

A Study of Devirtualization Techniques for a Java™ Just-In-Time Compiler

Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, Toshio Nakatani

IBM Research, Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato-shi, Kanagawa-ken, 242-8502, Japan

ishizaki@trl.ibm.co.jp

ABSTRACT

Many devirtualization techniques have been proposed to reduce the runtime overhead of dynamic method calls for various object-oriented languages, however, most of them are less effective or cannot be applied for Java in a straightforward manner. This is partly because Java is a statically-typed language and thus transforming a dynamic call to a static one does not make a tangible performance gain (owing to the low overhead of accessing the method table) unless it is inlined, and partly because the dynamic class loading feature of Java prohibits the whole program analysis and optimizations from being applied.

We propose a new technique called *direct devirtualization with the code patching mechanism*. For a given dynamic call site, our compiler first determines whether the call can be devirtualized, by analyzing the current class hierarchy. When the call is devirtualizable and the target method is suitably sized, the compiler generates the inlined code of the method, together with the backup code of making the dynamic call. Only the inlined code is actually executed until our assumption about the devirtualization becomes invalidated, at which time the compiler performs code patching to make the backup code executed subsequently. Since the new technique prevents some code motions across the merge point between the inlined code and the backup code, we have furthermore implemented recently-known analysis techniques, such as type analysis and preexistence analysis, which allow the backup code to be completely eliminated. We made various experiments using 16 real programs to understand the effectiveness and characteristics of the devirtualization techniques in our Java Just-In-Time (JIT) compiler. In summary, we reduced the number of dynamic calls by ranging from 8.9% to 97.3% (the average of 40.2%), and we improved the execution performance by ranging from -1% to 133% (with the geometric mean of 16%).

1. Introduction

Many devirtualization techniques [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11] have been proposed to reduce the overhead of dynamic method calls for various object-oriented languages. In general, a guard test is generated to test the receiver of the class (called *class test*) [1, 2, 3] or the method (called *method test*) [11] to ensure that it is valid

Permission to make digital or hard copies of part or all of this work or personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA '00, 10/00 Minneapolis, MN, USA
© 2000 ACM ISBN 1-58113-200-x/00/0010...\$5.00

to make a direct call to the corresponding target method. We call this approach *guarded devirtualization*. In dynamically-typed object-oriented languages such as Self [4], guarded devirtualization is extremely effective because of the high overhead of a dynamic method call. In statically-typed object-oriented language like Java [12], guarded devirtualization is less effective because of the low overhead of dynamic method call due to the fact that it can be translated into a few load operations followed by an indirect jump operation. In order to boost the performance with a dynamic method call, the method call must be inlined as much as possible after the guard test is eliminated.

To devirtualize a dynamic method call without generating a guard test (we call this approach *direct devirtualization*), the whole program analysis and optimizations [5, 6, 7, 8, 9] have been proposed in the context of static compilers. However, they are based on the *closed-world* assumption, in which no dynamic class loading is allowed. Therefore, these techniques cannot be directly applicable to Java. Dynamic recompilation can be used to make direct devirtualization possible in the non-closed world assumption, but it involves a complicated mechanism called *on-stack replacement* [13].

We propose a new technique called *direct devirtualization with the code patching mechanism* (the *code patching mechanism* or *code patch* in short) [10]. For a given dynamic call site, our compiler first determines whether the call can be devirtualized, by analyzing the current class hierarchy. When the call is devirtualizable and the target method is suitably sized, the compiler generates the inlined code of the method, together with the backup code (also called *backup path*) of making the dynamic call. Only the inlined code is actually executed until our assumption about the devirtualization becomes invalidated, at which time the compiler performs code patching to make the backup code executed subsequently. Since the new technique prevents some code motions across the merge point between the inlined code and the backup code, we have furthermore implemented recently-known analysis techniques, such as type analysis [14, 15, 16, 17] and preexistence analysis [11], which allow the backup code to be completely eliminated.

We made various experiments using 16 real programs to understand the effectiveness and characteristics of the devirtualization techniques in our Java Just-In-Time (JIT) compiler. In summary, we reduced the number of dynamic calls by ranging from 8.9% to 97.3% (the average of 40.2%), and we improved the execution performance by ranging from -1% to 133% (with the geometric mean of 16%).

This paper makes the following contributions:

- A new devirtualization technique called *direct devirtualization with the code patching mechanism*, which is much simpler to implement and has less overhead to execute than a re-

compilation approach with on-stack replacement. We eliminate the backup path by type analysis and preexistence analysis. We also optimize the inlined code by inserting compensation code in the backup path if it is not eliminated.

- Evaluation of various devirtualization techniques implemented in our Java JIT compiler, including direct devirtualizations (the code patching mechanism, type analysis, and preexistence) and guarded devirtualizations (class test and method test).

The rest of the paper is structured as follows. Section 2 discusses related work. Section 3 describes our devirtualization techniques that implemented in our Java JIT compiler. Section 4 gives experimental results with statistics and performance results on a set of real programs. Finally, Section 5 outlines our conclusions.

2. Related Work

Devirtualization techniques are important to improve the performance in object-oriented languages. Therefore, many devirtualization techniques have been proposed.

An inline cache technique was developed to speed up dynamic method calls. An inline cache records the class of the last receiver object at the call site, and jumps directly to the method for that class. A stub validates that the dynamic type of the receiver matches the expected type. If this test fails, a normal method lookup makes a dynamic method call and stores the class of the current receiver to the call site cache. Hölzle extended the technique to a polymorphic case of inline caches [18]. Type prediction [2, 3] and method test [11] have also been proposed. Type prediction and method test predict the type of a frequently-called object at compile time. A polymorphic inline cache, type prediction, and method test introduce new runtime tests, since these techniques take the cache approach with memory references. According to the results of simple experiments [19], type prediction without inlining, even with 100% accuracy of the predictions, cannot outperform direct devirtualization of dynamic method calls without inlining. Type prediction with inlining must achieve approximately 90% accuracy to outperform direct devirtualization without inlining. Finally, no known technique can outperform direct devirtualization with inlining. Lee et al. implemented both monomorphic and polymorphic inline caches in a Java Virtual Machine [20]. The experimental results did not achieve as good a speedup as in dynamically-typed object-oriented languages such as Self. In implementations of Java, the cost of dynamic method calls is not so different from that of using polymorphic inline caches, type prediction, and method test. In statically-typed object-oriented languages, guarded devirtualization is effective in enabling inline methods with dynamic calls to expand the intra-procedure optimization scope of a compiler.

Several systems directly devirtualize dynamic method calls, by allowing them to be inlined or implemented by direct method calls. Dean et al. used a class hierarchy analysis to devirtualize dynamic method calls [5]. Class hierarchy analysis determines when the static type of a receiver implies that an invoked method has only a single implementation in the set of classes used in a whole program. Fernandez [6] proposed a link-time optimization system. Bacon and Sweeney [7], Tip and Palsberg [8], and Vijay et al. [9] proposed more precise static analysis. All these techniques statically devirtualize dynamic method calls based on a closed-world assumption. Since Java supports dynamic class loading, these techniques cannot be used in a straightforward manner. This is why we propose direct devirtualization with the code patching mechanism [10]. Flow-sensitive type analysis [14, 15, 16,

17] attempts to tighten the static type constraints on the receiver expressions. It increases the opportunities for direct devirtualization to determine whether a call site has a single implementation. It can also directly devirtualize a dynamic method call without a backup path.

Several languages, such as C++, Dylan, and Java, have a linguistic mechanism that allows users to declare a class sealed, so that it is prohibited to subclass any new class from it. However, sealed methods are not common in the Java Core libraries such as `java.util.Vector`. Most of the methods in this class are not sealed in Java 2.

The Self system performs extensive inlining of dynamic method calls [13], whose correctness is ensured by the on-stack replacement mechanism. In the compiled code, there are deoptimization points at which the original state of the method's variables can be recovered from the optimized state. When a compilation assumption is violated by dynamic class loading, the Self system recovers the original state at a deoptimization point and recompiles the method. Such a system introduces several concerns. In the Self implementation, the compiler produces numerous data structures called scope descriptors for deoptimization. Deoptimization points also introduce inefficiency into the generated code to storing extra data in memory to recover the original context. The compiler cannot reorder instructions over a deoptimization point. It is also difficult to replace methods on stacks in the multi-threaded runtime environment. The Java HotSpot compiler [21] adopts a re-compilation approach using on-stack replacement. We did not explore this approach because of the complexity of its implementation. Preexistence analysis [11] is an approach to prevent on-stack replacement by determining whether direct devirtualization can be performed based on the analysis of the receiver expressions. We adopted it to increase the opportunity for compiler optimizations by eliminating backup paths.

