations handled by each dispatch mechanism.

In principle, this run-time check could fail, if a class loaded in the future defines a new receiver method for the call site. However, in our experiments, this case never occurred. The adaptive system delays compilation and optimization of each method until profile information indicates the method is hot. In practice, by the time the adaptive system flags a method as hot, all relevant classes have been loaded.

- *Profile-directed guarded inline*: Class hierarchy analysis determines that the interface method has multiple possible implementations. However, the online profile information identifies one or more dominant targets for the call site. Based on this information, the compiler speculatively inlined the dominant target(s) with a run-time guard.

- *Mispredicted profile-directed guarded inline*: The run-time guard at a profile-directed inline site failed, and the code fell back to an IMT dispatch. This occurs either due to inaccurate profile information, or at call sites that have one or more dominant targets but occasionally invoke other targets. In other words, the call site is dynamically polymorphic, but the compiler does not inline all receivers.

- *conflict-free IMT dispatch*: No other technique applied, and the compiler resorted to basic IMT-based dispatch. There was no conflict, so the dispatch through the IMT went directly to the target method.

- *conflict-free IMT dispatch*: No other technique applied, and the compiler resorted to basic IMT-based dispatch. Multiple methods were assigned to the IMT slot in question, so the dispatch went through a conflict resolution stub before finally invoking the right target method.

The mix of dispatching mechanisms varies widely from program to program. For example, the fraction of invocations that actually dispatched through the IMT ranged from 0% to 73%. In several cases, type analysis effectively virtualized interface invocations, covering 89% on `SPECjbb`, 76% on `opt-compiler` and 18% on `HyperJ`. We expected this on the `opt-compiler`, since we initially implemented the optimization to handle the most common patterns of interface usage in our own

compiler. However, we were pleasantly surprised with the virtualization success on `HyperJ` and `SPECjbb`. Note that in these results, the optimizing compiler relies almost exclusively on intra-procedural analysis. We expect even better virtualization and devirtualization results using more aggressive interprocedural type analysis.

The static guarded inlining heuristic succeeded for only three of the benchmarks; it had the most impact on `HyperJ`, where it applied to 21% of the interface invocations. It applied to 64% of the call sites in `mpegaudio`; however, interface dispatching is not a major cost in this benchmark. In practice, we expect the static heuristic to apply to programs that use only a single implementation of an interface from a general component architecture or library. The static heuristic does not work for the more heavily used interfaces in the Java standard library, such as `java.util.Enumeration`; the compiler must rely on profile-directed inlining or type analysis to handle these cases.

Regarding IMT conflicts: on `HyperJ`, `DOMCount` and `opt-compiler`, 45%, 16%, and 1.5%, respectively, of dispatches through the IMT went through a conflict resolution stub. The other benchmarks never dispatched through conflict resolution stubs.

Of all the benchmarks, `jess` stands out for its high (19%) rate of mispredicted profile-directed inlined calls. We investigated this phenomenon a bit further, and discovered that `jess` contains one frequently executed interface invocation call site. This site calls an interface method with 96 implementations. Of these 96, the call site invokes 14 distinct receivers during the size 100 benchmark run. The top four most frequent targets account for 52%, 18%, 13%, and 8% of the dynamic invocations, respectively. Each of the other targets accounts for less than 3% of the dynamic invocations. At the most frequently called instance of the call site, [12] the compiler heuristics inline only the two most frequently invoked targets. Thus, any calls to the other 12 targets at that call site instance are mispredicted.

7 Related and Future Work

The first subsection describes *itables* (interface tables), probably the most commonly used mechanism for interface method dispatch in high performance Java implementations. Issues for Java interface method dispatch closely resemble those faced for efficient virtual method dispatch in dynamically-typed object oriented languages. Approaches for solving this problem rely on either caching or selector (signature) indexed dispatch tables. The next two subsections review previous work in each category and discusses ways in which the Jalapeño techniques might be improved.

7.1 Interface tables

An itable is a virtual method table for a class restricted to the subset of the class's methods that match those of an interface that the class implements. One could view an itable as the moral equivalent of the vtable for a C++ sub-object that corresponds to the interface [38]. To dispatch an interface method to an instance of a class, the system must locate the itable that corresponds to the appropriate class/interface pair. Typically, the system stores itables in an array reachable from the class object. Sometimes a JIT compiler can determine statically what itable applies at a particular interface method invocation site. If not, it must search for the relevant itable at dispatch-time [36, 18, 37]. In a straightforward implementation, search time increases with number of interfaces implemented by the class. Thus, basic itable schemes tend to degrade when classes implement more than a handful of interfaces.

