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

While dynamic method calls are sources of reusability and extensibility in object-oriented languages, they have long been targets of optimizations because they incur runtime overhead and prevent classical compiler optimizations, resulting in many techniques to *devirtualize* the calls having been proposed. However, these techniques are less effective in Java since either Java is a statically-typed language, or not straightforwardly applicable since Java's capability of loading classes dynamically prevents whole program optimizations. This paper proposes a direct devirtualization technique with a code patching mechanism. This technique has lower overhead than a well-known recompilation approach. We have implemented a number of devirtualization techniques. This paper presents detailed characteristics and effectiveness as applied them to 16 real programs. We show reductions in the number of dynamic method calls ranging from 8.9% to 97.3% (an average of 40.4%). It conducts some interesting observations of program characteristics and potential performance problems. We also report performance improvements from –1% to 122% (an average of 19%).

1. Introduction

Java [1] is a recently designed object-oriented programming language. It is also a popular language suitable for writing programs that can be reused, because it has excellent extensibility and reusability. This extensibility and reusability were achieved by supporting dynamic method calls and dynamic class loading.

Java provides two dynamic method calls, `invokevirtual` and `invokeinterface`, with method lookup to find a target method like typical object-oriented languages. Furthermore, one of the innovative capabilities of the Java language is its provision for loading classes during the execution of a program. These features provide modularity of class libraries and applications, making it convenient for application programmers. However, they incur a performance penalty, because dynamic method calls requires method lookup at runtime, and dynamic class loading prevents the compiler from applying whole program analysis before execution.

To improve the performance of dynamic method calls, many research approaches for devirtualization [2, 3, 4, 5, 6, 7, 8] have been proposed. Most of them involve tests to guard devirtualized code (i.e. inlined code or direct method calls) in order to ensure that it is correct for the dynamic type of the current receiver. We call this approach *guarded devirtualization*. In dynamically-typed object-oriented languages such as Self [4], since the overhead of dynamic method calls is high, guarded devirtualization is extremely effective. On the other hand, in statically-typed object-oriented languages like Java, since the overhead of dynamic method calls is low due to the fact that a dynamic method call is translated into a few loads followed by a indirect jump, guarded devirtualization is less effective.

To devirtualize dynamic method calls without a guard test, techniques for whole program analysis [5, 6] as used in C++ and Modula-3 have been proposed. We call this approach *direct devirtualization*. They assume that new classes and methods will never be loaded during the execution of a program. Therefore, these techniques cannot be applied directly to Java. If dynamic recompilation [9] is used, direct devirtualization is possible. For its implementation, however, a complicated mechanism called *on-stack re-*

placement is required. It also introduces inefficiency of generated code such as an expansion of a stack frame and redundant store instructions to a memory in order to recompile and restart a method.

In this paper, we evaluate the effectiveness of several devirtualization techniques to maximize the efficiency of compiler optimizations. They are direct devirtualization techniques, such as a code patching mechanism [7] and preexistence analysis [8] using dynamic class hierarchy analysis, and type analysis [10, 11, 12]. The rest of them are indirect devirtualization techniques, such as class test [2, 3] and method test [8]. Especially, direct devirtualization with a code patching is a new approach to reduce the overhead of dynamic method calls, because we have adapted direct devirtualization to allow dynamic class loading with low overhead. Instead of recompilation of a whole method, it leaves an original dynamic method invocation, in the code, which will be executed when a code is patched. We call this method invocation a *backup path*. It introduces slightly optimization constraints that we will discuss in Section 3.1. Furthermore, preexistence analysis and type analysis increase the opportunities for compiler optimizations by direct devirtualization without backup paths. We have implemented a number of devirtualization techniques in our Java JIT compiler. We also present the engineering issues in implementing these techniques. We have measured the effectiveness of these techniques and the performance on a set of sixteen real programs using our JIT compiler.

We found a reduction in the number of dynamic method calls ranging from 8.9% to 97.3% (for an average of 40.4%). We show that the direct devirtualization technique that we propose in this paper can remove almost all test code generated by guarded devirtualization techniques, and can be applied to a wide category of dynamic method calls. We also show that type analysis and preexistence analysis can directly devirtualize with an average of 25.7% of call sites without backup paths that prevents compiler optimizations Furthermore, we also investigated the behavior of a program for which devirtualization is not very effective. At last, we report performance improvements ranging from –1% to 122% (an average of 19%). We also discuss some problems regarding performance degradation.

1.1 Contributions

This paper makes the following contributions:

- Direct devirtualization with a code patching mechanism: This paper presents a direct devirtualization technique with a code patching mechanism. The implementation of our mechanism is simpler and incurs smaller overhead than that of a recompilation approach with on-stack replacement.
- Evaluation of the characteristics and efficiency of devirtualization techniques: This paper presents detailed statistics on a set of real programs. We measure the effectiveness of some devirtualization techniques. We also describe some interesting observations regarding program characteristics, potential performance problems, and consider performance improvements provided devirtualization techniques.

The rest of the paper is structured as follows. Section 2 discusses related work. Section 3 describes devirtualization techniques that we explored. Section 4 gives experimental results with statistics and performance data on a set of real programs. Section 5 outlines our conclusions.

2. Related Work

Devirtualization techniques are important to improve the performance in object-oriented languages. Therefore, many significant 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 does a dynamic method call and stores the class of the current receiver to the call site cache. Holzle extended the technique to a polymorphic case of inline caches [13]. Type prediction [2, 3] and method test [8] 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 they are executed based on

the cache mechanism with memory references. According to the results of simple experiments [14], type prediction without inlining even at 100% accuracy cannot outperform devirtualization of dynamic method calls without inlining. Type prediction with inlining must achieve 90% accuracy to outperform devirtualization without inlining. Finally, no known technique can outperform devirtualization with inlining. Junpyo el at. implement both monomorphic and polymorphic inline caches in a Java Virtual Machine [15]. The experiment results cannot achieve as good a speedup as in Self. In implementations of Java, the cost of dynamic method calls is not so different from that of polymorphic inline caches, type prediction, and method test. As presented earlier, in dynamically-typed object-oriented language, even guarded devirtualization based on a caching approach is extremely effective because the implementation of dynamic method calls is not simple. 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 perform analyses to directly devirtualize dynamic method calls, allowing them to be inlined or implemented by direct method calls. Dean et al. use a class hierarchy analysis to devirtualize dynamic method calls [5]. Class hierarchy analysis is an inexpensive process that 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 [16] proposed more precise static analysis. All such analyses statically devirtualize dynamic method calls. Since Java supports dynamic class loading, these techniques cannot be used in a straightforward manner. Therefore, we have proposed dynamic class hierarchy analysis with a code patching mechanism [7]. Flow-sensitive type analysis [10, 11, 12] 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++, Trells, Dylan, and Java have linguistic a 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 have not been sealed in Java 2.

The Self system supports more aggressive optimizations such as extensive inlining of dynamic method calls [9], whose correctness is ensured by the on-stack replacement mechanism. In Self, there are deoptimization points within each method in which the original state of the method's variables can be recovered from the optimized context maintained by the compiled code. 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 without the violated assumption, and then the recompiled code continues execution. There are several concerns in such a system. In the Self implementation, the compiler produces numerous data structures called scope descriptors to enable deoptimization. Deoptimization points also introduce inefficiency of generated code such as an expansion of a stack frame and store instructions to a memory in order to recompile and restart a method. The compiler cannot also reorder two instructions over a deoptimization point. The Java HotSpot compiler [17] also adopts a recompilation approach with on-stack replacement. We did not explore this approach because of the complexity of its implementation and the overhead of generated code. Preexistence analysis [8] is an approach to prevent on-stack replacement by determining whether direct devirtualization can be performed based on analysis of the receiver expressions. We adopted it to directly devirtualization without a backup path. As a result, it increases the opportunity of compiler optimizations.

