Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless

Bowen Alpern Anthony Cocchi Stephen Fink David Grove
Derek Lieber

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
{alpern@watson,tonyl@us,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, Java interfaces need *not* be a source of significant performance degradation.

We present an efficient interface method dispatch mechanism, associating a fixed-sized *interface method table* (IMT) with each class that implements an interface. 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 a number of interface dispatch mechanisms on a suite of larger applications demonstrates that, even for applications that make only moderate use of interface methods, the choice of interface dispatching mechanism can significantly impact overall performance. Fortunately, several mechanisms provide good performance at a modest space cost.

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. For example, Vallee-Rai reported that the Kaffe JIT Compiler `invokeinterface` bytecode costs approximately 50 times an `invokevirtual` [43]. The initial implementation of interface invocation in the Jalapeño JVM performed similarly poorly (see Section 7).

There are three sources of potential inefficiency with interface methods: dynamic type checks, method dispatch, and inhibition of compiler optimizations. This paper describes techniques to overcome all three of these obstacles.

Section 2 considers the semantics of Java interfaces, elucidating the dynamic type checking 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.

Section 4 describes prior schemes for interface method dispatch and more generally for dispatching methods in the presence of multiple inheritance or dynamic typing. Then, Section 5 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 (roughly the same as a method prologue and epilogue).

Interface calls might hamper inlining or force extra runtime tests to guard inlined method bodies. Section 6 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 runtime 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 7 presents experimental results evaluating the performance impact of different interface dispatching mechanisms. The experiments consider four alternatives: a naïve implementation of the language specification, two variations on *itables*, and IMT-based dispatch. Micro-benchmark results verify that both IMTs and one of the two itable schemes are not significantly more expensive than virtual method dispatch. Results with a suite of larger applications examine the space and time tradeoffs of the four alternatives. Overall, IMT-based dispatch and one of the itable schemes achieve the best runtime performance, and have reasonably low space costs. The experiments also illustrate that even for applications that make only moderate use of interface methods, the choice of interface dispatching mechanism can significantly impact bottom line performance.

Section 8 describes several ways in which the implementation of IMT-based dispatch in Jalapeño could be improved. Section 9 concludes that performance concerns should not deter programmers from using Java interfaces.

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.[1] Additionally, an interface can extend other interfaces.

Most JVM implementations provide virtual method dispatch through a table a associated with each class. This *virtual method table* (VMT) holds a reference to the implementation of each method declared by the class. When the JVM loads a class `A`, it assigns each virtual method in `A` a unique offset in `A`'s VMT. Methods inherited from a superclass retain the unique offset assigned by the superclass. So, each new class that extends `A` inherits `A`'s VMT and offsets for inherited methods. If a class overrides an inherited method, it simply overwrites the VMT entry of the inherited method's offset. The net result is that for a method `foo()` of class `A`, a reference to a suitable `foo()` implementation resides at the same offset in the VMT of class `A` as in the VMT for any subclass of `A`. Furthermore, the VM can assign VMT offsets densely, to minimize the size of each VMT.

[1] The one class with no superclass, `java.lang.Object`, is not *proper* in this sense.

Interfaces and multiple inheritance preclude this dispatch mechanism. Suppose `foo()` is an abstract method of an interface `I`. If two otherwise unrelated classes `A` and `B` both implement `I`, then `A` and `B` must each provide a suitable `foo()` method. However, there is no guarantee that `A.foo()` and `B.foo()` will have been mapped to the same VMT offset, since `A` and `B` share no inheritance relationship.

The interface method dispatch bytecode, `invokeinterface`, also carries a greater runtime verification burden than does its virtual counterpart, `invokevirtual`. The first time the latter bytecode executes, 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 "`this`" object for the virtual method will have a suitable method at the appropriate slot in its VMT. Conversely, 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`.[2]

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 mechanisms employed for statically typed languages with more complex object models (notably C++ [39]) are not acceptable solutions for Java interface methods.

3. THE JALAPEÑO JVM

Jalapeño [1] is a research Java virtual machine targeting server applications. It is written in Java [2]. This design decision allows Java optimization techniques, including those described here, to apply to both application code and to the JVM's compilers, adaptive optimization system, thread scheduler, garbage collector, and other subsystems.

Jalapeño employs a compile-only strategy; it compiles each method to native code before the method executes. Two different compilers address distinct design points. The *baseline* compiler produces poor quality code quickly. The *optimizing* compiler provides several levels of optimization for methods deemed to require better performance. All optimization levels include linear scan register allocation [35] and BURS-based instruction selection [36]. Optimization level 0, the lowest, consists mainly of a set of on-the-fly optimizations performed during intermediate representation (IR) gener-

[2] 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 changes after an initial compilation, subsequent compilation of the file could create the offending class files (hence the name of the exception). Something similar could happen with `invokevirtual`, but, in that case, the incompatibility would be detected at class-loading. Since the interfaces a class implements are *not* necessarily 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.

ation. Level 1 augments level 0 with aggressive inlining (driven by both static heuristics and online profile information) and a number of other local and intra-procedural flow-insensitive optimizations. Level 2 augments level 1 with a suite of intra-procedural static single assignment (SSA) based optimizations.