The CACAO JVM [29] implements a variant of the basic itable scheme that avoids a dispatch-time search for the right itable. Rather than storing a class's itables in a dense array, it maintains a sparse array of itables for each class indexed by interface id. This sparse array grows down from (the CACAO analog of) the TIB, thus making it easily accessible for dispatching. To dispatch an

[12] There are multiple instances in the generated code due to inlining and dynamic recompilation.

interface method, CACAO simply loads the TIB from the object, loads the itable for the interface at a constant offset in the TIB, and obtains a pointer to the callee code from a constant offset into the itable. With this mechanism, an interface method dispatch introduces one more dependant load than a virtual method dispatch.

To somewhat reduce the space overhead of arrays of itables, CACAO can safely truncate the interface table for a class to end with its last non-empty entry, since empty entries will never be accessed. This optimization eliminates space overhead for classes that don't implement any interfaces. Nevertheless, in non-trivial programs, the interface tables for classes that implement any interface will be large and sparse, since most classes implement only a tiny fraction of the total set of interfaces.

7.2 Caching and Collision Abatement

Early Smalltalk-80 systems used dynamic caching [30] to avoid performing a full method lookup on every message send. The runtime system began method lookup by first consulting a global hash table (keyed by a class/selector pair) that cached the results of recent method lookups. Although consulting the hash table was significantly cheaper than a full method lookup, it was still relatively expensive.

Therefore, later Smalltalk systems added inline caches [14] as a mechanism to mostly avoid consulting the global cache. In an inline cache, the call to the method lookup routine is overwritten with a direct call to the method most recently called from the call site. The prologue of the callee method is modified to check that the receiver's type matches and calls the method lookup routine when the check fails. Inline caches are extremely effective if the call site is monomorphic, or at least exhibits good temporal locality, but perform poorly at most polymorphic call sites.

Polymorphic inline caches [22] were developed to overcome this weakness. In a polymorphic inline cache, the call site invokes a dynamically generated PIC stub that executes a sequence of tests to see if the receiver object matches previously seen cases. If a match is found, then the correct target method is invoked; if a match is not found, the PIC terminates with a call to the method lookup routine (which may in turn choose to generate a new PIC stub for the call site, extended to handle the new receiver object).

Similar ideas can be applied to interface method dispatch. When an interface method is dispatched, the system can cache some *history* information regarding the dynamic call [13]. For interface method dispatch, the history consists of a *key* and a VMT *offset*. The caching algorithm employed dictates the nature of the key. The VMT offset represents the offset of the dispatched method. The next time the system encounters a *similar* invocation, it can re-use the old offset if the new key matches the old one.

Any of dynamic caching, inline caches, or polymorphic inline caches could be used to dispatch interface methods. In fact, the first edition of The Java Virtual Machine Specification [31] defined a "quick bytecode" that acted as inline cache by caching history with the invocation site[14]. Other caching schemes could be used as well. For example, if invocations on the same *object*, or objects of the same *class*, are considered similar, the key represents the signature of the interface method and the information is cached either in the object or its class object. Or, if invocations of the same *interface method signature* are considered similar, the key will be the class of the object on which the method is invoked and the cache could be stored in a parallel structure to the table of interface-method signatures.

A feature of **any** caching scheme is that it relies on temporal locality and thus cannot guarantee efficient dispatching for all programs. Polymorphic inline caches are less vulnerable than simple inline

[13]The system must take care when caching on SMP computers. Unless the key-value pair is updated atomically, a processor might see the first value of one pair and the second value of another. In most circumstances, this spells disaster! Since the cost of explicit synchronization is often prohibitive, it may be beneficial to encode these pairs in a single word to exploit atomic single-word memory access.

[14]The quick bytecodes have been dropped from the second edition of the JVM specification [32].

caches, but they still can perform poorly at "megamorphic" call sites. This paper's experimental results indicate that cache mispredictions would be an issue even for a polymorphic inline cache on `jess` and possibly to a lesser extent on `HyperJ`.