Another related work involves specialization. The idea is that methods of a class are cloned based on the type of the receiver objects. Plevyak and Chien [18] proposed whole-program analysis that employs specialization to improve the precision of type analysis. Chambers and Ungar [19] describe the use of customization based on the type of arguments. A variation of specialization is proposed by Dean et al. [20]. Though specialization is an interesting research area, there are a number of concerns such a balance between execution speed and code size. It is difficult to apply in practice, and therefore we did not explore this method.

3. Devirtualization of Dynamic Method Calls

Dynamic method call is an important feature of object-oriented languages because of extensibility and reusability, and it is therefore used frequently. Since a dynamic method call requires method lookup at runtime, its overhead degrades the performance

of the program. As we have described in Section 2, many devirtualization techniques have been proposed to improve the performance of dynamic method calls. Since devirtualization enables the inlining of methods with dynamic calls, it increases the opportunities for intra-procedure optimizations such as data flow analysis and register allocation.

We now present an overview of our approach. First, the compiler performs flow-sensitive type analysis and preexistence analysis to directly devirtualize call sites without the backup paths that introduce constraints on compiler optimizations such as code motion. In addition, type analysis can reduce the overhead of dynamic method calls that may be invoked with an array object, and the preexistence analysis guarantees that on-stack replacement does not 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 dynamic method call with a class test verifying that the receiver has a particular class, or inlines a dynamic method call with a method test verifying that the receiver has a particular method. The class test is also used to optimize recursive method calls.

In this section, we described devirtualization techniques for the optimization of dynamic method calls: a 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 that support 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.

We have proposed a code patching mechanism in order to directly devirtualize dynamic method calls with dynamic class loading [7]. If class loading overrides a method that has not been overridden, the inlined code sequence for a specific implementation must be replaced with the original dynamic method call. Since Java is an explicitly multi-threaded language, all optimizations must be thread-safe. That is, the code sequence must be invalidated atomically. We implemented this atomic updating by rewriting only one instruction [21] as shown in Example 1 using the PowerPC instruction set. In the example, we assume an object layout that combines the class instance data and the header that are derived from the Sun Java 2 Software Development Kit (SDK) reference implementation [22], so that three load instructions are required to obtain the address of a compiled instruction.

```
Before overriding the method                After overriding the method
    // top word of inlined code      ->      b      original_call          // static jmp
    // the rest of inlined code              // the rest of inlined code
after_call:                                 after_call:
    :                                           :
    :                                           :

original_call:                              original_call:
    lwz   r1, (obj)                             lwz    r1, (obj)        // load class pointer
    lwz   r2, offset(r1)                        lwz    r2, offset(r1)   // load method pointer
    lwz   r3, offset(r2)                        lwz    r3, offset(r2)   // load code address
    mtctr r3                                    mtctr  r3
    blr   ctr                                   blr    ctr              // dynamic method call
    b     after_inline                          b      after_inline
```

Example 1: Example of the inlining of dynamic method call (`invokevirtual`)

We have implemented a code patching mechanism with dynamic CHA for supporting dynamic class loading as follows. When the new class is loaded at runtime, the compiler maintains the internal structure that represents whether or not each method is over-

ridden. 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 call that requires compilation of that method is issued, and the result is then cached 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 entry for that method. 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. The result cache is also stored for use at runtime. When the compiler generates the native code, it places the inlined code in the fall through path in Example 1. Because it knows the inlined code is executed very frequently. This improves the efficiency of the instruction 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 for dynamic class loading, the internal structures are updated appropriately. If the method related to the result cache is overridden, the instruction at the address stored in the result cache is replaced with a **b** instruction to the dynamic method call by the class loader in order to undo the direct devirtualization. Consequently, the code sequence for the dynamic method call will be executed correctly.

Java provides an interface for the implementation 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 is shown using PowerPC instruction set as shown in Example 2. When more than one class implements an interface class, the code patching mechanism cancels direct devirtualization to execute the original interface method call. This optimization is much more efficient than a naive implementation of an interface call, which requires a loop to search for an implementation class.

```
Only one class implements                    More than one class implements
    // top word of inlined code                    b      interface_call            // static jmp
    // 2nd word of inlined code                    // 2nd word of inlined code       // if implementing method is
    // the rest of inlined code                    // the rest of inlined code       // overridden, go to virtual_call

after_inline:                                after_inline:
    :                                            :
    :                                            :
virtual_call                                 virtual_call
    lwz   r1, (obj)                              lwz   r1, (obj)                   // load class pointer
    lwz   r2, offset(r1)                         lwz   r2, offset(r1)              // load method pointer
    lwz   r3, offset(r2)                         lwz   r3, offset(r2)              // load code address
    mtctr r3                                     mtctr r3
    blr   ctr                                    blr   ctr
    b     after_inline                           b     after_inline
interface_call:                              interface_call:
    mr    r1, <rcv obj reg>                      mr    r1, <rcv obj reg>           // move receiver object
    blr   rt_interface                           blr   rt_interface                // call runtime for interface call
    b     after_inline                           b     after_inline
```

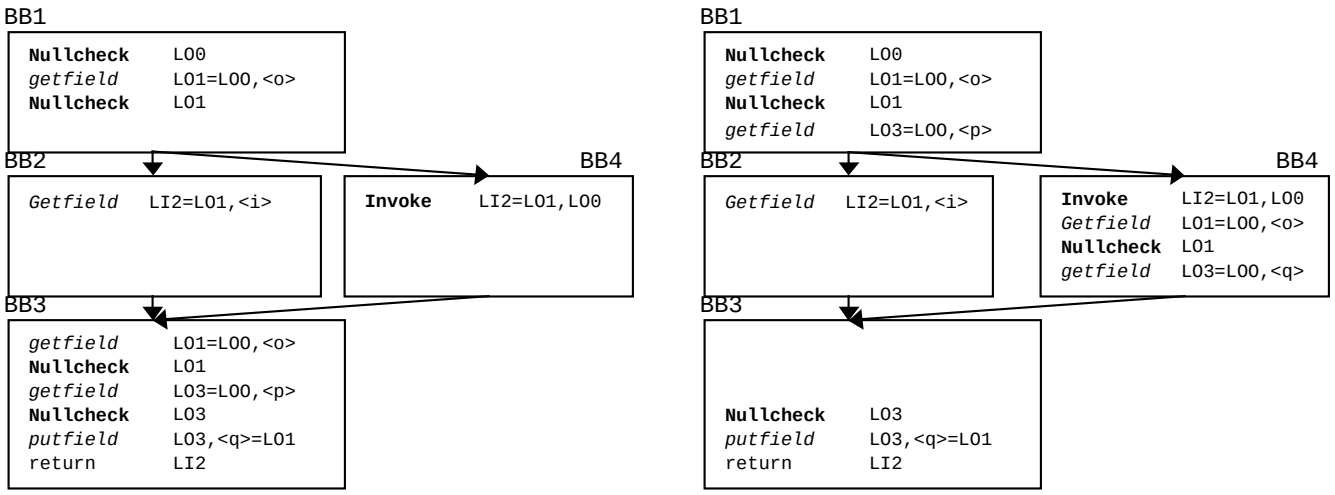
Example 2: Example of the inlining of dynamic method call (`invokeinterface`)