3. Devirtualization of Dynamic Method Calls

We present an overview of our devirtualization approach. First, the compiler performs flow-sensitive type analysis and preexistence analysis to directly devirtualize call sites without backup paths that introduce constraints on compiler optimizations such as code motion. The preexistence analysis also guarantees that the situations requiring on-stack replacement cannot occur. Next, the compiler performs dynamic class hierarchy analysis to directly devirtualize the dynamic method calls. It can detect when a call site has a single implementation at compile time and inline the callee code without any guard tests. However, Java allows new classes to be loaded during the execution of a program, and therefore the compiler has to prepare the original dynamic method call to allow for execution where the assumption of a single implementation is violated. Finally, if the compiler knows a call site has multiple implementations, it devirtualizes a dynamic method call with a guard test. It inlines the selected dynamic method call with a class test verifying that the receiver is of the proper class, or it inlines a dynamic method call with a method test verifying that the receiver has the proper method.

In the rest of this section, we describe devirtualization techniques for the optimization of dynamic method calls: the code patching mechanism, flow-sensitive type analysis, preexistence analysis, class test, and method test.

3.1 Code Patching Mechanism

Class hierarchy analysis (CHA) [5, 6] determines a set of possible targets of a dynamic method call by combining the static type of an object with the class hierarchy of the whole program. If it can

be determined that there is no overridden method, the dynamic method call can be replaced with inlined code or with a direct method call by direct devirtualization at compile time, and the method can be executed without method lookup. Previously, direct devirtualization with CHA has been investigated and implemented for languages limited to static class loading, in which the class hierarchy does not change during the execution of the program. However, Java supports dynamic class loading, which allows the class hierarchy to change during the execution of a program. Recompile with on-stack replacement was the only an approach to allow a compiler to invalidate the devirtualized method while there is an active context on stacks. However, the implementation is very difficult because it requires replacing methods and context on stacks.


We have designed the code patching mechanism in order to directly devirtualize dynamic method calls with dynamic class loading [10]. If a caller site has a single implementation, the compiler generates the inlined code of the target method and the original dynamic method call at compilation time. The compiler can also turn dynamic method calls into direct method calls for non-inlined methods. If runtime class loading overrides a method that was previously not overridden, the inlined code sequence for a specific implementation must be replaced with the original dynamic method call. This is done by rewriting the first instruction in the inlined code sequence. We show the generated code for an inlined method in Example 1 using the PowerPC instruction set. The generated code using direct devirtualization has no overhead at execution time because there are no tests requiring memory access in the method tests and class tests. This mechanism also has lower overhead and implementation costs than a recompilation approach with on-stack replacement. Further, when the compiler generates the native code, it places the inlined code in the fall through path in Example 1. Since it knows the inlined code is executed very frequently, this improves the efficiency of the instruction cache.

Java provides interfaces for the provision of multiple inheritances. The compiler also optimizes an interface method call by replacing it with inlined code. If CHA finds that only one class implements an interface class, a virtual method call with a single method lookup can be generated by using the implementation class as a static type. Furthermore, if the target method is not overridden anywhere in the implementation class hierarchy, the code can be


inlined instead of using the interface method call by using direct devirtualization. As a result, the generated code using the PowerPC instruction set is shown in Example 2. When the method is overridden in the implementation class hierarchy, the code patching mechanism cancels the direct devirtualization to execute the virtual method call. In addition, when non-subclass of the implementation class implements the interface class, the code patching mechanism cancels direct devirtualization to execute the original interface method call. The right column in Example 2 shows the latter case. This optimization is much more efficient than a naive implementation of an interface call, which requires executing a loop to search for an implementation class.

We have implemented the code patching mechanism using CHA for supporting dynamic class loading as follows. When the new class is loaded at runtime, the runtime routine refers to an internal data structure that represents whether or not each associated method is overridden. Furthermore, if a class implements an interface class, the compiler also counts the number of implementation classes of the interface class in order to devirtualize interface method calls. The compiler checks whether a caller site has a single implementation when it attempts to inline a dynamic method call. The result (whether or not the method has only a single implementation) is checked on demand, when the first check is issued, and the result is then stored in the result cache. When the native code is generated for the inlined code, the top address of the inlined code sequence is also recorded in the result cache for the call site. When the compiler next checks the implementation of the same method, the result is returned from the result cache immediately, and the new code address is also recorded in the result cache.

When the method is not yet overridden in the left column in Example 1, the inlined code is executed and the *italicized code sequence* for the dynamic method call is not executed at all. When the method is overridden because of dynamic class loading, the internal structures are updated appropriately. If the method related to the result cache is overridden, the class loader uses the result cache to find the address that should be replaced with a **b** (branch) instruction to the dynamic method call. This effectively undoes the direct devirtualization, and the inlined code becomes inaccessible. Consequently, the code sequence for the dynamic method call will be executed correctly.

<pre> Before overriding the method // top word of inlined code // the rest of inlined code after_inline: : original_call: lwz r1, (obj) lwz r2, offset(r1) lwz r3, offset(r2) mtctr r3 blr ctr b after_inline </pre>		<pre> After overriding the method b original_call // static jmp // the rest of inlined code after_inline: : original_call: lwz r1, (obj) // load class pointer lwz r2, offset(r1) // load method pointer lwz r3, offset(r2) // load code address mtctr r3 blr ctr // dynamic method call b after_inline </pre>
---	---	---

Example 1: Inlining of a dynamic method call (invokevirtual)

<pre> Only one class implements // top word of inlined code // 2nd word of inlined code // the rest of inlined code after_inline: : virtual_call lwz r1, (obj) lwz r2, offset(r1) lwz r3, offset(r2) mtctr r3 blr ctr b after_inline interface_call: mr r1, <rcv obj reg> blr rt_interface b after_inline </pre>		<pre> More than one class implements b interface_call // static jmp // 2nd word of inlined code // if implementing method is // the rest of inlined code // overridden, go to virtual_call after_inline: : virtual_call lwz r1, (obj) // load class pointer lwz r2, offset(r1) // load method pointer lwz r3, offset(r2) // load code address mtctr r3 blr ctr b after_inline interface_call: mr r1, <rcv obj reg> // move receiver object blr rt_interface // call runtime for interface call b after_inline </pre>
--	---	--

Example 2: Inlining of a dynamic method call (invokeinterface)

Since Java is an explicitly multi-threaded language, the invalidation of the old code sequence must be thread-safe. On the PowerPC RISC architecture, since the memory system guarantees atomicity only for full word instructions, we implemented this atomic updating by rewriting only one full word instruction, the branch instruction. Furthermore, on a processor with split caches for instruction and data, both the data and instruction caches should be flushed. This ensures synchronization between the instruction and data caches. In addition, any instruction prefetch buffer must be flushed [22]. This ensures that any pending instructions fetched from the instruction cache are ignored and the updated instruction will be executed. On the IA32 architecture, the memory system guarantees atomicity for a single write to memory aligned on a boundary of its length within a cache line [23], and the write instruction invalidates the instruction prefetch queue. Since the length of each instruction varies, the length of the rewrite target must be same as that of the new instruction to ensure the validity of the instruction sequence. If the length of a new branch instruction is two bytes, the compiler for the IA32 architecture generates the first instruction whose length is two bytes or over in the inlined code sequence, by means of padding using a special addressing mode. The new branch instruction is written by a single write to memory to ensure the atomicity. If the length of the branch instruction is five bytes, we replace the first two bytes of the first instruction in the inlined code sequence with spinning jump by a single memory write not using `xchg` instructions. Then, the rest three bytes of the first instruction are updated. Finally, the first two bytes of the branch instruction are written by a single memory write [24].

Going beyond our previous method [10], from the viewpoint of compiler optimizations, we now use an explicit intermediate representation of a branch affected by the code patching mechanism. This increases the opportunities for compiler optimizations. On the other hand, the branch may prevent the compiler from performing optimizations using dataflow analysis, because the generated intermediate representation includes a backup path (the original method call) as a kill point¹ due to its side effect. Scalar replacement of instance variables and code motion may also be restricted. We illustrate these problems in Example 3. A compiler translated a source program in Example 3 a) into a RISC-like intermediate representation in Example 3 b). Here, some instructions related to a loop exit are omitted for simplicity. In the example, the `getField` bytecode instructions

ple, the `getField` bytecode instructions are split into null-check instructions that are the potentially excepting instructions (PEI) in the **bold font** and `getField` instructions that are simple loads from a heap memory. At the end of basic block (BB) 3, there are two branches for direct devirtualization. One is a branch to BB3, which is a primary execution path. The other is a branch to BB5, which is a backup path.

In the Example 3 b), the compiler performs optimizations [25] as follows:

1. The compiler can perform nullcheck optimizations. It moves ‘nullcheck LO0’ out of BB3 and BB5 in the loop to BB1. Then, the compiler eliminates ‘nullcheck LO0’ in BB 4 since ‘nullcheck LO0’ instruction in BB 1 dominates it.
2. The compiler can perform partial redundancy elimination (PRE) [26]. It performs scalar replacements of the access of instance variables <x> and <z>. The compiler moves a `getField` instruction for <x> and <z> out of the loop, and then replaces the reference to variable <x> with a temporary variable in BB 3. Since the compiler moved the `getField` instruction for <x> and <z> across an `invoke` instruction that has side effects, it generates the compensation code for variable <x> and <z> after the kill point (`invoke` at BB5). Though it increases the inefficient code, it does not matter since it is generated in a backup path where it rarely executes.

