Dynamic Type Checking in Jalapeño

Bowen Alpern *Anthony Cocchi* *David Grove*
IBM T. J. Watson Research Center
Yorktown Heights, NY, 10598
alpern@watson.ibm.com *tony@watson.ibm.com* *groved@us.ibm.com*
`http://www.research.ibm.com/jalapeno`

Abstract

A Java virtual machine (JVM) must sometimes check whether a value of one type can be can be treated as a value of another type. The overhead for such dynamic type checking can be a significant factor in the running time of some Java programs. This paper presents a variety of techniques for performing these checks, each tailored to a particular restricted case that commonly arises in Java programs. By exploiting compile-time information to select the most applicable technique to implement each dynamic type check, the run-time overhead of dynamic type checking can be significantly reduced.

This paper suggests maintaining three data structures operationally close to every Java object. The most important of these is a display of identifiers for the superclasses of the object's class. With this array, most dynamic type checks can be performed in four instructions. It also suggests that an equality test of the run-time type of an array and the declared type of the variable that contains it can be an important short-circuit check for object array stores. Together these techniques result in significant performance improvements on some benchmarks.

1 Introduction

A *type check* is the determination of whether a value of one type, hereafter the *right-hand side* or *RHS* type, can legally be assigned to a variable of a second type, hereafter the *left-hand side* or *LHS* type. If so, the RHS type is said to be a *subtype* of (or *consistent* with) the LHS type and the LHS type is said to be a *supertype* of the RHS type. More informally, the types are said to *match*.

Java [9] is a strongly-typed programming language. Almost all type checking is static: done first by a Java source-to-bytecode compiler and then verified by a Java virtual machine (JVM) when classes are loaded. However, Java's object oriented type system makes provision for some run-time type checking. Explicitly casting a value of one type to another (the `checkcast` bytecode), or testing whether such a cast would succeed (`instanceof`), clearly requires such a check. So does storing an object in an array (`aastore`), dispatching a method through an interface (`invokeinterface`), or catching an exception (`athrow`).

While dynamic type checking overhead is unlikely to be the dominant performance characteristic on many Java programs, it is significant enough on some that it is important that care be taken as to how such tests are computed. This paper presents techniques for exploiting information available at compile time to generate efficient code sequences.

The next section presents some background on Jalapeño. Section 3 investigates the cases (`instanceof`, `checkcast`, and `invokeinterface`) where the putative type is known at compile time. Section 4 deals with the cases (`aastore` and `catch` blocks) where this type cannot be determined until run time. Section 5 shows that careful implementation of dynamic type check can result in significant performance improvements on some benchmarks. Section 6 reviews other approaches to fast dynamic type checking. And, section 7 concludes.

2 Jalapeño Background

The work reported here was undertaken in conjunction with the development of the Jalapeño JVM [1] at IBM's T. J. Watson Research Center. Jalapeño is primarily designed for servers.[1] It is written in Java, but (rather than running on top of another JVM) it runs directly on PowerPC-based multiprocessors running the AIX operating system [2].

Jalapeño does not interpret bytecodes. Rather, bytecodes are compiled into machine code and executed directly.[2] Jalapeño has a *baseline* compiler, which produces inefficient machine code very quickly, and an *optimizing* compiler, which can be used to get efficient machine code for selected methods. While the Jalapeño system can be used in a variety of configurations, this paper focuses on a configuration in which all methods are initially baseline compiled, and those observed to be computationally intensive or frequently executed are *re*compiled by the optimizing compiler. To implement dynamic type checking, the baseline compiler emits code invoking methods that implement the techniques presented in the remainder of the paper, while the optimizing compiler aggressively inlines them.

The next subsection provides a brief overview of the Jalapeño object model. The following subsections describe the prior handling of dynamic type checking by the two compilers. The final subsection deals with the interactions of dynamic class loading and dynamic type checking.

2.1 Jalapeño object model

Types in Java come in three flavors: *primitive* (*e.g.* `int` and `float`), *class* (*e.g.* `Object`, `Truck`, and `Serializable`), and *array* (*e.g.* `int[]`, `Object[][]`, and `Truck[]`). Classes are either *proper classes* (*e.g.* `Object` and `Truck`) or *interfaces* (*e.g.* `Serializable`). Jalapeño represents Java types as objects of the class VM_Type. This class is abstract with three final subclasses: VM_Primitive, VM_Class, and VM_Array. A field of the VM_Class class distinguishes proper classes from interfaces.

Jalapeño also maintains a *type information block* (TIB) for each type. A TIB is an Object array that contains an *interface method table* (IMT) and a *virtual method table* (VMT) for the type. The first slot of the TIB is a reference to its VM_Type object. The work reported in this paper introduces three new slots in the TIB.

Every object has a header. All object headers contain a reference to the TIB for the object's type. (Object headers for arrays contain their length.) Another reference to each TIB is kept in an array of static values called the *Jalapeño table of contents* (JTOC). During execution, a dedicated register holds a pointer to the base of the JTOC. Figure 1 depicts the data structures associated with the TIB of a prototypical class `Truck`. The purpose of the (new) second, third, and fourth slots in the TIB will be discussed in Sections 3.1, 3.2, and 4 respectively. (Note that this picture is a simplification of the Jalapeño object model [1]).