Jalapeño's adaptive optimization system [5] maintains statistical samples of the dynamic call graph. Using this information it schedules frequently called and/or computationally intensive methods for recompilation at an appropriate level of optimization. The adaptive system also relies on the on-line profile data to guide inlining decisions.

Jalapeño supports a variety of configurations; this paper assumes the following configuration. The JVM runs on a PowerPC-based SMP running the AIX operating system. It uses a parallel, non-generational 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 the method executes for the first time. The adaptive optimization recompiles hot methods 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 is a Java array of objects [3]. A TIB consists of a fixed-size header section, and a variable-size VMT. The first slot of a TIB header contains a reference to an object that describes the type. Three more slots are used to provide an efficient implementation of 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 a 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 grow. 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.

[3] Some of the performance implications of requiring the TIB to be a legal Java array are considered in Section 7.4

4. PRIOR TECHNIQUES FOR INTERFACE DISPATCH

The first subsection describes *interface tables* (itables), probably the most commonly used mechanism for interface method dispatch in high performance Java implementations. The problem of dispatching Java interface methods is closely related to that of virtual method dispatch in other object-oriented languages with dynamic typing. The following two subsections review previous work in caching and method signature (selector) indexed dispatch tables and describe how these techniques have been adapted for Java. Finally, the last subsection discusses mechanisms to implement virtual method dispatch in the presence of multiple inheritance in C++ that are less amenable to adaptation to Java.

4.1 Interface tables

An *itable* is a virtual method table for a class, restricted to those methods that match a particular interface the class implementation. To dispatch an interface method, the system must first locate the itable that corresponds to the appropriate class/interface pair. The JVM can then load the desired target method from a known offset in this itable. 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 [37, 17]. In a straightforward implementation, search time increases with number of interfaces implemented by the class. However, most systems augment this basic searched itable approach with some form of itable cache or move-to-front algorithm [14] to exploit temporal locality in itable usage to reduce expected search times.

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 list that must be searched, it maintains an array of itables for each class indexed by interface id. This (mostly empty) array grows down from (the CACAO analog of) the TIB, thus making it easily accessible for dispatching. To dispatch an 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 only one more dependant load than a virtual method dispatch.

To somewhat reduce the space overhead of arrays of directly indexed 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 mostly empty, since most classes implement only a tiny fraction of the total set of interfaces.

4.2 Caching

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[4] 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 [13] 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 (PICs) [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.[5] 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.[6] 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.

[4]the selector, or signature, of a method is its name, the types of its parameters, and its return type (if any).

[5]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.

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

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 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 some programs (*jess* and possibly *HyperJ*).

4.3 Selector Indexed Tables

For our purposes, the *signature* of a Java method is its name together with the types of its arguments, if any, and its return type (possibly `void`). Signatures of interface methods are assigned unique small integer identifiers called *selectors*. Selector indexed dispatch tables [10] provide a straightforward but space-intensive solution to the interface method dispatch problem. Each class maintains a (potentially large) table indexed by selector. Entries corresponding to a method signature of an interface 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 [16]. 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 [18, 19]. Although clever, this trick can significantly complicate both allocation and garbage collection.[7]

Selector coloring [15] has been applied to reduce the size of selector indexed dispatch tables. Just as in register allocation [6], the assignment of identifiers 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 [15, 4, 45, 44]. Unfortunately, all of these algorithms assume that the set of selectors and the classes that implement 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 directly indexed 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. However, this scheme is only applicable if all interfaces and all classes that implement interfaces are known to the JVM in advance. An optimistic coloring scheme with a recovery mechanism to recolor the selectors and patch previously

[7]However, 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.

compiled interface invocation sites has been considered for CACAO, but has not been implemented [28].

4.4 Multiple Inheritance in C++

Statically-typed object-oriented languages have faced similar problems with multiple inheritance. The usual solution in C++ [39] uses multiple dispatch tables for each type, one corresponding to each superclass. An object pointer indicates the dispatch table corresponding to the *static* type of the object reference. Virtual dispatch may require *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 this-pointer adjustment.

5. IMT-BASED INTERFACE DISPATCH

Jalapeño implements a new interface dispatch mechanism, called the interface method table (IMT). IMT-based dispatch aims to match the efficiency of selector indexed dispatch tables while avoiding both the potentially large space costs and the *a priori* knowledge requirements of selector coloring. The key idea is to include a mechanism to handle color collisions. By being able to tolerate color collisions, IMT-based interface method dispatch obtains most of the benefits of selector coloring without having to know all interface method signatures in advance, and without committing to making all dispatch tables large enough to be able to obtain a perfect coloring for every program.

Each selector (interface method signature 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, Jalapeño maps ids directly to IMT slots, modulo the size of the IMT, a fixed constant.