Code motion involving PEIs or instructions with side effects is limited and cannot cross over a kill point. If the ‘nullcheck LO1’ instruction is moved into BB3, the exception may be thrown before throwing an exception raised within the method `m`. This transformation violates the original semantics of the program. Therefore, the compiler cannot move the ‘nullcheck LO1’ instruction across BB5.

Generating compensation code can reduce the impact of merging the control flow on compiler optimizations such as scalar replacement along a backup path. The compiler could also perform escape analysis [27, 28] with stack object allocation by generating the compensation code in backup paths. By flow-sensitive type analysis and preexistence analysis in Section 3.2 and 3.3, we eliminate backup paths for devirtualized method calls.

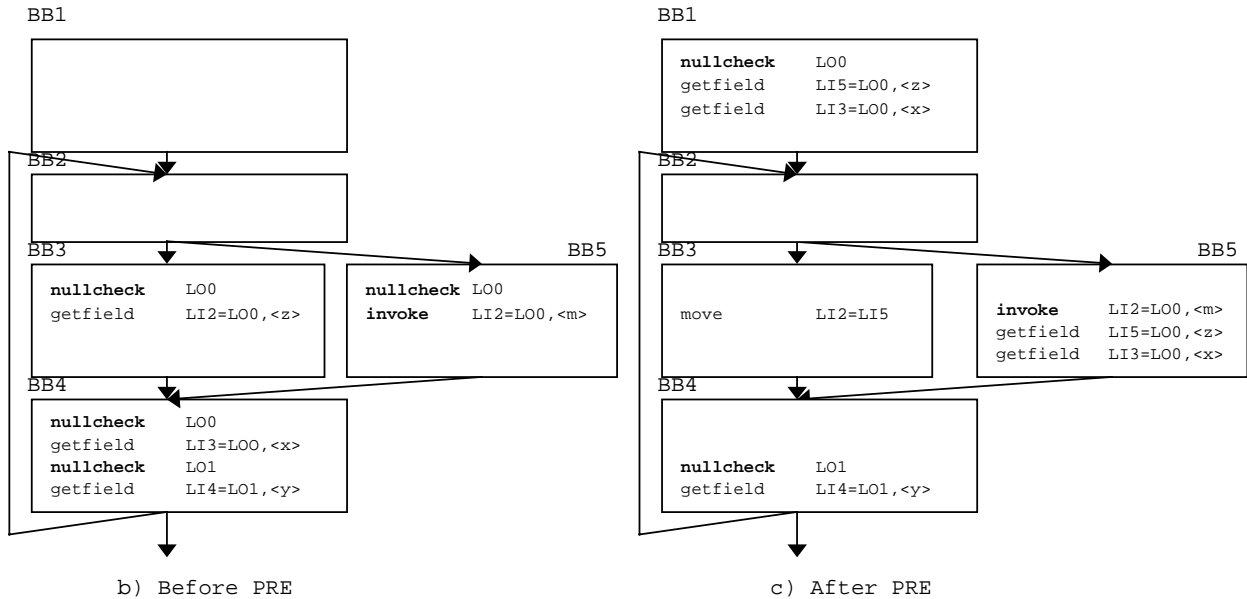
¹ If an instruction redefines a value, it is said to kill the definition, which means the collected information on the variable cannot be preserved before and after that point.

```

class Foo {
  int x, y, z;
  int m() {
    return this.z;
  }
  int caller(Foo a, Foo b) {
    do {
      i = a.m();      // BB3 and 5
      j = a.x;        // BB4
      k = b.y;        // BB4
    } while (cond)
    return i+j+k;
  }
}

```

a) Source code



Example 3: Partial redundancy elimination and code motion.

3.2 Flow-Sensitive Type Analysis

Flow-sensitive type analysis [14, 15, 16, 17] computes a type for every object reference point in an entire method. The compiler computes the dataflow information on static types with signatures and class instantiations (call to `new()`) at each object reference point. The analysis determines the set of classes reachable at each object reference point.

If the type analysis proves that all class instantiations that reach the receiver expression of the dynamic method call have the same definition, this proves that no method override occurs at the call site. Therefore, the dynamic method call can be directly devirtualized without a backup path. This reduces optimization constraints in our devirtualization techniques.

Type analysis can also recover missing type information. This loss can occur while translating source code into bytecode [11] (i.e. during a compilation by javac or jikes [29] (version 1.06)). We here explain it using Example 4. The source code of the method

`m()` indicates that the method call `a.equals()` invokes the method `equals()` in the class `A`. The javac compiler embeds the class `Object` and the method `equals()` as static types in the class file. The compiler may recover the more precise type `A` of the receiver through an interpretation like the bytecode verification process [30]. The missing type information causes the class hierarchy analysis to fail at the method call `a.equals()`. Without doing type analysis, the compiler checks whether `Object.equals()` is a single implementation rather than `A.equals()`. This always fails because the method in the class `String` that is never invoked by the method call `a.equals()` overrides the method `equals()`.

In practice, the missing type information frequently occurs at call sites involving the methods `equals()` and `hashCode()` that are declared in the class `java.lang.Object`. Therefore, type analysis improves the accuracy of class hierarchy analysis.

```

class Object { boolean equals(Object o) { ... }; }

class String extends Object {
    boolean equals(Object o) { ... };           // Overrides equals()
}

class A extends Object { ... }                // Does not override equals()

class X {
    void m(A a) {
        a.equals();
    }
}

```

Example 4: Missing type information at a call site

The methods `hashCode()`, `toString()`, and `equals()` are declared in the top-most class `java.lang.Object`, so they can be invoked on all objects. These methods are frequently called with the class `java.lang.Object` as a static type. The `hashCode()` method in several primitive classes such as `java.lang.Integer` and `java.lang.String` overrides the declaration from the class `java.lang.Object`. However, since these implementations are very simple and declared as `final`, if type analysis proves the static type of a dynamic method call is one of them, the compiler can inline these methods directly. Unfortunately, since the static type of a dynamic method call to `equals()` is frequently `java.lang.Object`, the compiler directly devirtualizes the method call using the code patching mechanism.

3.3 Preexistence Analysis

The concept of preexistence [11] is that if the receiver object for a method call has been allocated before the invocation of a caller method, then the method will not be overridden during the execution of the caller. This property can be used to directly devirtualize a dynamic method call without a backup path. This requires that the caller method must be recompiled with class hierarchy analysis at the next invocation in which the target method is overridden. However, it guarantees that such recompilation does not require on-stack replacement. This reduces the difficulty of the implementation.

We have implemented invariant argument analysis [11] to check for the preexistence of a receiver expression. If the receiver expression of a method call is shown to preexist and CHA shows that the method call has only a single target at compilation time, the compiler can directly devirtualize the method call without the backup path. It has two advantages. One is that it enables code motion involving potentially excepting instructions or instructions with side effects. The other is that the result of flow-sensitive type analysis is more accurate, because the merge point that creates the union type is removed from the control flow graph. Another solution for more precise flow-sensitive type analysis is message splitting [31]. It may increase the code size significantly because it

requires copying parts of the control flow, and therefore we did not pursue this alternative.

3.4 Class Tests and Method Tests

In previous research, most systems that use guarded devirtualization produced inlined code with a class test verifying that the receiver has the proper class. The class test [1, 2, 3] imposes a requirement that each object contains a pointer to its class information. The method test [11] makes a further assumption that the class information includes the method information. We show such code generated at an inlined call site in Example 5 a) and b) that appeared in [11].

Method test is more accurate than class test. Even when a class that does not override a method is tested by a class test, if the class is different from the particular class of the inlined method, the test fails and a dynamic method call is invoked. In a similar situation involving method invocation, the method test may succeed and the inlined code can be executed. Therefore, we have used method test at call sites that have multiple implementations, along with class hierarchy analysis at compilation time. The overhead of method test is slightly greater than that of class test. Our JIT compiler explicitly uses two loads to get class information and method information in the intermediate representation. This allows us to include these instructions in the scope of optimizations such as common subexpression elimination and code motion, and this can hide the overhead of method test.

These techniques also introduce some constraints on compiler optimizations because they have a merge point of the control flow on compiler optimizations for any backup path that includes an original method call. The constraints can be reduced by the same techniques described in Section 3.1.