The optimizing compiler's intermediate representation includes special-purpose operators to explictly represent manipulations of the Jalapeño object model. Thus, its entire suite of classical optimizations (common subexpression elimination, loop invariant code motion, etc.) can easily be applied to operations such as loading the TIB from an object (or from the JTOC) or loading the contents of the three new TIB slots.

2.2 Prior baseline dynamic type checking

Jalapeño's baseline compiler emitted code to call Java methods to handle each of the dynamic type-checking bytecodes. These methods in turn called a central `isAssignableWith` method. This method took two VM_Types and returned true, if a value of the second could be assigned to a variable of the first, and false, otherwise.

Occasionally, `isAssignableWith` needed to load classes dynamically. Class loading updates Jalapeño's global data structures, which requires holding a global lock. Acquiring the

[1] The main implications of being a *server* JVM are an obsession with performance, particularly on multiprocessors, and a relative insensitivity to space considerations.

[2] Hereafter, the term "compilation" will refer to translation from bytecode to machine code unless the translation from source code to bytecode is explicitly indicated.

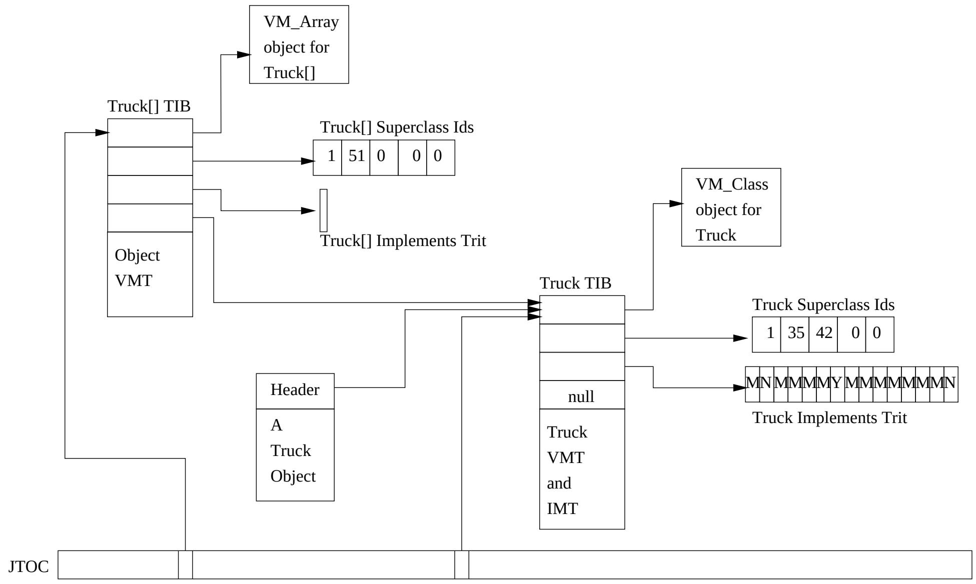

Figure 1: A simplified picture showing the structure of the TIB and associated data structures for the `Truck` class. Truck's TIB is reachable from the JTOC, from the object header of all Truck instances, and from the TIB of Truck[].

global lock to do trivial downcasts and object stores would be costly and would introduce unnecessary contention. Therefore, two short circuit tests are performed to avoid acquiring the lock. The first checks if the two types are equal (and therefore trivially match). If this equality check fails, a two value cache is consulted before acquiring the lock. Every VM_Type object maintains the last RHS type to be found to be assignable with this LHS type and the last RHS type to be found *not* to be assignable with this LHS type.

2.3 Prior optimizations

The optimizing compiler performs type analysis [10, 5] and constant propagation. This enables many `instanceof` and `checkcast` bytecodes to be completely evaluated at compile time. For instance,

- If the RHS value is known to be null, then `checkcast` succeeds and `instanceof` evaluates to `false`.
- If a `checkcast` follows on the true branch of an `instanceof` test for the same LHS type, the checkcast can be eliminated.
- Similarly, an `instanceof` (or `checkcast`) following a `new` of the same LHS type can be eliminated.

More obscure variations of these idioms are also handled. Similarly, the dynamic type check required as part of interface invocation can often be optimized away. Aggressive inlining, performed in tandem with these analyses, uncovers more opportunities to apply these transformations.

If a dynamic type check cannot be so eliminated, the optimizing compiler inlined code to handle one important special case. If the LHS type is known at compile-time to be a final class, there can only be a type match if the RHS type is exactly the same as the LHS type. In which case, the two types TIBs are at the same address. Figure 2 shows assembly code for the sequence used for this test. The LHS TIB is obtained at a fixed offset from the JTOC.[3] The RHS TIB is loaded from the object header. The two TIB pointers are compared. If they are equal, the types match. This test takes two independent loads and a comparison.