The generated code using direct devirtualization has no overhead at execution time because there are no tests requiring memory access such as method tests and class tests. On the other hand, from the viewpoint of compiler optimizations, generated code using direct devirtualization with the code patching mechanism prevents the compiler from performing some optimizations, because the generated code includes a backup path (i.e. the original method call) as a kill point[i]. Scalar replacement of instance variables and code motion may also be restricted. These problems are illustrated in Example 3 using RISC-like instructions. For example, the `getfield` bytecode instructions are translated into `nullcheck` instructions that are potentially excepting instructions (in the

[i] 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 the point.

bold font) and `getfield` instructions that are simple loads from a heap memory. At the end of basic block (BB) 1, there are implicit branches by direct devirtualization. One is a branch to BB2, which is a primary execution path. The other is a branch to BB4, which is a backup path.

In the example, partial redundancy elimination (PRE) [23] can perform scalar replacement of the access of instance variable `<o>` by generating the compensation code around the kill point (`invoke` at BB4), though increases the size of the code. In this example, three instructions after the `invoke` instruction in BB4 are generated for scalar replacement of LO3. Code motion involving potentially excepting instructions or instructions with side effects is also limited and cannot cross over the kill point. If a `nullcheck` instruction is moved a head of a method call, the exception may be thrown before throwing an exception raised within the callee method. This violates the original semantics of the program. Here, the compiler can move up the `getfield` instruction for LO3 from BB3 to BB1 crossing over BB4, and not move a `nullcheck` instruction involving LO3 across BB4, because that is a potentially excepting instruction.



a) Before PRE and code motion

b) After PRE and code motion

Example 3: An example of partial redundancy elimination and code motion.

The existence of a backup path is a disadvantage compared with devirtualization by a recompilation approach. A recompilation approach does not require backup paths instead of a recompilation of a whole method. To solve the problems, we attempt to directly devirtualize these sites without backup paths by using preexistence analysis and flow-sensitive type analysis that will be explained in the next sections.

3.2 Flow-Sensitive Type Analysis

Flow-sensitive type analysis [10, 11, 12] computes a type for every object reference point in an entire method. The compiler does this by performing data flow analysis on the control flow for the entire method. It computes the data flow information on static types with signatures and class instantiations (call to `new()`) at each object reference point. The analysis determines a set of classes reachable at each object reference point. This analysis has several advantages.

If the type analysis proves that all class instantiations that reach the receiver expression of the dynamic method call have the same definition, then the dynamic method call can be directly devirtualized without a backup path.

Type analysis can also recover missing type information. This loss can occurs when translating source code into bytecode [8] (i.e. during a compilation by javac or jikes [24] (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 a class file. The compiler may recover the more precise type `A` of the receiver through an interpretation like the bytecode verification process [25]. The missing type information causes the

class hierarchy analysis to fail at the method call `a.equals()`. Without 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`. As a result, type analysis improves 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: An example of missing type information at a call site

The methods, `hashCode()`, `toString()`, and `equals()`, are declared as part of an object's nature in the class `java.lang.Object`. 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 these declarations from the class `java.lang.Object`. However, since these implementation 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, the static type of a dynamic method call `equals()`is frequently ambiguous. In that case, the compiler directly devirtualizes the method call by inlining simple callee code.

The method `hashCode()` in the class `java.lang.String` caches the calculated hash value in each instance. If type analysis proves that the object `java.lang.String` reaches only a receiver expression of the method call, it allows the compiler to inline the code of the callee method partially. Partial inlining is a technique to inline a part of a method that will be executed frequently. We have found a good practical example in the method `hashCode()`in the class `java.lang.string.` We show an example in Example 5.

```
public final class String {
   private char value[];
   private int offset;
   private int count;
:
   /* Cache the hash code for the string */
   private int hash = 0;
   public int hashCode() {
      int h = hash;
      if (h == 0) {
         int off = offset, len = count;
         char val[] = value;
         for (int i = 0; i < len; i++) h = 31*h + val[off++];
         hash = h;
      }
      return h;
   }
}
```

a) Implementation of the method hashCode() in the class java.lang.String

```
Object o;
o  = (String)o.value;

h = o.hashCode();
```

before partial inlining

```
Object o
o  = (String)o.value;

h = o.hash;
if (h == 0) h = o.hashCode(); // call a method once
```

after partial inlining

b) An example of partial inlining for the method hashCode() in the class java.lang.String

Example 5: An example of partial inlining using the results of type analysis

In our system, the object layout is derived from the Sun SDK reference implementation. This is shown in Figure 1 a) and b). Though the headers for scalar objects and array objects are the same size, the difference is that the scalar object has a vtable pointer instead of the array length. This requires that a dynamic method call of an array object must be treated as special case. For this purpose, an `invokevirtualobject_quick` instruction in the bytecode is prepared [25]. If a constant pool reference is resolved and a dynamic method call refers to a method in the class `java.lang.Object` or array class is the target of the reference, then an `invokevirtual` instruction is rewritten by an `invokevirtualobject_quick` instruction. At runtime, the receiver object must be checked to see whether it is an array type. If it is an array object, the vtable of the class `java.lang.Object` is used. If type analysis proves that an array class never reaches a receiver expression of a method call, the compiler can generate a dynamic method call without checking the array type and reduce the runtime overhead. On the other hand, every object in HotSpot [17], Marmot [26], and Jalapeno [27] has a vtable pointer and a status field as its first two fields shown in Figure 1 c) and d). Only an array object has an extra field to store the length. In their system, though there is no concern with the vtable problem, an array object requires one extra word. In our system, the header length is same for scalar and array objects to save storage, since type analysis reduces the overhead of the `invokevirtualobject_quick` method call.

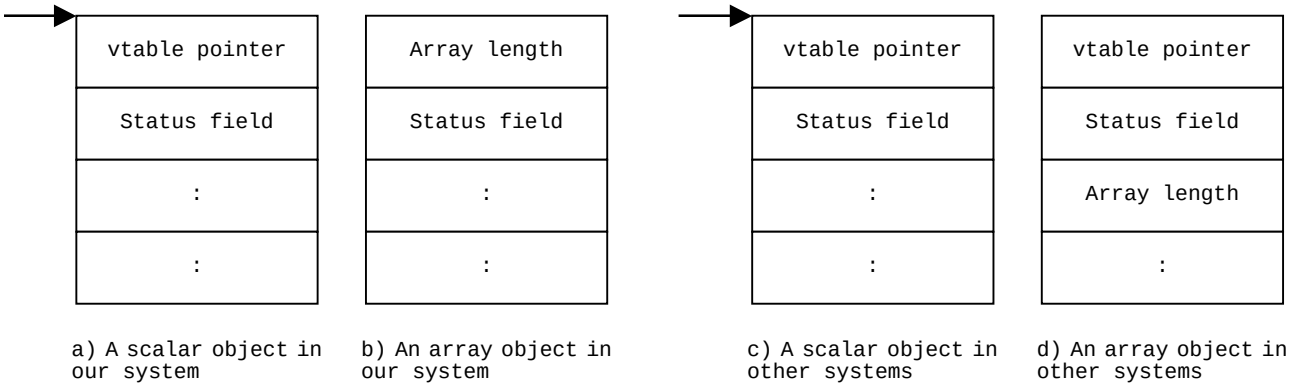


Figure 1: Object layout in Java implementations

3.3 Preexistence Analysis

The concept of preexistence [8] 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. Then, the property can be used to directly devirtualize a dynamic method call without a backup path. It 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 removes inefficiency of generated code.