Ideas previously employed to increase the efficiency of polymorphic inline caches at megamorphic call sites could also be used by Jalapeño to ameliorate the effects of a large number of collisions in an IMT slot. Currently, Jalapeño's conflict resolution stubs employ linear search; in larger stubs binary search could be used instead. A move-to-front algorithm [15] could also be employed, although the complexity of doing this safely on an SMP may overwhelm any performance advantage such a scheme would have over binary search. Stubs could also be periodically re-generated taking into account the dynamic profiling data being gathered by the adaptive system to order the most frequently taken cases first.

The adaptive system could also use a variant of caching to minimize array-bound-checking costs associated with dynamic type checking for interfaces. The Implements Trits Vector (ITV) can be thought of as being partitioned into *fast*, not requiring a bounds check, and *slow*, requiring a bounds check, parts. Initially, the fast part would be largely empty. (Heavily used interfaces, like `Enumeration`, could be preallocated in the fast part.) During execution, as an interface is discovered to be used heavily, the adaptive system could allocate a second entry for it in the fast part of the ITV.

7.3 Selector Indexed Tables

Selector indexed dispatch tables [11] provide a straightforward but space-intensive solution to the interface dispatch problem. Each class maintains a (potentially large) table indexed by interface signature id. Entries that correspond to an interface signature that the class actually implements point to the code for the matching virtual method; all other entries are null. Selector indexed dispatch tables were originally proposed to implement virtual method dispatch in dynamically typed object oriented languages, but were considered too space intensive to be practical.

Several approaches have been proposed to greatly reduce the space costs of selector indexed tables. Driesen considered using a specialized sparse array data structure [17]. The Sable VM also uses selector indexed dispatch tables for interface method dispatch, but reduces the space impact by releasing "gaps" in the dispatch tables to the allocator to reallocate as small objects [19]. Although clever, this trick can significantly complicate both allocation and garbage collection. [15]

Selector coloring [16] has been applied to reduce the size of selector indexed dispatch tables. Just as in register allocation [7], the assignment of ids to selectors can be viewed as a graph coloring problem. Two selectors can be assigned the same color if they are never implemented by the same class. Using this approach, several algorithms have been proposed that greatly reduce the size of the dispatch tables [16, 5, 44, 43]. Unfortunately, all of these algorithms assume that the set of selectors and the classes that understand them are known *a priori*. Thus, previous selector coloring algorithms are a poor match for Java, since the JVM cannot know this information in advance.

CACAO's second scheme for interface method dispatch is selector coloring [29]. This second scheme improves over their extended itable scheme described above by eliminating the need for an extra indirection (and thus virtual and interface invocations cost exactly the same), but the space implications still could be severe in some programs. Although they do not state so explictly in [29], their selector coloring scheme is only applicable if all interfaces and all classes that implement interfaces are known to the JVM in advance.[16]

Statically-typed object-oriented languages have faced similar problems with multiple inheritance. The usual solution in C++ [38] uses multiple dispatch tables for each type, one corresponding to

[15] It may also be worth noting that the Sable VM puts a limit (1000) on the number of interface signatures that are dispatched in this fashion. After the limit is exceeded it falls back to a slower dispatch mechanism.

[16] Alternatively an optimistic coloring could be used. But, if an interface were loaded that violated that coloring, a new coloring would have to be computed and existing interface invocation sites would have to be patched. Such a scheme has been considered for CACAO, but not implemented [28].

each superclass. An object pointer indicates the dispatch table corresponding to the *static* type of the object reference. Virtual dispatch may require *pointer bumping* or *this-pointer adjustment* to force the object pointer to refer to the appropriate offset in the object header.

The C++ solution uses significantly more space in the object than necessary for Java, which does not have multiple implementation inheritance. Myers [33] presented a sophisticated algorithm to reduce the space overhead by merging dispatch tables for compatible types and exploiting bidirectional layout in the object header.

Both of these techniques rely on complete knowledge of an object's superclasses at compile-time. Unfortunately, this knowledge is not necessarily available in Java, since the class loader may not load the full interface hierarchy before compiling a class's methods. Furthermore, even with bidirectional layout, these mechanisms may increase the object header size and add runtime overhead for pointer bumping.