We have used a class test for optimizing method calls within the class itself. If a compiler detects many dynamic calls with the caller's object as a receiver, the compiler provides two copies of the part. A class test with the receiver object is generated to determine which copy is executed. Two versions of the method are then generated: one for optimized calls within the class itself, and the other version is for the general case as it appeared in the original code, as shown in Example 6.

```

r0 = <receiver object>
r1 = load(r0 + <offset-of-class-in-object>)

if (r1 == <address-of-proper-class>) {
  <inlined code>
} else {
  r2 = load(r1 + <offset-of-method-in-class>)
  call r2
}

```

a) pseudo code of a class test

```

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

```

b) pseudo code of a method test

Example 5: Pseudo code for class test and method test [11]

```

class R {
  void bar() {...}
  void foo() {
    for (...) {
      this.bar();
      this.bar();
    }
  }
}

```

a) Original code

```

class R {
  void bar() {...}
  void foo() {
    if (classtest(this, R)) {
      for (...) {
        R.bar(); // a direct call
        R.bar(); // a direct call
      }
    } else {
      for (...) {
        this.bar();
        this.bar();
      }
    }
  }
}

```

b) Optimized code with a class test

Example 6: Optimized calls within the class itself

4. Experiments

In this section, we evaluate the characteristics and effectiveness of the devirtualization techniques in our system. Section 4.1 explains the system used in our experiments. Section 4.2 gives an overview of the programs used in our experiments. Section 4.3 shows the characteristics of the non-devirtualized programs. Section 4.4 shows the results by applying each of devirtualization techniques cumulatively. Section 4.5 discusses the evaluation of the results. Section 4.6 shows the performance results.

4.1 System

Our experiments were performed using a prototype version of the IBM Developers Kit for AIX, Java Technology Edition, Version 1.3. We have implemented the devirtualization techniques we described here in our Just-In-Time Compiler [32]. The JIT compiler is a highly optimizing compiler that uses a register-based intermediate representation. Register-based representations provide greater flexibility for code transformations than stack-based representations. The JIT compiler performs static method inlining, devirtualization, dataflow optimizations, loop optimizations, and low-level optimizations. Dataflow optimizations are copy propagation, constant propagation, dead code elimination, common subexpression elimination, scalar replacement, and elimination of redundant exception checks [25]. The loop optimization uses loop versioning. Low-level optimizations are register allocation, instruction scheduling, and shrink wrapping [33].

The JIT compiler inlines methods except when they have exception handlers or they are larger than the predefined maximum size. It inlines both static and dynamic method calls for up to four

nested levels in each call hierarchy tree. Here, dynamic method calls mean virtual and interface method calls. Since the JVM in the Sun SDK reference implementation must be able to traverse the original call stack in order to get the caller class at runtime, we have implemented a subset of the scope descriptor [13] just to recover the original call stack from the inlined call stack. This allows the compiler to inline methods extensively. Though the JIT compiler has a selective compilation mechanism, all the measurements were performed by compiling all methods.

The measurements were performed on an IBM RISC System 6000 Model 7044-170 (containing a 400 MHz POWER3-II with 768 MB of RAM) running AIX 4.3.3.

4.2 Overview of the Programs

Table 1 shows 16 Java programs used to evaluate our devirtualization techniques. The programs cover a wide spectrum of programming styles and application categories such as computational benchmarks, transaction processing, a parser, browsers, graphical applications, a word processor, and a Web server. Note that the results using SPECjvm98 [34] programs do not follow the official SPEC rules.

4.3 Characteristics of Method Calls

For each program, Table 2 details the characteristics of both static and dynamic methods.

The geometric mean of 73.5% (ranging from 33.4% to 99.5%) of the virtual method calls are monomorphic. The results show generally higher usages of dynamic monomorphic methods from the application classes in programs without GUIs (jess, db, javac, mpegaudio, mtrt, jack, jbb, and XML parser), (compress and Java Server are exceptions.)

Table 1: Descriptions of the programs used in our experiments

Program	Description
compress	LZW compression and decompression in SPECjvm98. Run the benchmark with size = 100.
jess	NASA's CLIP expert system in SPECjvm98. Run the benchmark with size = 100.
db	Search and modify a database in SPECjvm98. Run the benchmark with size = 100.
javac	Source to bytecode compiler in SPECjvm98. Run the benchmark with size = 100.
mpegaudio	Decompress audio file in SPECjvm98. Run the benchmark with size = 100.
mtrt	Multi-threaded image rendering in SPECjvm98. Run the benchmark with size = 100.
jack	Parser generator generating itself in SPECjvm98. Run the benchmark with size = 100.
jbb [35]	SPECjbb2000 is a transaction processing benchmark. Run the benchmark with the number of warehouses = 1.
XML parser [36]	IBM's XML parser. XML4J version 3.0.1. Run a sample program to parse an XML file.
Java Server [37]	Java Server Web Development Kit 1.0.1. Run the Web server and access it while running some servlets.
swing	GUI components version 1.1.1 written in pure Java. Run a demo application including many components.
Java2D	2D graphics library. Run a demo application including many components.
jfig [38]	A Java version of the xfig drawing program. Version 1.38b. Run the application and open a document.
ICE Browser [39]	Simple Internet browser version 5.01. Run the application and open a Web page.
HotJava [40]	HotJava browser version 1.1.5. Run the application and open a Web page.
Ichitaro Ark [41]	Word processor written in pure Java. Run the application and open a document.

Table 2: Characteristics of static and dynamic method calls

Program	Static Call	Virtual Call	Monomorphic Virtual Call %		Interface Call	Monomorphic Interface Call %	
			Lib.	App.		Lib.	App.
compress	225,975,805	12,039	49.6%	25.0%	446	41.3%	58.7%
jess	78,375,454	36,872,088	0.2%	83.8%	706,505	0.0%	0.7%
db	52,992,991	52,529,114	0.1%	97.1%	14,931,539	0.0%	100.0%
javac	57,019,624	48,408,808	5.1%	62.2%	3,379,096	0.0%	99.8%
mpegaudio	99,702,499	9,853,620	0.2%	33.2%	182,220	0.1%	99.9%
mtrt	17,406,471	269,740,419	0.3%	90.7%	402	46.3%	53.7%
jack	24,400,198	25,219,092	20.3%	59.5%	4,155,315	0.0%	55.0%
jbb	132,586,167	173,403,868	15.9%	80.6%	4,036,513	0.3%	99.6%
XML parser	1,812,996	516,133	2.1%	97.4%	1,217,916	0.1%	99.9%
Java Server	337,899	74,901	67.9%	11.9%	3,118	65.7%	28.8%
swing	3,143,213	1,754,935	57.4%	0.3%	177,638	49.8%	0.1%
Java2D	17,956,992	6,490,662	72.6%	4.1%	1,446,333	49.3%	0.1%
jfig	1,274,203	296,283	67.4%	0.0%	33,006	51.0%	0.5%
ICE Browser	1,732,313	261,235	62.1%	10.3%	47,519	67.8%	10.2%
HotJava	1,882,711	504,321	78.8%	0.0%	55,523	64.2%	0.3%
Ichitaro Ark	4,960,087	2,421,789	23.7%	32.2%	806,600	16.4%	16.4%
geom. mean			73.5%			55.3%	

- Static Call: The total number of static calls.
- Virtual Call: The total number of virtual method calls.
- Monomorphic Virtual Call: The percentage of virtual method calls that are performed at monomorphic call sites.
- Interface Call: The total number of interface method calls.
- Monomorphic Interface Call: The percentage of interface method calls that are performed at monomorphic call sites.
- Lib.: The percentage within Java class libraries.
- App.: The percentage within the application.

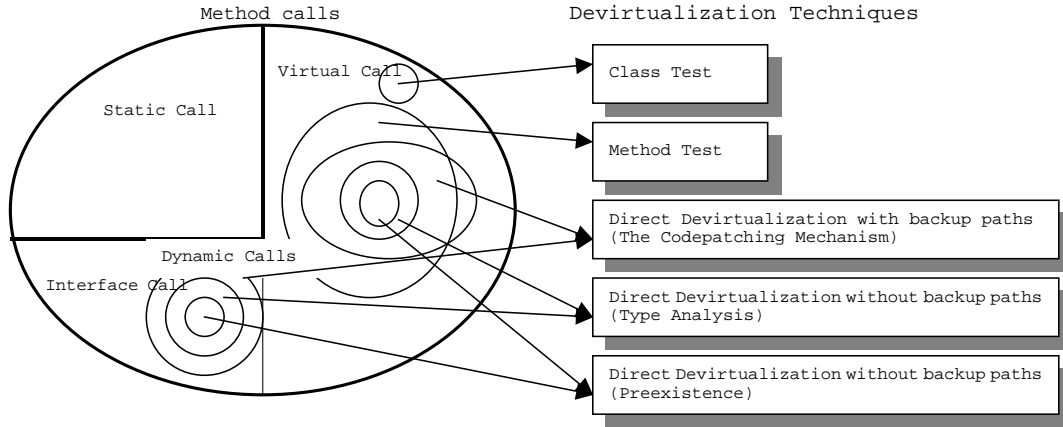


Figure 1: Venn diagram of applicable categories of devirtualization techniques

The dynamic method calls tend to be monomorphic within the Java class libraries in programs with GUIs (*swing*, *Java2D*, *jfig*, *ICE Browser*, *HotJava*, and *Ichitaro Ark*). The results also show dynamic method calls are surprisingly monomorphic in all of the programs except *mpegaudio*. This shows that we have many opportunities to perform devirtualization. On the other hand, the program *compress* is not expected to be much affected by devirtualization techniques, since the number of virtual calls is extremely small. Note that *jbb*, unlike the other programs, is a benchmark program to measure the throughput in a constant time. There are some differences in execution counts among optimizations within the same program with GUI since we perform the scenario manually.