We have implemented invariant argument analysis [8] to check for the preexistence of a receiver expression. If a receiver expression of a directly devirtualized method call is shown preexistence and the method call has only a single target by CHA at a compilation time, the compiler can directly devirtualize the dynamic 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 results of flow-sensitive type analysis is more accurate, because the merge point that creates the union type is removed from the control flow graph, and the return type of the inlined code is known. To directly devirtualize a dynamic method call without a backup path, the compiler has to record that the caller method must be recompiled at the next invocation when the callee method to be inlined is overridden, instead of patching code at a caller site. Another solution for more precise flow-sensitive type analysis is message splitting [28]. It may increase the code size significantly because it requires copying parts of the control flow, and therefore, we did not explore this alternative.

3.4 Class Tests and Method Tests

In previous research, most systems that use guarded devirtualization have the inlined code with a class test verifying that the receiver has a particular class. In pseudo-machine code, the code generated at an inlined call site shown in Example 6 a).

The class test [2, 3] imposes the reasonable requirement that each object contains a pointer to its class information. The method test [8] imposes a further assumption that the class information includes a vtable. These assumptions are satisfied in our object layout. The generated code for a method test at an inlined call site appears in Example 6 b).

```
r0 = <receiver object>
r1 = load(r0 + <offset-of-vtable-ptr-in-object>)

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

a) pseudo code of a class test

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

}
```

b) pseudo code of a method test

Example 6: Pseudo code for class test and method test

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 inlined code can be executed. Therefore, we have used method test at call sites that have multiple implementations with class hierarchy analysis at compilation time. The overhead of method test is slightly more expensive than that of class test. Our JIT compiler requires two loads to get a vtable entry and a method block for the intermediate representation. This method 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.

We have used a class test for optimizing recursive methods, because the compiler predicts that most of the method invocations will be with the same called object. When a recursive call is invoked by a dynamic method call, the compiler provides two copies of the method. At the method entry point, 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 a true recursive call is unrolled, and the other version is for the general case as it appeared in the original code.

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 as performed using devirtualization. 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 that we described here in a Just-In-Time Compiler. 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, data flow optimizations, loop optimizations, and low-level optimizations. Data flow optimizations are copy propagation, constant propagation, dead code elimination, common subexpression elimination, and elimination of redundant exception checks. The loop optimization uses loop versioning. Low-level optimizations are register allocation, instruction scheduling, and shrink wrapping [29].

The JIT compiler inlines methods except when they have exception handlers and are larger than the maximum size. It inlines methods with both static and dynamic calls until the call hierarchy tree is four levels deep. 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 [9] 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 measurements except one were performed with compiling all methods.

The measurements were performed on an IBM RISC System 6000 Model 7044-170 (containing a 333-MHz POWER3-II with 1024 MB of RAM) running AIX 4.3.3.

4.2 Overview of the Programs

Table 1 shows sixteen 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 in SPECjvm98 [30] do not follow the official SPEC rules.

Program	Description
compress	LZW compression and decompression in SPECjvm98 Run a benchmark with size = 100.
jess	NASA's CLIP expert system in SPECjvm98. Run a benchmark with size = 100.
db	Search and modify a database in SPECjvm98. Run a benchmark with size = 100.
javac	Source to bytecode compiler in SPECjvm98. Run a benchmark with size = 100.
mpegaudio	Decompress audio file in SPECjvm98. Run a benchmark with size = 100.
mtrt	Multi-threaded image rendering in SPECjvm98. Run a benchmark with size = 100.
jack	Parser generator generating itself in SPECjvm98. Run a benchmark with size = 100.
pBOB [31]	Transaction processing benchmark. Version 2.0k. Run a benchmark with a number of warehouse = 1.
XML parser [32]	IBM's XML parser. XML4J version 2.0.13. Run a sample program to parse an XML file.
Java Server [33]	Java Server Web Development Kit 1.0.1. Run a Web server and access it with 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 [34]	A Java version of the xfig drawing program. Version 1.38b. Run an application and open a document.
ICE Browser [35]	Simple Internet browser version 5.01. Run an application and open a Web page
HotJava [36]	HotJava browser version 1.1.5. Run an application and open a Web page.
Ichitaro Ark [37]	Word processor written in pure Java. Run an application and open a document.

Table 1: Descriptions of the programs used in our experiments

4.3 Characteristics of Method Calls

For each program, Table 2 details the characteristics of both static and dynamic method. S-Call is the total number of static calls and V-Call is the total number of virtual method calls. V-Mono means the percentage of virtual method calls that are called at monomorphic call sites. Java means they are within Java class libraries. App means they are within the application. If-Call means the total number of interface method calls. If-Mono means the percentage of interface method calls that are called at monomorphic call sites.

An average of 75.8% (ranging from 33.4% to 99.9%) of the virtual method call are monomorphic. The results show higher usage of that dynamic method calls in programs without GUIs (compress, jess, db, javac, mpegaudio, mtrt, jack, pBOB, XML parser, and Java Server), though compress and Java Server are monomorphic within application classes. Programs with GUIs (swing, Java2D, jfig, ICE Browser, HotJava, and Ichitaro Ark) are monomorphic within the Java class libraries. The results also show the programs except mpegaudio are surprisingly monomorphic. It shows 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 pBOB is a benchmark program to measure throughput in a constant time. The

better compilation results in more executions and therefore more calls, but this result cannot be compared directly with the other results. This characteristic differs from the other programs.

Program	S-Call	V-Call	V-Mono %		If-Call	If-Mono %	
			Java	App		Java	App
compress	225975805	12039	49.6%	25.0%	446	41.3%	58.7%
jess	78375454	36872088	0.2%	83.8%	706505	0.0%	0.7%
db	52992991	52529114	0.1%	97.1%	14931539	0.0%	100.0%
javac	57019624	48408808	5.1%	62.2%	3379096	0.0%	99.8%
mpegaudio	99702499	9853620	0.2%	33.2%	182220	0.1%	99.9%
mtrt	17406471	269740419	0.3%	90.7%	402	46.3%	53.7%
jack	24400198	25219092	20.3%	59.5%	4155315	0.0%	55.0%
pBOB	56775910	72595733	16.7%	79.8%	1618994	0.1%	99.9%
XML parser	5108864	3451279	0.4%	99.5%	5463833	0.0%	100.0%
Java Server	337899	74901	67.9%	11.9%	3118	65.7%	28.8%
swing	3143213	1754935	57.4%	0.3%	177638	49.8%	0.1%
Java2D	17956992	6490662	72.6%	4.1%	1446333	49.3%	0.1%
ifig	1274203	296283	67.4%	0.0%	33006	51.0%	0.5%
ICE Browser	1732313	261235	62.1%	10.3%	47519	67.8%	10.2%
HotJava	1882711	504321	78.8%	0.0%	55523	64.2%	0.3%
Ichitaro Ark	4960087	2421789	23.7%	32.2%	806600	16.4%	16.4%
average			75.8%			73.5%	

Table 2: The characteristics of static and dynamic method calls

4.4 Results of Devirtualization

In this section, we show the results of performing these optimizations in four categories. We perform each optimization cumulatively. At first, we start guarded devirtualization. Secondly, we add direct devirtualization with code patching. Thirdly, we add type analysis. At last, we perform preexistence analysis cumulatively.