7.4 Summary

The approach taken by Jalapeño is best described as selector coloring, extended to efficiently handle coloring "mistakes" via custom-generated conflict resolution stubs. Conflict resolution stubs are similar in function to the PIC stubs used in polymorphic inline caching, although conflict stubs are global (not call site specific) and complete (do not fall back to a secondary dispatching scheme). By including a mechanism to handle color collisions, Jalapeño is able to obtain most of the benefits of selector coloring without having to know all interface signatures in advance, and without committing to making all IMTs large enough to be able to obtain a perfect coloring for every program.

However, Jalapeño should improve on its current (trivial) algorithm for selector coloring by combing some large collection of standard Java classes, looking for interface method signatures and for interfaces which are simultaneously implemented. Based on this information and profile data to indicate which interface signatures are likely to be dynamically important, the signatures could either be perfectly hashed to prevent all collisions or a standard register allocation algorithm could be employed to minimize the expected dynamic number of collisions in a more constrained IMT. During JVM execution new (unexpected) method signatures could either be assigned to reserved empty slots or to infrequently-used slots.

8 Conclusions

An early conference paper on the Jalapeño runtime [2] admitted that "[w]e don't make much use of `interfaces` because the performance overhead was too high to use it to call frequently executed methods."[17] This was particularly damning since part of the stated rationale for writing Jalapeño in Java was the hope that doing so would "give us more experience with the language, help us identify some of its problematic features, and give some insight into how to implement them efficiently." An anonymous reviewer observed:

> "The comment about not using interfaces is sad. And invites the question: if they had been used, would the performance of interface invocations now be better?"

That remark spawned the work reported here.

Interface methods introduce three potential sources of run-time overhead: the cost of implicit dynamic type checking entailed by interface calls, the overhead of dispatching a virtual method through an interface call, and the opportunity cost of not being able to inline interface calls as effectively as virtual calls. None of these costs is prohibitive.

[17] The Jalapeño optimizing compiler group early on adopted a stylized form of interface usage that could be easily special-cased and optimized away, violating the letter but not the spirit of this admission.

The implicit dynamic type check costs two loads, a comparison, an untaken branch and an array bounds check (a load, a comparison, and another untaken branch). In most cases, the bounds check can be eliminated. Furthermore, the compiler can often amortize a single type-check over many interface invocations, if not eliminate it completely.

If there is no conflict in the IMT, an interface call introduces one register move more than its virtual counterpart. If there is an IMT conflict, the cost is a few instructions. Furthermore, the size of the IMT, and the assignment of anticipated interface method signatures to IMT slots, can be chosen so as to keep the probability of a conflict small.

Inability to inline interface method calls effectively would have been a show stopper, since overhead for making any method call (two transfers of control, method prologue, and epilogue) dwarfs the difference between a cost of an interface method call (including any dynamic type check) and the cost of a virtual method call. Happily, it is almost always possible (using static analysis and/or online profiling information) to inline the dominant target(s) of any frequently executed interface method call. Jalapeño's adaptive compiler uses the same criteria to inline virtual and interface methods and uses the same guard to protect the inlined code (except for the implicit dynamic type check which can almost always be eliminated in this case).

In conclusion, a programmer, or program generating system, should feel free to fully exploit Java interfaces without fear of inherent performance degradation.

Acknowledgments

This work would not have been possible without the efforts of the entire Jalapeño team. Thanks especially to David Bacon and Peter Sweeney for invaluable feedback, and to Harold Ossher for contributing the `HyperJ` benchmark. We are also indebted to the anonymous reviewer of an earlier paper.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[2] B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, D. Lieber, S. Smith, and T. Ngo. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 314–324, 1999.

[3] B. Alpern, A. Cocchi, and D. Grove. Dynamic type checking in Jalapeño. In *USENIX Java Virtual Machine Research and Technology Symposium*, Apr. 2001.

[4] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS conference*, volume 30, pages 483–485, 1967.

[5] P. André and J.-C. Royer. Optimizing method search with lookup caches and incremental coloring. In *Proceedings OOPSLA '92*, pages 110–126, Oct. 1992. Published as ACM SIGPLAN Notices, volume 27, number 10.

[6] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2000.

[7] G. J. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages 6*, pages 47–57, 1981.

[8] C. Chambers, J. Dean, and D. Grove. Whole-program optimization of object-oriented languages. Technical Report UW-CSE-96-06-02, Department of Computer Science and Engineering. University of Washington, June 1996.