Figure 1 depicts the virtual and IMT-based interface dispatching sequences on a PowerPC. The extra instruction in the IMT dispatch sequence prepares for the possibility of a collision by loading the id of the interface signature being dispatched into a *hidden parameter*. If there is no collision, then the IMT entry holds a reference to the executable code for the callee method, and the system would branch directly to it (the hidden parameter being ignored). If there is a collision, then the IMT entry holds a reference to a custom-generated conflict resolution stub. The stub uses the hidden parameter to determine which of the several signatures that share this slot in the receiver class's IMT is the desired target, and then loads the VMT offset for the appropriate method, and transfers control to the code referenced at this offset in the VMT.

Conflict stub code generation must proceed carefully, due to the restricted context in which the stub must execute. Before executing it, 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, and thus may use only a small number of registers without introducing save/restore overhead.

Figure 2 shows a conflict resolution stub for an IMT slot with four 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 registers (`s0`, `s1`, and `r0`) are readily available for use by the stub.[8] Although the details of the stub are highly architecture dependent, similar ideas apply on other platforms.[9]

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

Instead the system can build 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 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.

6. INLINING INTERFACE INVOCATIONS

The previous sections describe efficient schemes for interface method dispatch. However, interfaces could conceivably also inhibit compiler optimizations, method inlining in particular. This section discusses optimizations performed by the

[8] 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.

[9] 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.

```
//   virtual invocation sequence
//   t0 contains a reference to the receiver object
L    s2, tibOffset(t0)  // s2 := TIB of the receiver
L    s2, vmtOffset(s2)  // s2 := VMT entry for method being dispatched
MTLR s2                 // move target address to the link register
BLRL                    // branch to it (setting LR to return address)


//   IMT-based interface invocation sequence
//   t0 contains a reference to the receiver object
L    s2, tibOffset(t0)  // s2 := TIB of the receiver
L    s2, imtOffset(s2)  // s2 := IMT entry for signature being dispatched
L    s1, signatureId    // put signature id in hidden parameter register
MTLR s2                 // move target address to the link register
BLRL                    // branch to it (setting LR to return address)
```

Figure 1: Sequences for virtual and IMT-based interface dispatch

```
    //    id1 < id2 < id3 < id4
    //    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, id2           // compare hidden parameter to id of second method
    BLT   L1                // if less than branch to (id1..id1) search tree
    BGT   L2                // if greater than branch to (id3..id4) search tree
    L     s0, offset2(s0)   // load VMT entry for second method
    MTCTR s0                // move this address to the count register
    BCTR                    // branch to it (preserving contents of the LR)
L1: L     s0, offset1(s0)   // load VMT entry for the first method
    MTCTR s0                // move this address to the count register
    BCTR                    // branch to it (preserving contents of the LR)
L2: CMPI  s1, id3           // compare hidden parameter to id of third method
    BGT   L3                // if greater than branch to (id4..id4) search tree
    L     s0, offset3(s0)   // load VMT entry for the third method
    MTCTR s0                // move this address to the count register
    BCTR                    // branch to it (preserving contents of the LR)
L3: L     s0, offset4(s0)   // load VMT entry for the fourth method
    MTCTR s0                // move this address to the count register
    BCTR                    // branch to it (preserving contents of the LR)
```

Figure 2: A conflict resolution stub with four entries.

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 3: 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 3. 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 `i3.foo()`, since intra-procedural type analysis [27, 9] 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 (without guarding), 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 [13, 8], profile information [24, 21], and/or static examination of the program's class hierarchy [7, 11]. Jalapeño uses both class tests and method tests [12] to perform guarded inlining of virtual calls based on both class hierarchy analysis and on-line profile information.[10]

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 inter-procedural analysis to virtualize or devirtualize call sites for methods of these interfaces. Such analysis usually costs too much for a JIT or runtime compiler, which must normally rely on less expensive, solely intra-procedural 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.[11]

Profile-directed 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. If the heuristics do not inline a hot target method, deeming it

[10] 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 [12] as a partial substitute for a full-fledged invalidation mechanism.

[11] 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 example, figure 5). In this case, the compiler must insert a dynamic type check to ensure that the receiver implements the interface before executing the inlined body.