Each dynamic method call is applied the devirtualization techniques to as shown in Figure 2.

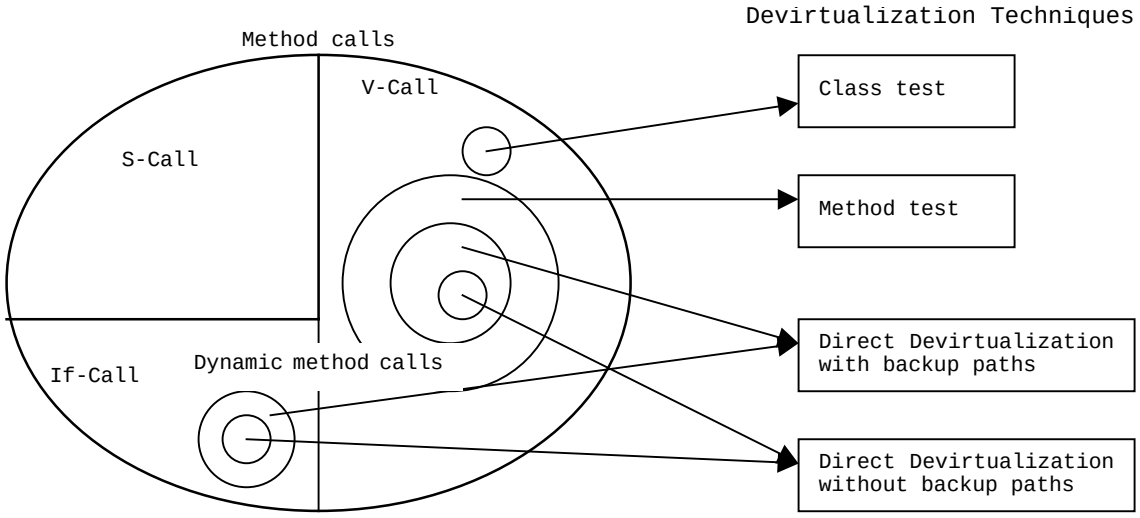


Figure 2: Applicable categories of devirtualization techniques

4.4.1 Guarded Devirtualization

We here started by performing test on guarded devirtualization by class and method tests together. Table 3 shows the characteristics of programs with guarded devirtualization. M-Test means the total count of method tests. C-Test means the total count of

class tests. Inlined execs for each kind of test is the percentage of the actually executed inline code. We apply class test only to method calls that are recursive calls, and apply method test to virtual method calls that have a single or multiple targets at compilation time only for code that can be inlined. We do not apply method test to method calls that can be replaced with direct method calls. Even if we did such testing, the cost of the test and the actual method call would be the same as that without method test in the base Java implementation. If a method call has multiple targets, one of the methods defined in a leaf class is inlined.

We adopted method test for guarded devirtualization, even though the cost of method test is slightly higher than that of class test as presented in Section 3.3. This is because when we attempted to apply class test to mtrt, the success percentage of the inlining was only 70%. Moreover, we can hide this overhead by using compiler optimizations.

As is shown in the following table, the percentages of actually executed inline code at the call sites devirtualized by method test vary range from 50.7% to 100% (an average of 92.7%).

Program	V-Call	If-Call	M-Test	M-Test inlined execs	C-Test	C-Test inlined execs
compress	9967	446	2040	97.2%	6	100.0%
jess	12212470	706505	24660863	100.0%	4547	100.0%
db	46680205	14931539	5833525	100.0%	6	100.0%
javac	41274813	3381204	7973933	90.9%	39930	100.0%
mpegaudio	6821177	182220	3037975	99.8%	6	100.0%
mtrt	75811042	402	193929371	100.0%	6	100.0%
jack	17683893	4155315	7529626	100.0%	4375	100.0%
pBOB	12130906	1605136	60062063	100.0%	24	95.8%
XML parser	1726616	2733183	7163	100.0%	65	100.0%
Java Server	40952	2960	19961	99.1%	283	94.3%
swing	1249252	176796	498142	86.9%	691	99.9%
Java2D	4813273	1453963	2153279	50.7%	1523	100.0%
jfig	205157	33616	96697	86.9%	549	100.0%
ICE Browser	162833	45118	93499	92.0%	186	100.0%
HotJava	346182	56558	108735	90.2%	3877	100.0%
Ichitaro Ark	1558901	594341	663773	90.2%	2285	100.0%

Table 3: Characteristics of programs with guarded devirtualization

4.4.2 Direct Devirtualization with a Code Patching Mechanism

For the next tests, we added direct devirtualization with a code patching mechanism. This is applied to virtual method calls that have only a single target at a compilation time and to interface method calls that are implemented by a single class. We also applied it to method calls only that will be inlined, but to be replaced with direct method calls.

Table 4 shows the characteristics of these directly devirtualized programs. CP-Dev means the total execution count of directly devirtualized call sites with the code patching mechanism. CP-Dev inlined execs refers to the percentage of the actually executed inline code. Invalidation sites means 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. As is shown in the following table, the percentages of actually executed inline code at directly devirtualized call sites vary from 88.8% to 100% (an average of 98.2%).

Program	V-Call	If-Call	M-Test	M-Test inlined execs	C-Test	C-Test inlined execs	CP-Dev	CP-Dev inlined execs	Invalidation sites
compress	9796	443	657	91.2%	6	100.0%	1596	97.7%	18
jess	10790816	701785	10798	82.9%	4659	100.0%	26083410	100.0%	22
db	46557413	14931536	5082	99.2%	6	100.0%	5951271	100.0%	18
javac	29161105	3379389	2157383	66.0%	39930	100.0%	18349878	99.0%	32
mpegaudio	6804623	395	31908	82.4%	6	100.0%	3204602	100.0%	18
mtrt	7244261	399	1667	95.0%	6	100.0%	262494525	100.0%	18
jack	16317679	2624376	33849	99.6%	4375	100.0%	10925027	99.4%	22
pBOB	8133725	1577179	201156	99.9%	22	100.0%	62757008	100.0%	4
XML parser	3442303	2589055	279	100.0%	65	100.0%	2882788	100.0%	13
Java Server	40292	2753	2215	92.3%	283	94.3%	18943	99.0%	44
swing	1251871	164057	222752	63.3%	663	99.8%	373117	94.2%	199
Java2D	5038999	1422935	1504829	20.0%	1588	100.0%	1130901	88.8%	77
ifig	167970	28230	28055	57.6%	547	100.0%	72518	99.1%	43
ICE Browser	144733	37185	11245	47.6%	186	100.0%	96555	99.3%	79
HotJava	316788	47202	19416	52.5%	4067	95.0%	111078	96.5%	158
Ichitaro Ark	1446609	575172	120782	38.6%	4806	100.0%	641550	97.7%	215

Table 4: Characteristics of directly devirtualized programs

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 the following table, the percentages of actually executed inline code at directly devirtualized call sites vary from 84.1% to 100% (an average of 97.4%).