[9] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 146–160, July 1989. *SIGPLAN Notices, 24(7).*

[10] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: Optimizating dynamically-typed object-oriented programs. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 150–164, 1990.

[11] B. J. Cox. *Object Oriented Programming: An Evolutionary Approach.* Addison-Wesley, 1987.

[12] J. Dean. *Whole Program Optimization of Object-Oriented Languages.* PhD thesis, University of Washington, Nov. 1996. TR-96-11-05.

[13] D. Detlefs and O. Agesen. Inlining of virtual methods. In *13th European Conference on Object-Oriented Programming*, 1999.

[14] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *11th Annual ACM Symposium on the Principles of Programming Languages*, pages 297–302, Jan. 1984.

[15] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. *Proceedings of the nineteenth annual ACM Symposium on Theory of Computing, New York City, May 25–27, 1987*, pages 365–372, May 1987.

[16] R. Dixon, T. McKee, M. Vaughan, and P. Schweizer. A fast method dispatcher for compiled languages with multiple inheritance. In *Proceedings OOPSLA '89*, pages 211–214, Oct. 1989. Published as ACM SIGPLAN Notices, volume 24, number 10.

[17] K. Driesen. Selector table indexing & sparse arrays. In *Proceedings OOPSLA '93*, pages 259–270, Oct. 1993. Published as ACM SIGPLAN Notices, volume 28, number 10.

[18] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An optimizing compiler for Java. Technical Report MSR-TR-99-33, Microsoft Research, June 1999.

[19] E. Gagnon and L. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. Technical Report Sable Technical Report No. 2000-3, School of Computer Science, McGill University, Nov. 2000.

[20] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1996.

[21] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 108–123, Oct. 1995.

[22] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In P. America, editor, *Proceedings ECOOP '91*, LNCS 512, pages 21–38, Geneva, Switzerland, July 15-19 1991. Springer-Verlag.

[23] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 32–43, June 1992.

[24] U. Holzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 326–336, June 1994. *SIGPLAN Notices, 29(6).*

[25] IBM Research, 2001. http://www.research.ibm.com/hyperspace/.

[26] K. Ishizaki, M. Kawahito, T. Yasue, and H. K. andToshio Nakatani. A study of devirtualization techniques for a Java just-in-time compiler. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2000.

[27] R. Johnson. TS: An optimizing compiler for Smalltalk. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 18–26, 1988.

[28] A. Krall. Personal Communication, Sept. 1999.

[29] A. Krall and R. Grafl. CACAO – a 64 bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997.

[30] G. Krasner. *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley, 1983.

[31] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, 1996.

[32] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification Second Edition.* The Java Series. Addison-Wesley, 1999.

[33] A. C. Myers. Bidirectional object layout for separate compilation. *ACM SIGPLAN Notices*, 30(10):124–139, Oct. 1995.

[34] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer, 2001. *to appear.*

[35] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, Sept. 1999.

[36] G. Ramalingam and H. Srinivasan. Object model for Java. Technical Report 20642, IBM Research Division, Dec. 1996.

[37] J. Shepherd. Personal Communication, Mar. 2001.

[38] B. Stroustrup. Multiple inheritance for C++. In *Proceedings of the Spring 1987 European Unix Users Group Conference*, Helsinki, 1987.

[39] The Apache XML Project, 2001. http://xml.apache.org/xerces-j.

[40] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98, 1998.

[41] The Standard Performance Evaluation Corporation. SPEC JBB 2000. http://www.spec.org/osg/jbb2000, 2000.

[42] R. Vallee-Rai. Profiling the Kaffe JIT compiler. Technical Report 1998-02, McGill University, Feb. 1998.

[43] J. Vitek and N. Horspool. Compact dispatch tables for dynamically typed object oriented languages. In *Proceedings of International Conference on Compiler Construction (CC'96)*, pages 281–293, Apr. 1996. Published as LNCS vol 1060.

[44] J. Vitek and R. N. Horspool. Taming message passing: Efficient method look-up for dynamically typed languages. In M. Tokoro and R. Pareschi, editors, *Proceedings ECOOP '94*, LNCS 821, pages 432–449, Bologna, Italy, July 1994. Springer-Verlag.