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

Figure 4: A common interface idiom requiring a single dynamic type check.

```
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 5: 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`. This situation can arise when the class `A` was modified to no longer implement `I` after the class files for `Test` and `B` were produced.

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 statically 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 4. Similarly, the compiler will optimize redundant loads in the dispatch sequence; the TIB base pointer load for object `e` in the figure can also be hoisted from the loop.

As discussed in Section 2, the virtual machine specification generates some fringe cases that the compiler must handle correctly. For example, in Figure 5, 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 intra-procedural 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`. The dynamic type check is required to detect and raise an IncompatibleClassChangeError if `createA` should ever return a subclass of A that does not implement `I`.

7. EXPERIMENTAL RESULTS

This section empirically assesses four interface method dispatch schemes: a naïve implementation, two itable variants, and IMTs. It also explores the effectiveness of the various optimizing compiler techniques described in Section 6. The next subsection precisely describes the implementation details of the four alternative schemes. The following subsection presents micro-benchmark results that focus on the direct costs of virtual and interface dispatch under each alternative scheme. The third subsection describes our suite of larger benchmarks and presents data on the dynamic frequency of interface invocation in each program. The final subsection first assesses the effectiveness of the compiler techniques for avoiding interface dispatching entirely, and then presents data comparing the various dispatching techniques focusing on their impact on application runtime and the space implications of each alternative.

All performance results reported below were obtained on an IBM RS/6000 Enterprise Server F80 running AIX v4.3. The machine has 4GB of main memory and six 500MHz PowerPC RS64 III processors each with 4MB of L2 cache.

7.1 Interface Dispatch Implementations

The alternative implementations of interface method dispatch are as follows.

- *Class Object Search*: a naïve implementation of the `invokeinterface` specification. On every interface method dispatch, the VM invokes a runtime service to find the target method. The service routine takes the receiver object and a description of the desired interface method, and searches the class hierarchy to find a matching virtual method. The search routine also performs the required dynamic type check.

- *Searched ITables*: the VM searches a per-class list of itables to find the appropriate itable, and then indexes into the itable to find the target method. On every interface method dispatch, a runtime service routine is invoked to find the desired itable. The service routine takes the receiver object and the id of the desired interface. It searches the class's list of itables (reachable from the TIB of the object); if it finds the itable then it simply returns it. If it fails to find the itable, then it must perform a dynamic type check to ensure that the class actually implements the interface. As a side-effect of performing the dynamic type check, the itable for the target interface is added to the class's list of itables. The search routine employs a move-to-front algorithm [14] to exploit temporal locality of interface usage and partially mitigate the search overhead when a class implements a large number of interfaces.

- *Directly Indexed ITables*: a per-class array of itables is loaded from the TIB and indexed into by interface id to find the appropriate itable, which is then indexed into to find the target method. This is one of the interface dispatching schemes used in the CACAO JVM [29]. The interface dispatch must be preceeded (either at compile-time or at run-time) by a dynamic type check to ensure that the class of the receiver object actually implements the target interface. As a side-effect, the dynamic type check adds the itable for the interface to the class's itables array. This mechanism ensures that the required type check has been performed for every populated itable.
- *IMT*: interface method tables as described in Section 5. As in the directly indexed itables, the IMT is lazily initialized as a side-effect of dynamic type checking. Data is reported for two different IMT sizes: 5 entries and 40 entries (*IMT-5* and *IMT-40*).