Program	V-Call	If-Call	M-Test	M-Test inlined execs	C-Test	C-Test inlined execs	CP-Dev	CP-Dev inlined execs	Invalidation sites
compress	9585	443	657	91.2%	6	100.0%	1282	97.2%	16
jess	7895376	701785	10798	82.9%	4659	100.0%	24978943	100.0%	20
db	46557233	14931536	5082	99.2%	6	100.0%	5950246	100.0%	16
javac	27540151	3381201	2157387	66.0%	39186	100.0%	19816214	92.6%	54
mpegaudio	6804397	395	31908	82.4%	6	100.0%	3204247	100.0%	16
mtrt	7244059	399	1667	95.0%	6	100.0%	245247103	100.0%	16
jack	12322620	2624376	33849	99.6%	4375	100.0%	13443883	99.5%	20
pBOB	8463698	1626697	211100	99.9%	22	100.0%	64800581	100.0%	3
XML parser	3441954	2589055	279	100.0%	65	100.0%	2880904	100.0%	13
Java Server	38941	2753	2215	92.3%	279	94.3%	16624	98.9%	35
swing	1209389	163713	212813	66.6%	683	99.9%	362306	94.0%	210
Java2D	4802108	1392895	1435134	20.8%	1653	100.0%	1110593	84.1%	91
ifig	181572	29796	28117	57.3%	547	100.0%	68222	99.0%	47
ICE Browser	155810	46080	15774	45.0%	188	100.0%	92482	98.9%	68
HotJava	333557	47946	20206	53.3%	4063	95.0%	102599	96.2%	157
Ichitaro Ark	1374676	542976	102232	39.2%	4814	100.0%	576341	97.5%	217

Table 5: Characteristics of programs using flow-sensitive type analysis

4.4.4 Preexistence Analysis

Finally, we also performed preexistence analysis. Table 6 shows the characteristics of programs including preexistence analysis. Recompilation candidate methods means the number of method recompilation candidates when a class is loaded during the execution of a program and the method is later overridden. As is shown in the following table, the percentages of the actually executed inline code at the directly devirtualized call sites vary from 81.5% to 100% (97.0% on average).

Program	V-Call	If-Call	M-Test	M-Test inlined execs	C-Test	C-Test inlined execs	CP-Dev	CP-Dev inlined execs	Invalidation sites	Recompilation candidate methods
compress	9585	443	669	89.5%	6	100.0%	1059	97.7%	10	6
jess	7895376	701785	10822	82.7%	4659	100.0%	18261070	100.0%	14	6
db	46557233	14931536	5085	99.2%	6	100.0%	5950061	100.0%	10	6
javac	27704969	3379221	2461630	57.9%	39186	100.0%	18199383	91.8%	50	10
mpegaudio	6804397	395	31959	82.2%	6	100.0%	2173264	100.0%	10	6
mtrt	7244059	399	1678	94.3%	6	100.0%	191710141	100.0%	10	6
jack	12322620	2624376	33887	99.5%	4375	100.0%	9314605	99.2%	14	6
pBOB	9727441	1887573	243493	99.9%	21	100.0%	59533300	100.0%	3	0
XML parser	3441954	2589055	279	100.0%	65	100.0%	2880183	100.0%	13	0
Java Server	38949	2753	2215	92.3%	279	94.3%	12500	98.6%	35	0
swing	1266590	163528	221685	64.5%	767	99.9%	279771	93.6%	180	26
Java2D	4783173	1418776	1410625	21.8%	1549	100.0%	919675	81.5%	75	16
ifig	165701	28043	21452	50.0%	547	100.0%	44622	98.8%	28	13
ICE Browser	136834	36675	10212	46.4%	182	100.0%	67071	99.2%	67	2
HotJava	321514	47553	16913	50.0%	5263	96.1%	72835	94.5%	144	16
Ichitaro Ark	1451855	577285	121842	39.9%	4789	100.0%	459639	97.3%	196	22

Table 6: Characteristics of programs including preexistence analysis

4.5 Evaluation and Breakdown of the Results

Figure 3 summarizes the reductions of each operation on some programs applied all devirtualization techniques corresponding to Section 4.4.4. We use "(a)" to denote all optimizations are performed. All values are given in relative execution counts against non-devirtualized version corresponding to Section 4.3.

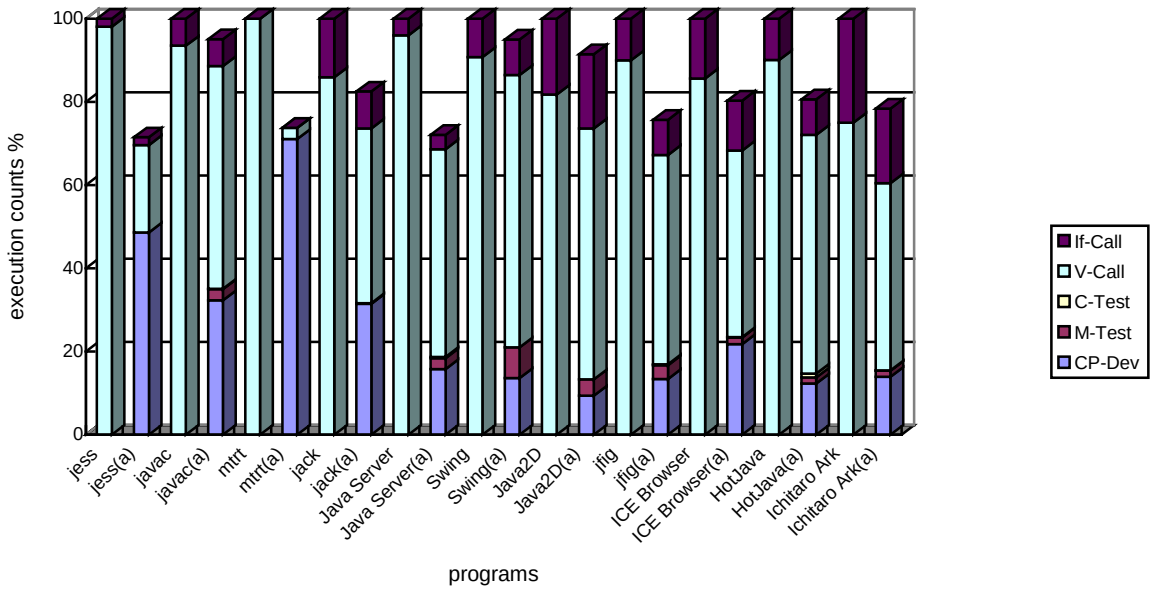


Figure 3: Percentages of total number of each operation

Table 7 also shows details of the reduction of execution counts of each operation. Here, the reductions of pBOB are excluded. The reason is that optimizations increases 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. In some programs, the reductions for method

and class tests are negative. The reason is that the compiler can replace more dynamic method calls with method tests or class tests by using extensive devirtualization.