4.4 Results of Devirtualization

In this section, we show the results by applying each of four optimizations cumulatively. First, we start with guarded devirtualization. Second, we add direct devirtualization with the code patching mechanism. Next, we add type analysis, and finally we include preexistence analysis.

Figure 1 is a Venn diagram to help clarify some of the relationships among the devirtualization techniques we applied. For example, in the lower left quadrant, some of the interface calls for dynamic methods can use direct devirtualization with backup paths, and a subset of those calls may also be candidates for direct devirtualization without backup paths.

4.4.1 Guarded Devirtualization

We started by performing guarded devirtualization with class and method tests together. Table 3 shows the characteristics of programs with guarded devirtualization. We apply class test only to method calls within the class itself, as described in Section 3.4. We omit the characteristics of class test in Table 3, 4, 5, 6, and, 7 because they are not executed so much. We apply method tests to virtual method calls that have a single or multiple targets at compilation time only for code that can be inlined. If a method call has multiple targets, only a method defined in a leaf class is inlined. We do not apply method test to method calls that are merely replaced with direct method calls.

We adopted method test for guarded devirtualization, even though the runtime cost of method test is slightly higher than that of class test as presented in Section 3.3. In general, this is because the method test can allow the inlined version of the code to be exe-

cuted more often than the class test can. For example, when we attempted to apply only the class test to *mtrt*, the success ratio was decreased from 100% to 70%. On the other hand, the runtime overhead can be hidden by using compiler optimizations.

In summary, as in Table 3, the success ratio for the execution of inlined version varies from 50.7% to 100% (the geometric mean of 91.7%).

4.4.2 Direct Devirtualization with the Code Patching Mechanism

For the next tests, we added direct devirtualization with the code patching mechanism. This is applied to virtual method calls that have only a single target at compile time and to interface method calls that are implemented by a single class. We apply this technique not only to dynamic method calls that can be inlined, but also to dynamic method calls that can be replaced with direct method calls.

Table 4 shows the characteristics of these directly devirtualized programs. Here, the execution frequencies of the inline code at directly devirtualized call sites vary from 88.8% to 100% (the geometric mean of 98.1%).

4.4.3 Type Analysis

Next, we added in flow-sensitive type analysis. Table 5 shows the characteristics of programs using flow-sensitive type analysis. As is shown in Table 5, the percentages of actually executed inline code at directly devirtualized call sites vary from 84.1% to 100% (the geometric mean of 97.3%).

4.4.4 Preexistence Analysis

Finally, we also performed preexistence analysis. Table 6 shows the characteristics of programs including preexistence analysis. As is shown in Table 6, the percentages of the inline code actually executed at the directly devirtualized call sites vary from 81.5% to 100% (the geometric mean of 96.9%).

Table 3: Characteristics of programs with guarded devirtualization described in Section 3.4

Program	Remaining Virtual Call		Remaining Interface Call	Method Test	
	Counts	Reduction % (from no devirtualization)	Counts	Counts	inlined execs
compress	9,967	17.2%	446	2040	97.2%
jess	12,212,470	66.9%	706,505	24,660,863	100.0%
db	46,680,205	11.1%	14,931,539	5,833,525	100.0%
javac	41,274,813	14.7%	3,381,204	7,973,933	90.9%
mpegaudio	6,821,177	30.8%	182,220	3,037,975	99.8%
mtrt	75,811,042	71.9%	402	193,929,371	100.0%
jack	17,683,893	29.9%	4,155,315	7,529,626	100.0%
jbb	30,279,681	N/A	3,973,859	140,157,002	100.0%
XML parser	80,328	84.4%	1,217,916	435,657	100.0%
Java Server	40,952	45.3%	2,960	19,961	99.1%
swing	1,249,252	28.8%	176,796	498,142	86.9%
Java2D	4,813,273	25.8%	1,453,963	2,153,279	50.7%
jfig	205,157	30.3%	33,616	96,697	86.9%
ICE Browser	162,833	37.7%	45,118	93,499	92.0%
HotJava	346,182	31.4%	56,558	108,735	90.2%
Ichitaro Ark	1,558,901	35.6%	594,341	663,773	90.2%

Remaining Virtual Call: The non-devirtualized virtual method calls after performing optimizations.

Remaining Interface Call: The non-devirtualized interface method calls after performing optimizations.

Counts: The total count of each kind. For example, in compress, the inlined version of the code was executed for 97.2% of all the method tests.

Method Test: The guarded test described in Section 3.4.

Reduction: The percentage difference .with respect to the specified case.

Inlined execs: The percentage of the inlined code actually executed.

Table 4: Characteristics of programs with guarded devirtualization plus direct devirtualization with the code patching mechanism described in Section 3.1

Program	Remaining Virtual Call		Remaining Interface Call		Method Test		Codepatch		Invalidation sites
	Counts	Reduction % (from no devirtualization)	Counts	Reduction % (from no devirtualization)	Counts	inlined execs	Counts	inlined execs	
compress	9,796	18.6%	443	0.7%	657	91.2%	1,596	97.7%	18
jess	10,790,816	66.9%	701,785	0.0%	10,798	82.9%	26,083,410	100.0%	22
db	46,557,413	11.1%	14,931,536	0.0%	5,082	99.2%	5,951,271	100.0%	18
javac	29,161,105	39.8%	3,379,389	-0.1%	2,157,383	66.0%	18,349,878	99.0%	32
mpegaudio	6,804,623	30.8%	395	99.8%	31,908	82.4%	3,204,602	100.0%	18
mtrt	7,244,261	97.3%	399	0.7%	1,667	95.0%	262,494,525	100.0%	18
jack	16,317,679	35.3%	2,624,376	36.8%	33,849	99.6%	10,925,027	99.4%	22
jbb	18,644,624	N/A	3,771,951	N/A	468,892	100.0%	144,852,219	100.0%	17
XML parser	80,173	84.5%	1,146,448	5.9%	114,407	100.0%	362,940	100.0%	8
Java Server	40,292	46.2%	2,753	11.7%	2,215	92.3%	18,943	99.0%	44
swing	1,251,871	28.7%	164,057	7.6%	222,752	63.3%	373,117	94.2%	199
Java2D	5,038,999	22.4%	1,422,935	1.6%	1504,829	20.0%	1,130,901	88.8%	77
jfig	167,970	43.3%	28,230	14.5%	28,055	57.6%	72,518	99.1%	43
ICE Browser	144,733	44.6%	37,185	15.6%	11,245	47.6%	96,555	99.3%	79
HotJava	316,788	37.2%	47,202	15.0%	19,416	52.5%	111,078	96.5%	158
Ichitaro Ark	1,446,609	40.3%	575,172	28.7%	120,782	38.6%	641,550	97.7%	215

Codepatch: The direct devirtualization with the code patch mechanism described in Section 3.1.

Reduction: The percentage difference .with respect to the specified case.

Inlined execs: The percentage of the inlined code actually executed.

Invalidation sites: The number of call sites where the code patching is performed when a class is loaded and a method is overridden during the execution of a program.

Table 5: Characteristics of programs with guarded and direct devirtualizations plus flow-sensitive type analysis described in Section 3.2

Program	Remaining Virtual Call		Remaining Interface Call	Method Test		Codepatch			Invalidation sites
	Counts	Reduction % (from no devirtualization)		Counts	inlined execs	Counts	Reduction % (from direct devirtualization)	inlined execs	
compress	9,585	20.4%	443	657	91.2%	1,282	20.1%	97.2%	16
jess	7,895,376	78.6%	701,785	10,798	82.9%	24,978,943	4.2%	100.0%	20
db	46,557,233	11.4%	14,931,536	5,082	99.2%	5,950,246	0.0%	100.0%	16
javac	27,540,151	43.1%	3,381,201	2,157,387	66.0%	19,816,214	-1.0%	92.6%	54
mpegaudio	6,804,397	30.9%	395	31,908	82.4%	3,204,247	0.0%	100.0%	16
mtrt	7,244,059	97.3%	399	1,667	95.0%	245,247,103	6.6%	100.0%	16
jack	12,322,620	51.1%	2,624,376	33,849	99.6%	13,443,883	-23.2%	99.5%	20
jbb	19,458,683	N/A	3,950,837	491,829	100.0%	150,415,652	N/A	100.0%	16
XML parser	79,782	84.5%	1,146,448	144,407	100.0%	361,906	0.3%	100.0%	8
Java Server	38,941	48.0%	2,753	2,215	92.3%	16,624	12.4%	98.9%	35
swing	1,209,389	31.1%	163,713	212,813	66.6%	362,306	3.1%	94.0%	210
Java2D	4,802,108	26.0%	1,392,895	1435,134	20.8%	1,110,593	7.0%	84.1%	91
jfig	181,572	38.7%	29,796	28,117	57.3%	68,222	6.1%	99.0%	47
ICE Browser	155,810	40.4%	46,080	15,774	45.0%	92,482	11.8%	98.9%	68
HotJava	333,557	33.9%	47,946	20,206	53.3%	102,599	7.9%	96.2%	157
Ichitaro Ark	1,374,676	43.2%	542,976	102,232	39.2%	576,341	10.4%	97.5%	217

Codepatch: The direct devirtualization with the code patch mechanism described in Section 3.1.
Reduction: The percentage difference .with respect to the specified case.
Inlined execs: The percentage of the inlined code actually executed.
Invalidation sites: The number of call sites where the code patching is performed when a class is loaded and a method is overridden during the execution of a program.