The fact that Jalapeño is implemented in Java adds an extra indirection of overhead to the itable and IMT dispatch implementations. The Jalapeño TIB is implemented as a Java array in the run-time system. As such, the TIB itself must conform to the object model, like any other Java object in the VM. As a result, Jalapeño does not have the ability to grow the TIB in two directions. So, Jalapeño cannot reference two variable-size tables (eg. the VMT and IMT, or VMT and itable list) with just one pointer in the object header. So, in all schemes described, the current implementation adds an extra indirection from a TIB entry to acquire the first level of interface dispatch data structure. We could eliminate this extra indirection by circumventing the Java object model for TIBs, and allow these structures to grow in two directions. We evaluate the cost of the extra indirection with a microbenchmark in the next section. A JVM implemented in C would not face this difficulty.

7.2 Micro-benchmarks

This Sections presents several micro-benchmarks to 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 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.

The micro-benchmarks call one of two target methods. The first target (*Trival Callee*) simply returns the integer constant 1. For this trivial 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 (*Normal 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 the 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 addition to the four interface dispatching schemes previously described, a fifth, *Embedded IMT*, is also included just for the micro-benchmarks. The *Embedded IMT* configuration simulates a runtime system in which the extra indirection imposed by Jalapeño's restriction to using Java objects for TIBs could be eliminated by growing the TIB in both directions, thus supporting a variable-size VMT and a fixed-size IMT that is only present for those classes that actually use the interfaces they implement. Based on the differences between the *Embedded IMT* and *IMT* data, it appears that the extra dependent load in the dispatching sequence adds approximately two cycles to the dispatching cost. This is further supported by the observation that the primary difference between a conflict-free IMT dispatch and a dispatch through a directly indexed itable is a single dependent load, and these data points also differ by approximately two cycles.

Overall, a conflict-free IMT-based interface method dispatch is the most efficient mechanism, followed closely by directly indexed itables. The difference between the conflict-free dispatch through an Embedded IMT and a virtual dispatch is insignificant (0.1 cycles). The hardware successfully overlaps the extra register move immediate to set up the hidden parameter with other operations, resulting in almost zero observed overhead.

Dispatch through a conflict resolution stub is surprisingly inexpensive. The cost of dispatching through a conflict resolution stub (even one with 20 entries) is roughly equivalent to the cost of a prologue/epilogue sequence. The difference between Trivial Callee and Normal Callee on *virtual* is 11 cycles; the difference on IMT-5 between a 1 method interface and 100 method interface is 12 cycles. IMT-based dispatch with conflicts is not as efficient as interface dispatch through a directly indexed itable. But, in a fairly typical case where the callee is non-trivial and the conflict stub only has to mediate between a small number of candidate methods (IMT-40, 100, Normal), the difference is only 5 cycles.

Both *Searched ITables* and *Class Object Search* are relatively slow interface dispatching mechanisms. But only *Class Object Search* is truly pathological with costs of 214x and 92x greater than a virtual dispatch. An interface dispatch using *Searched ITables* is only 9.5x and 4.6x slower than the equiv-

Dispatching Mechanism	Number of Methods in Interface	Trivial Callee	Normal Callee
virtual	Not applicable	8.13	19.18
Embedded IMT-5	interface with 1 method (no IMT conflict)	8.23	19.25
Embedded IMT-5	interface with 100 methods (20 element stub)	20.25	32.28
Embedded IMT-40	interface with 1 method (no IMT conflict)	8.23	19.25
Embedded IMT-40	interface with 100 methods (2 or 3 element stub)	14.23	27.40
IMT-5	interface with 1 method (no IMT conflict)	10.18	21.20
IMT-5	interface with 100 methods (20 element stub)	22.20	32.23
IMT-40	interface with 1 method (no IMT conflict)	10.18	21.20