Program	reduction % from non-inlined to preexistence			reduction % from guarded devirtualization to preexistence		reduction % from a code patching mechanism to preexistence
	V-Call	If-Call	average	M-Test inlined execs	C-Test inlined execs	CP-Dev inlined execs
compress	20.4%	0.7%	19.7%	70.6%	0.0%	35.2%
jess	78.6%	0.7%	77.1%	100.0%	-2.5%	30.0%
db	11.4%	0.0%	8.9%	99.9%	0.0%	0.0%
javac	42.8%	0.0%	40.0%	82.1%	1.9%	8.9%
mpegaudio	30.9%	99.8%	32.2%	99.1%	0.0%	32.2%
mtrt	97.3%	0.7%	97.3%	100.0%	0.0%	27.0%
jack	51.1%	36.8%	49.1%	99.6%	0.0%	15.4%
XML parser	0.3%	52.6%	32.4%	96.1%	0.0%	0.1%
Java Server	48.0%	11.7%	46.5%	89.8%	7.1%	35.0%
swing	27.8%	7.9%	26.0%	71.3%	-10.9%	29.8%
Java2D	26.3%	1.9%	21.9%	85.7%	-1.7%	33.7%
ifig	44.1%	15.0%	41.2%	88.9%	0.4%	39.2%
ICE Browser	47.6%	16.8%	43.2%	94.7%	0.5%	31.1%
HotJava	36.2%	14.4%	34.1%	92.2%	-30.5%	38.0%
Ichitaro Ark	40.1%	28.4%	37.1%	92.7%	-109.6%	30.3%
average	40.2%	19.2%	40.4%	90.8%	-9.7%	25.7%

Table 7: Reduction of execution count for each program

A number of interesting observations can be made from the above results.

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 monomorphicly. This usage pattern based on the experiments with real Java programs is very encouraging. It increases the opportunity for devirtualization without creating a burden for programmers.

As is shown in Table 7, we have measured the reductions of dynamic method calls ranging from 8.9% to 97.3% (an average of 40.4%). The program where we measured the highest reduction in virtual method calls is mtrt. Mtrt has a hotspot loop that calls some small methods to get instance variables very frequently. Devirtualization with the code patching mechanism can eliminate almost all virtual method calls, and furthermore 25.7% of them can be directly devirtualized without their backup paths. We have attempted to execute mtrt compiled with eliminating all backup paths. Even in the extremely case, its version is about 6% faster. The overhead of the existence of backup paths usually may be smaller.

As can be seen from Table 4, Table 5, and Table 6, an average of CP-Dev inlined execs decreases from 98.2% to 97.0% with type analysis and preexistence analysis. It shows that direct devirtualization without backup paths are actually performed. Table 7 also shows a reduction with an average of 25.7% of CP-Dev inlined execs with type analysis and preexistence analysis. We cannot measure execution counts of directly devirtualized sites without backup path 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 covers only 25.7% of direct devirtualization with a backup path by a code patching mechanism. The capability of devirtualization by recompilation approach is same as that by a code patching mechanism.

The benchmark for which we measured the worst rate of executing inlined code at directly devirtualized call sites is Java2D. As can be seen from Table 6, 19.5% of the executions invoke the original dynamic method calls. We are interested in the causes of this behavior. What method calls invokes the original dynamic method calls? Since our JIT compiler provides a selective compila-

tion mechanism, we have measured the percentages of executing inlined code at directly devirtualized call sites with a variety of threshold values to start a compilation. The JIT compiler also starts to compile a method when it defects loops in the method.

Table 8 details the characteristics of Java2D with a variety of compilation threshold values. The result shows that the saturated successful rate is 90% and some recompilation candidate methods remain even when the threshold value is large. It means that 10% of devirtualized code are executed at slow paths in hotspot methods and some hotspot methods are required recompilations. Since the JIT compiler uses part of an application's runtime resources, the overall performance may be degraded. Recompilation approach is better for quality of generated code related to methods without recompilations. According to our experiment, almost all recompilation target methods are within Java AWT and Java2D class libraries. If the compiler knows this property and applies the code patching mechanism to GUI libraries, we believe that a hybrid approach using both code patching and recompilation mechanisms may be better to minimize runtime overhead and improve performance.

Compilation threshold value	M-Test	M-Test inlined execs	C-Test	C-Test inlined execs	CP-Dev	CP-Dev inlined execs	Recompilation candidate methods (by preexistence)
0	1410625	21.8%	1549	100.0%	919675	81.5%	58(16)
2	500061	68.7%	157721	100.0%	563932	81.0%	37(6)
5	474910	68.5%	156726	100.0%	560701	79.9%	33(6)
10	504298	74.7%	156753	100.0%	581531	79.4%	27(3)
20	429620	77.0%	156720	100.0%	558680	81.0%	27(1)
30	450785	77.0%	156593	100.0%	570122	81.5%	22(1)
40	439788	78.8%	156641	100.0%	601561	84.8%	19(1)
50	426514	78.0%	156675	100.0%	579187	84.3%	18(1)
75	444315	79.4%	156712	100.0%	613827	84.2%	18(1)
80	430211	79.7%	156460	100.0%	557131	90.0%	16(1)
100	419284	79.7%	156576	100.0%	544530	90.1%	16(1)
250	420098	88.2%	155902	100.0%	539297	90.0%	16(0)
300	227553	88.1%	155908	100.0%	390761	90.7%	13(0)
400	396661	96.1%	155520	100.0%	512271	89.0%	11(0)
500	381212	96.3%	155326	100.0%	485028	88.9%	10(0)
1000	368031	97.2%	154600	100.0%	438773	88.1%	10(0)
5000	261480	99.0%	150586	100.0%	371151	86.2%	8(0)
10000	368031	97.2%	154600	100.0%	438773	88.1%	10(0)

Table 8: Characteristics of Java2D with a range of compilation threshold value

Finally, we are also surprised that the number of interface method calls is almost unchanged in db. We have investigated the reason using the statistics. The count 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 causes type information for variables to be missing, for example the receiver expression `e` in the method `foo()`. If it is not declared as `final`, another target method may be invoked since type analysis returns the `Enumeration` class as an ambiguous type (i.e. the compiler determines the call site is polymorphic). The `Enumeration` class is always implemented by few classes. If the method is declared as `final`, the method can be directly inlined and the return type is known as an inner class. Therefore, type analysis can only prove an inner class that is never overridden is certain to reach the receiver expressions `e` of the interface method calls. As a result, we can translate interface method calls into virtual method calls or inlined code. In that case, we can still get a huge reduction for 99% of the interface method calls in db. Since the receiver expression is not assigned by arguments, specialization and customization are not effective. Message splitting [28] can help in this situation. However, message