Table 6: Characteristics of programs with guarded and direct devirtualizations and flow-sensitive type analysis plus preexistence analysis described in Section 3.3

Program	Remaining Virtual Call	Remaining Interface Call	Method Test		Codepatch			Invalidation sites	Methods that must be recompiled
			Counts	inlined execs	Counts	Reductions% (from direct devirtualization)	inlined execs		
compress	9,585	443	669	89.5%	1,059	33.7%	97.7%	10	6
jess	7,895,376	701,785	10,822	82.7%	18,261,070	30.0%	100.0%	14	6
db	46,557,233	14,931,536	5,085	99.2%	5,950,061	0.0%	100.0%	10	6
javac	27,704,969	3,379,221	2,461,630	57.9%	18,199,383	8.1%	91.8%	50	10
mpegaudio	6,804,397	395	31,959	82.2%	2,173,264	32.2%	100.0%	10	6
mtrt	7,244,059	399	1,678	94.3%	191,710,141	27.0%	100.0%	10	6
jack	12,322,620	2,624,376	33,887	99.5%	9,314,605	14.8%	99.2%	14	6
jbb	20,658,308	4,213,999	524,173	100.0%	128,558,128	N/A	100.0%	14	2
XML parser	79,782	1,146,448	144,407	100.0%	360,837	0.6%	100.0%	6	2
Java Server	38,949	2,753	2,215	92.3%	12,500	34.3%	98.6%	35	0
swing	1,266,590	163,528	221,685	64.5%	279,771	25.5%	93.6%	180	26
Java2D	4,783,173	1,418,776	1,410,625	21.8%	919,675	25.4%	81.5%	75	16
jfig	165,701	28,043	21,452	50.0%	44,622	38.7%	98.8%	28	13
ICE Browser	136,834	36,675	10,212	46.4%	67,071	30.7%	99.2%	67	2
HotJava	321,514	47,553	16,913	50.0%	72,835	35.8%	94.5%	144	16
Ichitaro Ark	1,451,855	577,285	121,842	39.9%	459,639	28.7%	97.3%	196	22

Codepatch: The direct devirtualization with the code patch mechanism described in Section 3.1.
Reduction: The percentage difference .with respect to the specified case.
Inlined execs: The percentage of the inlined code actually executed.
Invalidation sites: The number of call sites where the code patching is performed when a method is overridden during the execution of a program.
Method that must be recompiled: The number of method recompilation candidates when a class is loaded during the execution of a program and the method is overridden.

4.5 Evaluation and Breakdown of the Results

In this section, we discuss a number of observations that can be made from the above results.

Figure 2 summarizes the breakdown of call sites optimized by each devirtualization technique on some programs when we applied all the devirtualization techniques (corresponding to Section 4.4.4). We use “(o)” to denote all optimizations are performed. All values are given in relative execution counts against the non-devirtualized version (corresponding to Section 4.3). Table 7 also shows the effectiveness of devirtualization techniques (in execution counts) for all the programs excluding *jbb*. The reason is that optimizations increase the number of executed instructions and we cannot show the reductions since this benchmark measures throughput in a constant time, as we pointed out in Section 4.3.

The results from Table 2 show a trend that dynamic method calls in programs with GUI (such as AWT and Swing) tend to be monomorphic within the common class libraries that Java provides. The programs use extensible and reusable common class libraries, but they use them monomorphically. This usage pattern based on the experiments with real Java programs is very encouraging. It increases the opportunity for devirtualization.

As is shown in Table 7, we have measured the reduction of dynamic method calls ranging from 8.9% to 97.3% (the average of 40.2%). The program where we measured the highest reduction in virtual method calls is *mtrt*. *Mtrt* has a kernel loop in the method `Intersect()` in the class `spec.benchmarks._205_raytrace.Octnode`. It calls some small methods such as the methods `GetX()`, `GetY()`, and `GetZ()` in the class `spec.benchmarks._205_raytrace.Point` to get instance variables very frequently. Direct devirtualization with the code patching mechanism can inline almost all virtual method calls. Furthermore, 24.3% of them can be directly devirtualized without any backup paths.

In Table 7, the reduction of virtual method calls is also relatively high in *jess*. Here, 60% of the method tests are converted to direct devirtualization. These call sites are in the method `CallNode()` in the class `spec.benchmarks._202_jess.jess.Node2`. The devirtualization with method inlining can expand the analysis scope of target methods, and this helps to prove the type of receiver objects by type analysis. Therefore, type analysis can remove the method calls to `equals()` in the class `java.lang.Object` in the most frequently-called method `equals()` in the class `spec.benchmarks._202_jess.jess.Value`. The method call with the static type `java.lang.Object` has a small runtime overhead, since the receiver may have an array object and the method call has to check whether the object type is an array. Therefore, type analysis is an effective optimization.

As can be seen from Table 4 and Table 5, type analysis is effective in reducing the number of virtual method calls (see the column of **Remaining Virtual Call**). In the program *jess*, it reduces the number of virtual method calls by 43.9%, which are to call `hashCode()` in the class `java.lang.Object` and `java.lang.Integer` with a small runtime overhead, as we described in Section 3.2. For other programs, it also reduces the number of virtual method calls with a small runtime overhead in *javac* by 36.3%, in *mpegaudio* by 11.6%, and in *jack* by 88.4%. These method calls are part of the column of **Remaining Virtual Call**.

As can be seen from Table 4, Table 5, and Table 6, the average of **Codepatch inlined execs** decreases from 98.1% to 96.9% with type analysis and preexistence analysis. This shows that direct devirtualization without backup paths are actually executed. Table 7 also shows the reduction by the average of 24.3% for **Codepatch inlined execs** with type analysis and preexistence analysis. We cannot measure execution counts of directly devirtualized sites without backup paths since a highly optimizing compiler moves or removes individual instructions of devirtualized call sites freely. The results also show that direct devirtualization by type analysis and preexistence applies to 24.3% of the direct devirtualizations with a backup path generated by the code patching mechanism.

As can be seen from Table 4 and Table 5, when the compiler performs type analysis, the number of **Invalidation sites** increases in *javac*, *swing*, *Java2D*, *jfig*, and *Ichitaro Ark*. If type analysis proves that an instance of an array class does not reach a receiver of a dynamic method call `equals()` in the class `java.lang.Object`, the method call can be directly devirtualized using the code patching mechanism. The call site will be invalidated to execute the dynamic method call when the method is overridden by class loading. As a result, the number of **Invalidation sites** increases rather than decreases in comparison to the case where no type analysis is used. The increase in the column of **Codepatch inlined execs** for type analysis also shows this.

In Table 7, the programs where we measured the smallest reduction made by type analysis and preexistence in **Codepatch inlined execs** are *db* and *XML parser*. At a few dominant call sites in *db* and *XML parser*, the forms of virtual method invocations are `this.f.m()` or `arg.f.m()`, where *this* is an expression of the current instance, *arg* is an expression of an argument, and *f* is a field of that class. In *db*, since the program assigns only to a non-private field *f* in constructors once, immutable field analysis [11] can reduce the number of **Codepatch inlined execs**.