IMT-40	interface with 100 methods (2 or 3 element stub)	18.20	28.23
Directly Indexed ITables	interface with 1 method	12.18	23.20
Directly Indexed ITables	interface with 100 methods	12.18	23.20
Searched ITables	interface with 1 method	77.45	88.50
Searched ITables	interface with 100 methods	77.45	88.50
Class Object Search	interface with 1 method	352.55	362.73
Class Object Search	interface with 100 methods	1,743.13	1,759.48

Table 1: Cost, in clock cycles, of round-trip method dispatch in Jalapeño, under each alternative interface dispatching mechanism.

alent virtual dispatch. This could potentially be improved further with inline caching techniques.

7.3 Application Characteristics

Table 2 describes the application benchmark suite, comprising the SPECjvm98 [41] benchmarks and several larger codes. Improving interface invocation will only help an application with non-negligible overhead due to interfaces. 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.[12] All macro-benchmarks run in a testing harness that executes the benchmark ten times, printing and clearing the counters at the start of each run.

Figure 6 shows the rate of non-inlined invocations per second on the tenth run for each of the four different `invoke` bytecodes. The bar for *compress* appears invisible because after inlining, *compress* makes only 249 method calls per second. Based on this data, we can expect 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. Therefore, results for *compress*, *mpegaudio*, and *mtrt* 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.

[12]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.

7.4 Application Performance

Figure 7 provides data on the effectiveness of the various compiler techniques to avoid performing a full interface dispatch. It shows 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

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 [42]	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 [40] 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.

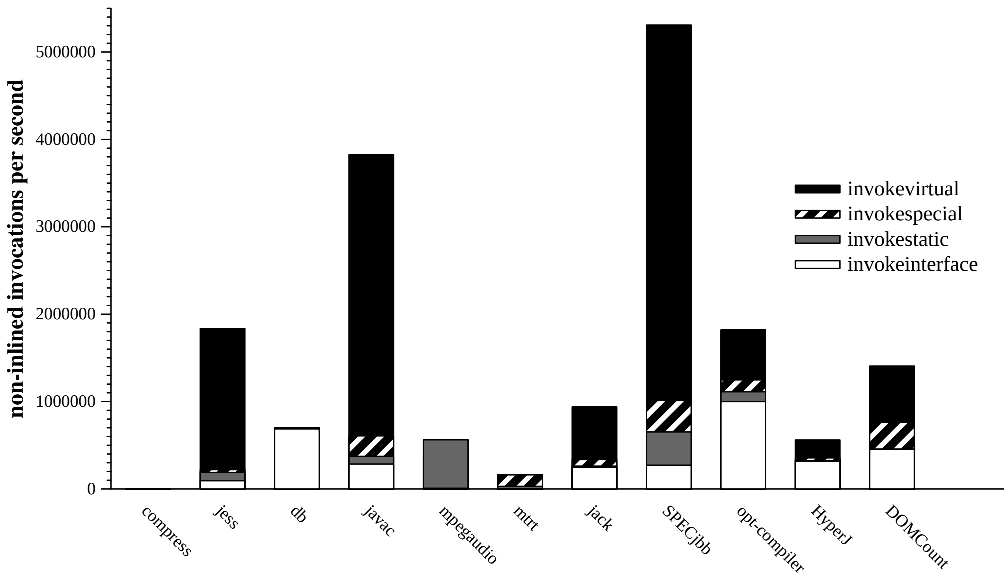

Figure 6: Dynamic rate of non-inlined invocations categorized by invoke bytecode. Optimizing interface invocation will only improve the performance of those applications with a significant rate of interface invocation (the bottom white portion of each bar).

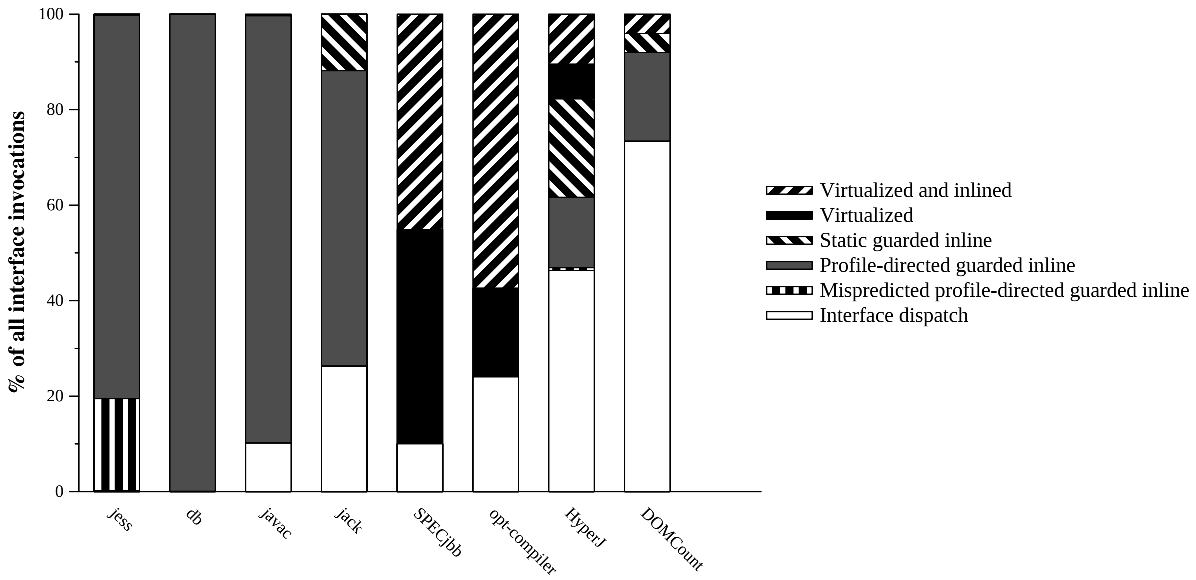

Figure 7: Dynamic percentage of interface invocations handled by each dispatch mechanism.

the single static target, guarded by a runtime check.[13]

- *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 runtime guard.
- *Mispredicted profile-directed guarded inline*: The runtime 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.
- *Interface dispatch*: No other technique applied, and the compiler resorted to a full-blown run-time method dispath.

The mix of dispatching mechanisms varies widely from program to program. For example, the fraction of invocations dispatched as full-blown run-time calls 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 inter-procedural 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. 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.

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. The compiler heuristics inline only the two most frequently invoked targets. Thus, any calls to the other 12 targets at that call site are mispredicted.

Figure 8 depicts the performance impact from alternative interface dispatching mechanisms. The Figure shows that the naïve *ClassObjectSearch* scheme can indeed significantly

[13] In principle, this runtime 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.

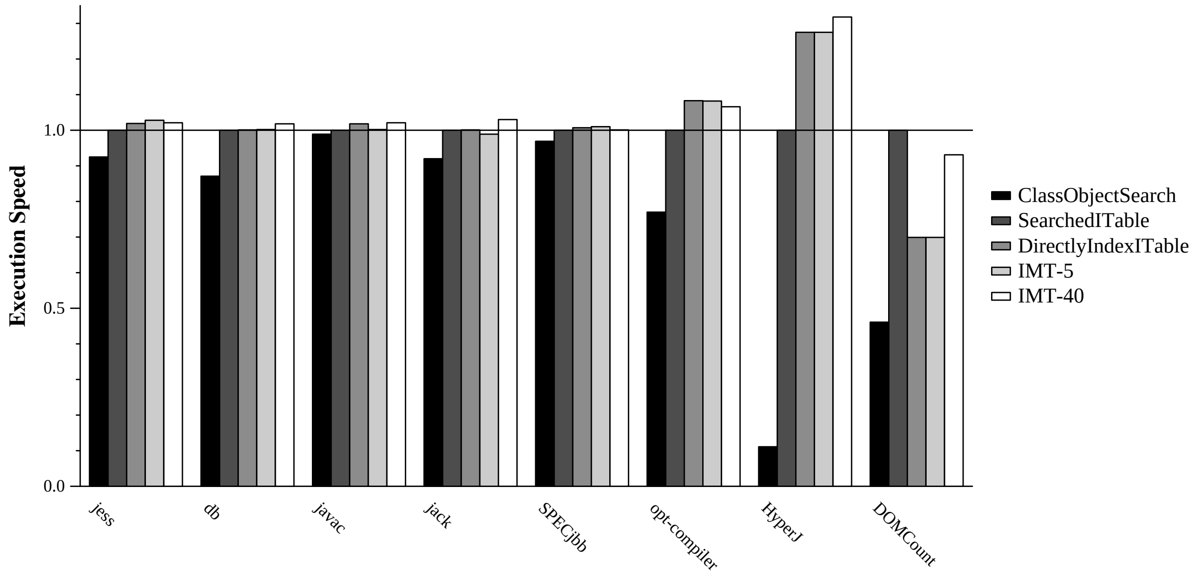

Figure 8: Performance impact of alternative interface method dispatching schemes. Speeds are normalized to that of *SearchedITable*.

hurt performance, slowing down *HyperJ* by a factor of 10. Among the two itable variants and the two IMT variants, each performs the best on at least one benchmark, and relative performance varies depending on the application.

The interface dispatch mechanism impacts performance most on the three largest benchmarks, *opt-compiler*, *HyperJ*, and *DOMCount*. On *opt-compiler* and *HyperJ*, directly-indexed itables and the IMTs significantly outperform the searched itable. Several factors could account for this difference; namely, polymorphic call sites that defeat the move-to-front search heuristic, along with fairly rich interface hierarchies. Note that the searched itable also under-performs on jess, which was previously discussed as having a hot polymorphic call site.