splitting increases the code size by duplicating loop structures completely. This apparently accidental change of sealed classes loses an opportunity for the performance improvement.

```
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 of the usage of interface method calls

4.6 Performance Results

We measured the execution time of the eight non-interactive programs (compress, jess, db, javac, mpegaudio, mtrt, jack, and pBOB). The other programs were difficult to measure because of their interactive nature and dependencies within AWT. Figure 4 shows the performance improvements resulting from the cumulative optimizations. Here, all measurements are performed by compiling all methods. All values are given in relative speed up against non-devirtualized versions. Each of the bar shows cumulative effects including prior optimizations. For each of the bars, the following combinations of techniques are used:

- base (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.
- +guarded dev: Base optimizations and guarded devirtualization (i.e. class and method tests) are performed (corresponding to Section 4.4.1).
- +direct dev: 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).

On average, we have measured a speedup of 4% by guarded devirtualization with class tests and method tests. Direct devirtualization with the code patching mechanism improves the performance by 18% on average. It especially improves the performance of mtrt that calls some small methods very frequently.

Type analysis also improves the performance of jess, javac, mpegaudio, and jack. These programs include parsers and expert systems that manipulate many string objects using the methods `hashcode()`, `equals()`, and `toString()`. In these programs, the reductions of method calls with `invokevirtualobject_quick` instructions by type analysis are higher. These

method calls are a part of V-Call. We have measured the reductions of the total number of method calls with `invokevirtualobject_quick` instructions such as jess with 43.9%, mpegaudio with 11.6%, and jack with 88.4%. We have also measured the higher reduction with javac with 36.3%.

Using all of the optimizations presented in this paper, we have measured a speedup of 19% on average. The degradation of a result of pBOB with preexistence analysis is strange, but we cannot find a reason.

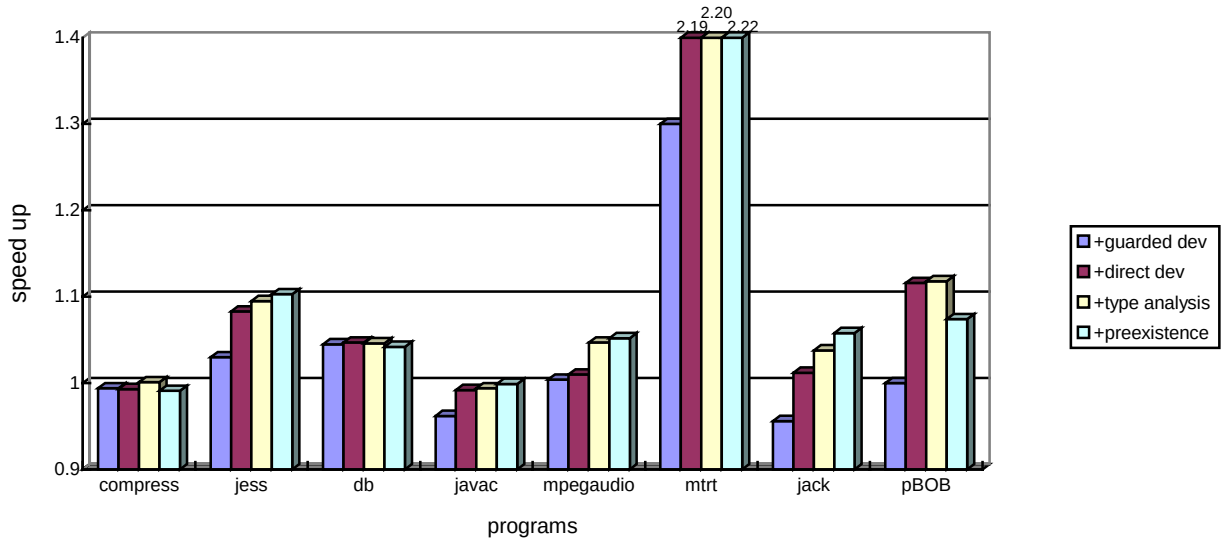


Figure 4: Speed-up measurements for the non-interactive benchmarks

5. Conclusions

We have presented devirtualization techniques that we have implemented here. We evaluated them based on various statistics collected by running a set of real programs in widely different application categories. We have found a reduction of dynamic method calls ranging from 8.9% to 97.3% (an average of 40.4%) by using these devirtualization techniques. We have shown that the direct devirtualization technique that we propose in this paper can remove almost all class and method tests generated by guarded devirtualization, and be applied to a wide range of dynamic method calls. The runtime overhead of this approach is smaller than that of a recompilation-based approach. This approach introduces backup paths that prevent some compiler optimizations, but the overhead of its existence may be small in practice. Furthermore, we have shown type analysis and preexistence analysis can directly devirtualize only 25.7% without backup paths of the directly devirtualized sites with backup paths. And overall we have reported performance improvements ranging from -1% to 122% (an average of 19%).

For a specific application program that is not effectively accelerated by these devirtualization techniques, we have also investigated the behavior of the program with a variety of threshold values of selective compilation. This encourages us to consider adopting a hybrid approach for direct devirtualization. We have also pointed out some problems such as non-sealed class library and missing type information that introduce performance degradation in a Java runtime environment. Now we are beginning to investigate compilation policies to decide which methods should be applied to by a hybrid approach between code invalidation and recompilation-based approaches.

Acknowledgement

We are grateful to the people in Network Computing Platform at Tokyo Research Laboratory for implementing our JIT compiler and for participating in helpful discussions.

References

[1] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*, Addison-Wesley, 1996.

[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 Holzle. 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 Holze. "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] 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.

[8] 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.

[9] Urs Holzle, 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.

[10] 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.

[11] Ole Agesen and Urs Holzle. 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.

[12] Paul R. Carini, Hirini Srinivasan, and Michael Hind. Flow-Sensitive Type Analysis for C++, *IBM Research Report*, RC 20267, 1995

[13] Urs Holzle, 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.

[14] David F. Bacon. Fast and Effective Optimization of Statically Typed Object-Oriented Languages, Ph.D. thesis, University of California at Berkeley, 1997.

[15] Junpyo Lee, Byung-Sun Yang, Suhyun Kim, SeugIl 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*, 2000

[16] David F. Bacon and Peter F. Sweeny. 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.

[17] Sun Corp. "The Java HotSpot Performance Engine Architecture," Available at http://java.sun.com/products/hotspot/whitepaper.html.

[18] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages, In *Proceedings of the Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA '94*, pp. 324-340, 1994.

[19] Craig Camber and David Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language, In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pp. 146-160, 1989.

[20] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages, In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pp. 93-102, 1995.

[21]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.

[22] Sun Corp. Java™ 2 Platform, Standard Edition (J2SE™), at http://www.sun.com/software/communitysource/java2/.

[23] 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.

[24] IBM Corp. Jikes, available at http://oss.software.ibm.com/developerworks/opensource/jikes/project/index.html.

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

[26] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Traditi. Marmot: an Optimizing Compiler for Java, available at http://www.research.microsoft.com/apl/.

[27] Bowen Alpern, C. R. Attanasio, John J. Barton, Michael G. Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn-Hummel, Derek Lieber, Vassily Litvinov, Mark F. Mergen, Ton Ngo, James R. Russell, Vivek Sarker, Mauricio J. Serrano, Janice C. Shepherd, Stephen E. Smith, V. C. Sreedhar, Harini Srinivasan, John Whaley. The Jalapeno virtual machine, *IBM Systems Journal* 39 No.1, pp.211-238, 2000.

[28] 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

[29] 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.

[30] Standard Performance Evaluation Corp. SPEC JVM98 Benchmarks, available at http://www.spec.org/osg/jvm98/.

[31] Sandra Johnson Baylor, Murthy Devarakonda, Stephen J. Fink, Eugene Gluzberg, Michael Kalantar, Prakash Muttineni, Eric Barsness, Rajiv Arora, Robert Dimpsey, Steven J. Munroe. Java server benchmarks, *IBM Systems Journal* 39 No.1, pp.57-81, 2000

[32] IBM Corp. "XML Parser for Java," available at http://alphaworks.ibm.com/tech/xml4j.

[33] Sun Corp. "JavaServer™ Web Development Kit (JSWDK) 1.0.1 Reference Implementation," available at http://java.sun.com/products/jsp/download.html.

[34] Norman Hendrich. "jfig," available at http://tech-www.informatik.uni-hamburg.de/applets/javafig/

[35] ICEsoft. "ICE Browser," available at http://www.icesoft.no/

[36] Sun Corp. "HotJava™ Browser," available at http://java.sun.com/products/hotjava/index.html

[37] JUSTSYSTEM Corp. "ICHITARO ARK for Java," available at http://www.justsystem.com/ark/index.html.