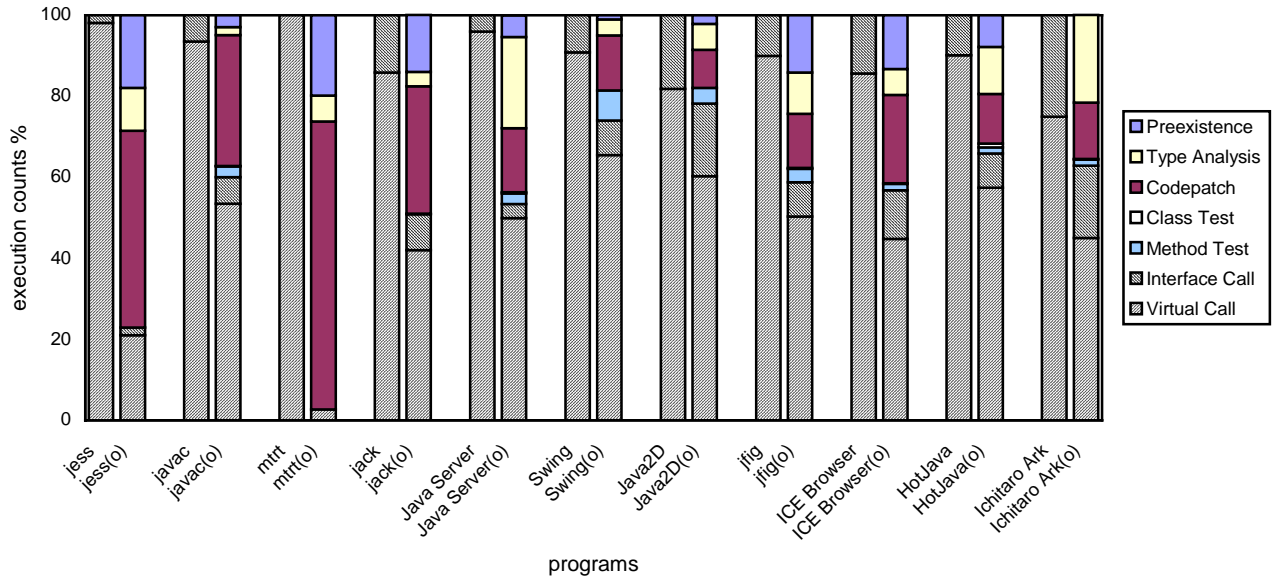


Figure 2: Breakdown of call sites optimized by each devirtualization technique (in execution counts)

Table 7: Effectiveness of devirtualization techniques (in execution counts)

Program	Reduction % from no devirtualization to preexistence			Reduction % from guarded devirtualization to preexistence	Reduction % from codepatch to preexistence
	Virtual Call	Interface Call	Both	Method Test inlined execs	Codepatch inlined execs
compress	20.4%	0.7%	19.7%	70.6%	33.7%
jess	78.6%	0.7%	77.1%	100.0%	30.0%
db	11.4%	0.0%	8.9%	99.9%	0.0%
javac	42.8%	0.0%	40.0%	82.1%	8.1%
mpegaudio	30.9%	99.8%	32.2%	99.1%	32.2%
mrt	97.3%	0.7%	97.3%	100.0%	27.0%
jack	51.1%	36.8%	49.1%	99.6%	14.8%
XML parser	84.5%	5.9%	29.3%	66.9%	0.6%
Java Server	48.0%	11.7%	46.5%	89.8%	34.3%
swing	27.8%	7.9%	26.0%	71.3%	25.5%
Java2D	26.3%	1.9%	21.9%	85.7%	25.4%
jfig	44.1%	15.0%	41.2%	88.9%	38.7%
ICE Browser	47.6%	16.8%	43.2%	94.7%	30.7%
HotJava	36.2%	14.4%	34.1%	92.2%	35.8%
Ichitaro Ark	40.1%	28.4%	37.1%	92.7%	28.7%
average	45.8%	16.0%	40.2%	88.9%	24.3%

```

public class Vector {
    protected Object elementData[];
    protected int elementCount;

    public Enumeration elements() {           // in JDK 1.1, this method is declared as final
        return new Enumeration() {
            int count = 0;
            public boolean hasMoreElements() { return count < elementCount; }
            public Object nextElement() {
                synchronized (Vector.this) {
                    if (count < elementCount) return elementData[count++];
                }
                throw new NoSuchElementException("Vector Enumeration");
            }
        }
    }
}

class Sample {
    Vector v;
    Object o[];
    void foo() {
        int i = 0;
        Enumeration e = v.elements();
        while (e.hasMoreElements())         // interface method call
            o[i++] = e.nextElement();       // interface method call
    }
}

```

Example 7: A sample usage of interface method calls

We were also surprised that the number of interface method calls is almost unchanged in db. We have investigated the reason by looking into statistics. The number of interface method calls is dominated by call sites in the method `set_index()` in the class `spec.benchmarks._209_db.Database` and the method `equals()` in the class `spec.benchmarks._209_db.Entry`. At these call sites, the interface method calls are used as shown in Example 7. In JDK 1.1, the method `elements()` in the class `java.lang.Vector` is declared as `final`. In Java 2, however, the method is not declared as `final`. This change causes the type information to be lost for a receiver expression `e` in the method `foo()`.

If the method is declared as `final` (in the case of JDK 1.1), the method can be directly inlined and the return type is known as an inner class. Therefore, type analysis can prove that only the inner class that is never overridden comes to the receiver expression `e` of the interface method call. Based on the results, we can translate interface method calls into virtual method calls, direct method calls, or inlined codes. In that case, we could get a huge reduction of 99% for the interface method calls in db.

On the other hand, if it is not declared as `final` (in the case of Java 2), type analysis returns the `Enumeration` class as an ambiguous type and the compiler determines the call site is polymorphic. Furthermore, the `Enumeration` class is always implemented by a few classes. Thus, no devirtualization technique can be applied. In this case, specialization and customization would not be effective since the receiver expression does not depend on its arguments. Message splitting [31] could help in this situation. However, message splitting will increase the code size by duplicating a complete loop structure. In summary, eliminating `final` from the method declaration caused a large performance loss.

4.6 Performance Results

We measured the execution time of eight non-interactive programs (`compress`, `jess`, `db`, `javac`, `mpegaudio`, `mrtt`, `jack`, and `jbb`). The other programs were difficult to measure because of

their interactive nature and dependencies within AWT. Figure 3 shows the performance improvement resulting from the cumulative optimizations. Here, all the measurements are performed by compiling all methods. All the values are given in relative speed up against non-devirtualized versions (only with base optimizations). Each of the bars shows the cumulative effect including prior optimizations. For each of the bars, the following combinations of techniques are used:

- Base optimizations (not shown in the figure): All optimizations except the devirtualization techniques that we described in Section 4.1 are performed (corresponding to Section 4.3) and static method inlining are performed.
- +Method Test, Class Test: Base optimizations and guarded devirtualization (i.e. class and method tests) are performed (corresponding to Section 4.4.1).
- +Codepatch: Base optimizations, guarded devirtualization, and direct devirtualization with the code patching mechanism are performed (corresponding to Section 4.4.2).
- +Type Analysis: Base optimizations, guarded devirtualization, direct devirtualization with the code patching mechanism, and flow-sensitive type analysis are performed (corresponding to Section 4.4.3).
- +Preexistence: Base optimizations, guarded devirtualization, direct devirtualization with code patching mechanism, flow-sensitive type analysis, and preexistence analysis are performed (corresponding to Section 4.4.4).

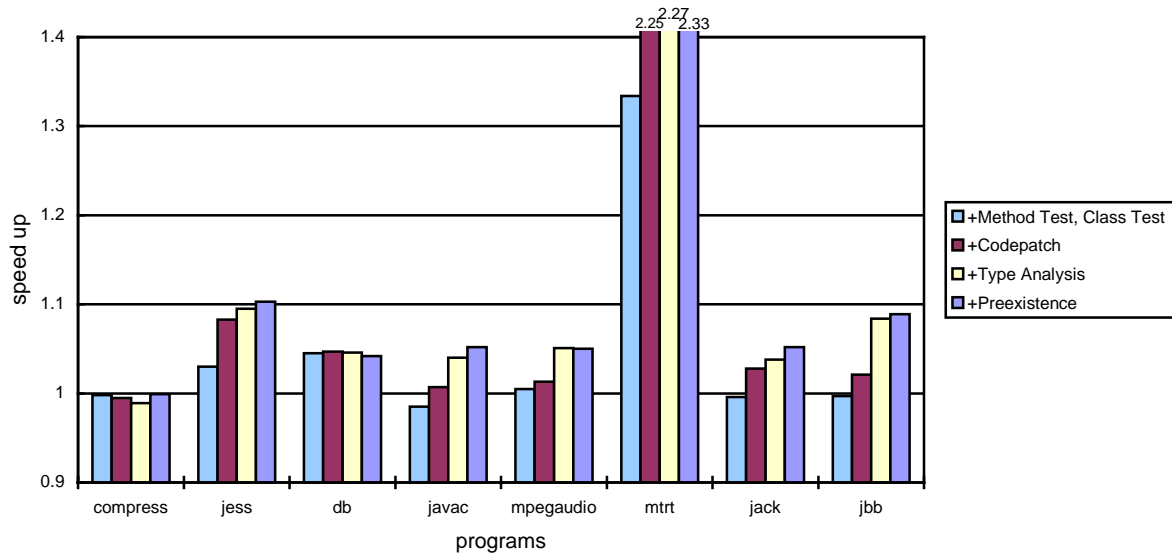


Figure 3: Performance improvement of the non-interactive benchmarks from the non-devirtualized version

We have measured the speedup of 4% in the geometric mean by guarded devirtualization with class tests and method tests. Direct devirtualization with the code patching mechanism improves the performance by 13% in the geometric mean. It especially improves the performance of `mtrt`. This is because the program calls some small methods in a kernel loop very frequently, and almost all of these method calls can be directly devirtualized by the code patching mechanism, as we described in Section 4.5. To show the performance impact of the existence of backup paths, we have made a small experiment to execute `mtrt` by eliminating of all backup paths. Even in the extreme case, the performance of this version is only 6% faster. The result shows that the overhead of the existence of backup paths is usually smaller than we thought. Direct devirtualization with the code patching mechanism also improves the performance of `jess`. On this program, almost all the method tests in the kernel are converted to direct devirtualizations as we described in Section 4.5.