On the other hand, searched itables outperform the others on the *DomCount* XML parser benchmark. Possibly, the searched itable wins by combining the dynamic type check with the interface dispatch. Section 8 discusses a possible enhancement to the IMT dispatch sequence to gain the same benefit.

The *db* performance result is anomalous, since Figure 7 reports that practically all interface calls were inlined. We have occasionally observed unstable *db* performance on Jalapeño, possibly due to instability of TLB conflicts resulting from differences in memory layout [38].

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.

Figure 9 shows the space costs of the alternative interface dispatch schemes. The figure shows three categories of space usage:

1. *Data Structures:* This category represents the space allocated to data structures introduced to support interface dispatch. These data structures include itables, IMTs, itable dictionaries, and TIB slots holding pointers to interface dispatch structures.

2. *Conflict Stubs:* This category represents the space allocated to conflict resolution stub code in the IMT variants.

3. *Invocation Sequence:* This category represents space allocated for *inline* code inserted into compiled machine code to support interface invocation. This includes inline code for dynamic type checks, as well as inlined interface dispatch sequences.

The naïve *ClassObjectSearch* scheme introduces no extra data structure overhead, over and above class loader data structures required for other purposes.

The figure shows that in most cases, the space allocated for the inline invocation sequence exceeds space costs for the other two categories. *Jess* is an exception; as discussed previously, *jess* relies on few (only one hot) interface call sites, but uses a fairly large number of classes which implement an interface. This results in substantial space costs for the direct itable scheme. IMTs with 40 entries waste space compared to IMTs with only 5 entries, since most IMT entries are blank.

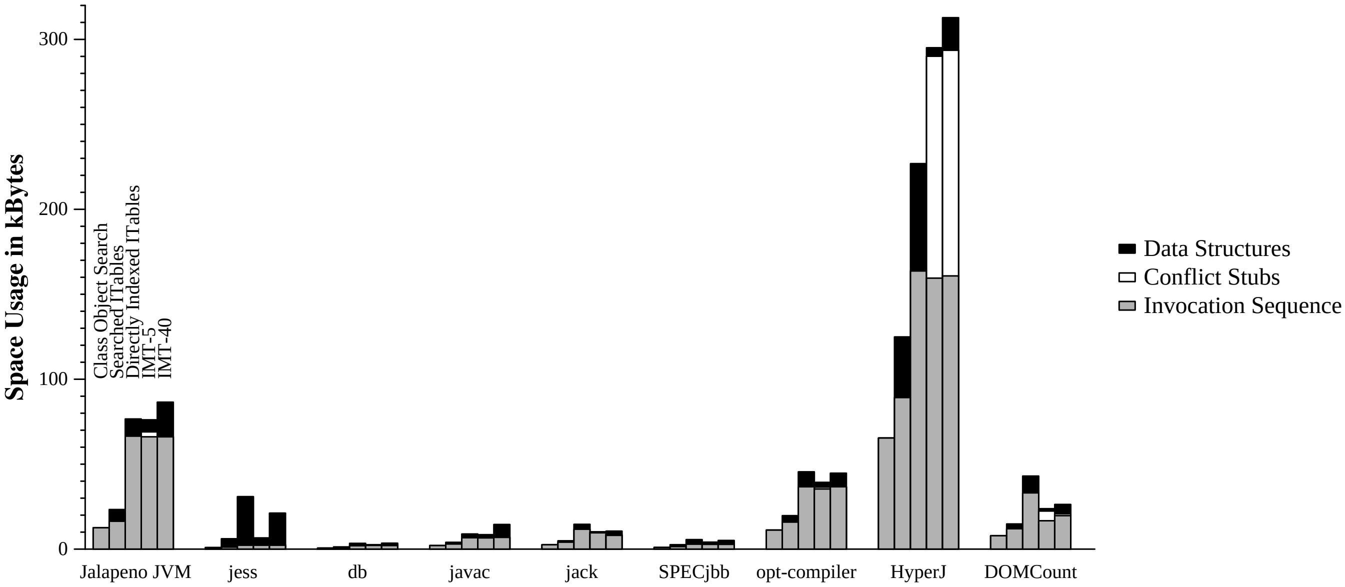

Figure 9: Space usage of alternative interface method dispatching schemes. Five bars are plotted for each benchmark, showing from left to right the space usage for *Class Object Search*, *Searched ITables*, *Directly Indexed ITables*, *IMT-5*, and *IMT-40*. Each bar is subdivided to show the code space consumed by the interface dispatching sequence, the code space taken by conflict resolution stubs, and the data space used for supporting data structures. The first set of bars labeled *Jalapeño JVM* reports the space used to support interface dispatching for the approximately 900 classes that comprise Jalapeño itself. This fixed space cost was deducted from the bars for each benchmark, so that the bars for each benchmark represent the space usage induced by the application code over and above the cost for a bare Jalapeño VM.

HyperJ stands out for its large space usage. Both IMT-5 and IMT-40 pay a significant space cost for conflict resolution stubs. It turns out that some classes in HyperJ implement interfaces with over 100 methods. For these cases, no coloring scheme will avoid saturating even the 40-slot TIB, resulting in conflicts at every TIB slot. So, both IMT-40 and IMT-5 result in the same number of total targets embodied in conflict resolution stubs. However, IMT-40 results in conflict stubs with fewer entries, resulting in improved run-time performance in Figure 8. HyperJ also demonstrates some of scalability problems with direct itable dispatch, as the direct itable data structure space cost is significant. However, this cost is dominated by the inlined invocation sequence space; so, the direct itable scheme appears to be practical on this platform for at least medium-sized applications.

8. FUTURE WORK

The IMT mechanism enjoys fixed-size tables, at the cost of extra space and time for conflict resolution stubs. Jalapeño could improve on its current (trivial) algorithm for selector coloring by analyzing off-line a collection of standard Java classes to discover sets of interfaces that are simultaneously implemented by classes. Based on this information and profile data, a standard register allocation algorithm could minimize the expected dynamic number of collisions in a fixed IMT size. During JVM execution new (unexpected) interface method signatures could be assigned to empty (or infrequently-used) slots in an *expanded* IMT that would only be associated with those classes that implemented the unexpected interfaces.