Type analysis improves the performance of `jess`, `javac`, `mpegaudio`, and `jack`. The reason is that these programs include parsers and expert systems, which manipulate many string objects using the methods `hashCode()`, `equals()`, and `toString()`. The reduction of these method calls by type analysis is high as we described in Section 4.5. Type analysis also improves the performance of `jbb`. It is effective with the classes `spec.jbb.JBBmain` and `spec.jbb.JBButil`.

Using all of the optimizations presented in this paper, we have measured a speedup of 16% in the geometric mean.

5. Conclusions

We have shown that the direct devirtualization with the code patch mechanism we proposed in this paper can remove almost all class and method tests generated by guarded devirtualization, and that it can be applied to a wide range of dynamic method calls. The runtime overhead of our approach is smaller than that of a recompilation-based approach. We eliminated the backup path by type analysis and preexistence analysis. We also optimized the inlined code by inserting compensation code in the backup path if

it is not eliminated. We evaluated the devirtualization techniques implemented in our JIT compiler based on various statistics collected by running a set of real programs in various application categories. We have observed the reduction of dynamic method calls ranging from 8.9% to 97.3% (the average of 40.2%) by using these devirtualization techniques. Furthermore, we have shown that type analysis and preexistence analysis eliminated the backup path for 24.3% of the directly devirtualized sites that used the backup path. Overall, we have reported performance improvements ranging from -1% to 133% (with the geometric mean of 16%). We have also pointed out a few problems such as non-sealed class library and missing type information, which caused performance degradation in a Java runtime environment.

Acknowledgement

We are grateful to the people in Network Computing Platform at Tokyo Research Laboratory for implementing our JIT compiler. We thank Takeshi Ogasawara and Toshio Suganuma for information on implementing the code patching mechanism on the IA32 architecture. We also thank Shannon Jacobs for his editorial assistance. We appreciate the insightful comments from the anonymous reviewers and the committee members of OOPSLA.

References

- [1] Brad Calder and Dirk Grunwald. Reducing Indirect Function Call Overhead In C++ Programs, In *Proceedings of the ACM SIGPLAN '94 Symposium on Principles of Programming Languages*, pp. 397-408, 1994
- [2] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-Guided Receiver Class Prediction, In *Proceedings of the Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA '95*, pp. 108-123, 1995.
- [3] Gerald Aigner, and Urs Hölzle. Eliminating Virtual Function Calls in C++ Programs, In *Proceedings of the 10th European Conference on Object-Oriented Programming – ECOOP '96*,

- volume 1098 of Lecture Notes in Computer Science, Springer-Verlag, pp. 142-166, 1996.
- [4] Urs Hölzle. Adaptive Optimization For SELF: Reconciling High Performance With Exploratory Programming, PhD thesis, Stanford University, 1994
 - [5] Jeffery Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy, In *Proceedings of the 9th European Conference on Object-Oriented Programming – ECOOP '95*, volume 952 of Lecture Notes in Computer Science, Springer-Verlag, pp. 77-101, 1995.
 - [6] Mary F. Fernandez. Simple and Effective Link-Time Optimization of Modula-3 Programs, In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pp. 103-115, 1995.
 - [7] David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls, In *Proceedings of the Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA '96*, pp. 324-341, 1996.
 - [8] Frank Tip and Jens Palsberg. Scalable Propagation-Based Call Graph Construction Algorithm, In *Proceedings of the Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA 2000*, 2000.
 - [9] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Garnon, and Charles Godin. Practical Virtual Method Call Resolution for Java, In *Proceedings of the Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA 2000*, 2000.
 - [10] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler, In *ACM 1999 Java Grande Conference*, pp.119-128, 1999.
 - [11] David Detlefs and Ole Agesen. Inlining of Virtual Methods, In *Proceedings of the 13th European Conference on Object-Oriented Programming – ECOOP '99*, volume 1628 of Lecture Notes in Computer Science, Springer-Verlag, pp. 258-278, 1999.
 - [12] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*, Addison-Wesley, 1996.
 - [13] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization, In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 32-43, 1992.
 - [14] Jens Palsberg and Michael I. Schwartzbach. Object-Oriented Type Inference, In *Proceedings of the Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA '91*, pp. 146-161, 1991.
 - [15] Ole Agesen and Urs Hölzle. Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object-Oriented Languages, In *Proceedings of the Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA '95*, pp. 91-107, 1995.
 - [16] Paul R. Carini, Hirini Srinivasan, and Michael Hind. Flow-Sensitive Type Analysis for C++, *IBM Research Report*, RC 20267, 1995
 - [17] Etienne M. Gagnon, Laurie J. Hendren, and Guillaume Marcéau. Efficient Inference of Static Types for Java Bytecode, Static Analysis Symposium 2000, 2000
 - [18] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches, In *Proceedings of the 5th European Conference on Object-Oriented Programming – ECOOP '91*, volume 512 of Lecture Notes in Computer Science, Springer-Verlag, pp. 21-38, 1991.
 - [19] David F. Bacon. Fast and Effective Optimization of Statically Typed Object-Oriented Languages, Ph.D. thesis, University of California at Berkeley, 1997.
 - [20] Junpyo Lee, Byung-Sun Yang, Suhyun Kim, SeungIl Lee, Yoo C. Chung, Heungbok Lee, Je Hyung Lee, Soo-Mook Moon, Kemal Ebcioglu, Erik Altman. Reducing Virtual Call Overheads in a Java VM Just-In-Time Compiler, *The 4th Annual Workshop on Interaction between Compilers and Computer Architectures*, pp.21-33, 2000
 - [21] Sun Corp. The Java HotSpot Performance Engine Architecture, Available at <http://java.sun.com/products/hotspot/whitepaper.html>.
 - [22] Bowen Alpern, Mark Charney, Jong-Deok Choi, Anthony Cocchi, and Derek Lieber. Dynamic Linking on a Shared-Memory Multiprocessor, *The 1999 International Conference on Parallel Architecture and Compilation Techniques*, 1999.
 - [23] Intel Corp. Intel Architecture Software Developer's Manual, order number 243192, 1997.
 - [24] Michal Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing JUDO: Java™ Under Dynamic Optimizations, In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pp. 13-26, 2000.
 - [25] Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. Effective Null Pointer Check Elimination Utilizing Hardware Trap, To appear in *the International Conference on Architectural Support for Programming Language and Operating Systems*, 2000.
 - [26] Jens Knoop, Ruthing Oliver, and Steffen Bernhard. Lazy Code Motion, In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pp. 224-234, 1992.
 - [27] Jong-Deok Choi, Manish Gupta, Muaricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape Analysis for Java, In *Proceedings of the Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA '99*, pp. 1-19, 1999.
 - [28] John Whaley and Martin Rinard. Compositional Pointer and Escape Analysis for Java Programs, In *Proceedings of the Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA '99*, pp. 187-206, 1999.
 - [29] IBM Corp. Jikes, available at <http://oss.software.ibm.com/developerworks/opensource/jikes/project/index.html>.

- [30] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*, Addison-Wesley, 1996.
- [31] Craig Chambers and David Unger. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object Oriented Programs, In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 150-164, 1990
- [32] Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. Overview of the IBM Java Just-in-Time Compiler, *IBM Systems Journal*, Vol. 39, No. 1, pp.175-193, 2000
- [33] Frederick Chow. Minimizing Register Usage Penalty at Procedure Calls, In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pp. 85-94, 1988.
- [34] Standard Performance Evaluation Corp. SPEC JVM98 Benchmarks, available at <http://www.spec.org/osg/jvm98/>.
- [35] Standard Performance Evaluation Corp. SPECjbb2000 Benchmarks, available at <http://www.spec.org/osg/jbb2000/>.
- [36] IBM Corp. XML Parser for Java, available at <http://alphaworks.ibm.com/tech/xml4j>.
- [37] Sun Corp. JavaServer™ Web Development Kit (JSWDK) 1.0.1 Reference Implementation, available at <http://java.sun.com/products/jsp/download.html>.
- [38] Norman Hendrich. jfig, available at <http://tech-www.informatik.uni-hamburg.de/applets/javafig/>
- [39] ICESoft. ICE Browser, available at <http://www.icesoft.no/>
- [40] Sun Corp. HotJava™ Browser, available at <http://java.sun.com/products/hotjava/index.html>
- [41] JUSTSYSTEM Corp. ICHITARO ARK for Java, available at <http://www.justsystem.com/ark/index.html>.