This approach would naturally be limited by the set of classes available for off-line analysis. Alternatively, the adaptive optimization system could color interface method signatures based on on-line profile information, although removing the assignment of a selector to a color on-line could entail significant code-patching.

An anonymous reviewer, borrowing an idea from polymorphic inline caches, proposed folding the dynamic type check into the conflict resolution stubs. The stub shown in Figure 2 assumes that the only possible values for the hidden parameter are the four interface method signatures that it was generated to handle. The stub could instead be generated to treat unexpected values of the hidden parameter as a signal that a dynamic type check needs to be performed and the stub extended.

This approach would require all interface method dispatches to go through conflict resolution stubs, even if only one method was actually mapped to a slot. Although the expected cost of a stub and a dynamic type check are similar, the conflict stub is invisible to the compiler and thus not easily amenable to optimizations such as code motion and partial redundancy elimination. On the other hand, in our experiments Jalapeño's optimizing compiler was able to eliminate the dynamic type check from fewer than 20% of the interface calls that were actually dispatched through the IMT.

While our profile-directed inlining handles interfaces as effectively as virtual calls, both suffer the overhead of a runtime guard. We are also currently implementing an invalidation mechanism to allow us to omit the guards on inlined interface and virtual call sites, in order to mitigate the penalty of failed devirtualization.

9. CONCLUSIONS

An early conference paper on the Jalapeño runtime [2] admitted that "[w]e don't make much use of `interface`s because the performance overhead was too high to use it to call frequently executed methods." 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 provided the impetus for the work reported here.

This paper investigates three potential sources of interface inefficiency: implicit dynamic type checking, actual interface method dispatch overhead, and possible opportunity costs of forgone optimizations. Type checking overhead is real, but quite small, and can often be optimized away (usually in conjunction with other optimizations). The interface method table (IMT) mechanism for dispatching interface methods is only a few cycles more expensive than its counterpart for virtual methods. The third problem proved illusory: the Jalapeño optimizing compiler performs similar optimization at both virtual and interface method dispatch sites.

A number of different interface dispatch mechanisms were compared. Both IMTs and direct itables provide good performance with a moderate space overhead. Only the naive class-object search scheme provided truly atrocious performance.

Although early implementations of Java interfaces did not perform well, their reputation as being inherently inefficient is undeserved. A Java programmer, or program generating system, should feel free to fully exploit interfaces without concern for 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 anonymous reviewers of this and of an earlier paper. Thanks also to the OTI VAME team for providing an implementation of the Java class libraries.

10. 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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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).*

[9] 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.

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

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] E. Gagnon and L. Hendren. SableVM: A Research Framework for the Efficient Execution of Java Bytecode. In *USENIX Java Virtual Machine Research and Technology Symposium*, Apr. 2001.

[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] T. A. Proebsting. imple and efficient burs table generation. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 331–340, June 1992. *SIGPLAN Notices* 27(6).

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

[38] Y. Shuf. Personal Communication, 2001.

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

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

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

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

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

[44] 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.

[45] 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.