In all other cases, the optimizing compiler emitted code to call the same Java methods as the baseline compiler. The techniques de-

[3] A pointer to the LHS TIB cannot easily be stored as a constant in the compiled code in Jalapeño since a copying collector might move the TIB between compilation and execution.

```
L     R1,LHSoffset(JTOC)     // get TIB address of LHS from JTOC
L     R2,TIBoffset(RHS)      // get TIB address from RHS Object
CMP   R1,R2                  // check for equality
```

Figure 2: Assembly code for type equality test.

scribed in the following sections are now inlined by the optimizing compiler to handle these cases.

2.4 Dynamic class loading

If the LHS type of a `checkcast` or `instanceof` bytecode has not loaded when the bytecode is compiled, the compiler cannot determine whether the LHS type is a proper class or an interface. Furthermore, the first execution of the generated code may need to load, resolve, and initialize this type. To handle this situation both compilers emit code that invokes a helper method.

After the first call, the overhead of calling the helper method is superfluous, the type will have been loaded. It is not expected that superfluous calls to the helper method will be frequently executed. Any method containing a frequently executed call site in baseline compiler code will be recompiled by the optimizing compiler. By the time the code is recompiled, any class that has not yet been loaded is unlikely to be referenced often in the future.[4]

Hereafter, it is assumed that the LHS type has been loaded at compile time.

3 When the LHS type is known

The `instanceof`, `checkcast`, and `invokeinterface` bytecodes name the type that the RHS value is supposed to match. This section examines how this information can be exploited if the named class has been fully loaded at compile time.[5] The next subsection addresses the case that the LHS type is a proper class. Subsequent subsections consider interfaces and arrays.

[4]To handle the rare case that a previously unexecuted path becomes the hot path of a method, the helper method could report its caller to the adaptive system's controller [3] every time it is called. If a particular method calls a helper method too often, the controller could schedule it for re-recompilation.

[5]The instanceof and invokeinterface bytecodes fail if the RHS value is null, checkcast succeeds. Often the compiler can infer the result of this test at compile time. It is assumed below that the value is known not to be null.

3.1 Proper classes

This section considers by far the most prevalent case of dynamic type checking: the LHS is known to be a proper class that has been loaded at compile time. To handle this case, Jalapeño maintains two (`short`) fields in each VM_Type: a unique type id and a type depth. The depth of Object (and of primitive types) is 0. The depth of arrays is 1. And, the depth of a class (other than Object) is one more than the depth of its superclass.

Following Cohen [6], one of the new slots in a type's TIB is devoted to a display of superclass ids. The `i`th component of this array (of `shorts`) is the id of the superclass of the type at depth `i`. The id of Object is the zeroth component of this array for every proper class. The class's own id is the depth'th component of its display.

To answer the dynamic type checking question in this case, the RHS TIB is loaded from the header of the RHS object, the RHS superclass ids display is loaded from the TIB, the component of this array corresponding to the depth of LHS is loaded, this value is compared to the id of LHS, the match succeeds if, and only if, these quantities are equal. Both the depth and the id of LHS are compile-time constants.

There is flaw in the above scenario: what if the depth of LHS is greater than the depth of RHS? In this case, the types don't match. But, to check for it requires an additional (semi-independent) load and an additional comparison. This is especially annoying since most classes (particularly those that appear on the left hand side of a dynamic type checking question) have very small depth. Assembly code with and without this checking is shown in Figure 3.

To avoid most bounds check, all superclass

```
Without array bounds checking
  L    R1,TIBoffset(RHS)     // get TIB adddress from RHS Object
  L    R1,SuperclassIDs(R1)  // get superclass ids display
  L    R1,LHSdepth*2(R1)     // get ID from display
  CMPI R1,LHSid              // compare ids

With array bounds checking
  L    R1,TIBoffset(RHS)     // get TIB adddress from RHS Object
  L    R1,SuperclassIDs(R1)  // get superclass ids display
  L    R2,LengthOffset(R1)   // get array length
  CMPI R2,LHSdepth           // is the array large enough?
  BGE  NOMATCH               // if not, the types don´t match
  L    R1,LHSdepth*2(R1)     // get ID from display
  CMPI R1,LHSid              // compare ids
```

Figure 3: Assembly code for subclass test.

ids arrays are padded out to some minimum length[6] (currently six).[7] For any test with LHS depth less than this length, the bounds check can be (and is) omitted. Thus, the bulk of dynamic type checks can be performed in four instructions: three (dependent) loads and a comparison. It is so fast that it does not pay to perform a short-circuit test for the case when LHS and RHS are equal (see Figure 2).

3.2 Interfaces

Closely related to the normal case, but much less prevalent, is the case that the LHS type is an interface.[8] This case occurs for the `invokeinterface` bytecode [12], and for the `instanceof` and `checkcast` bytecodes if the LHS type happens to be an interface.

To handle interfaces, a slot in the TIB is devoted to an array indexed by interface id. The value at an entry in this array tells whether the class implements this interface. Since the result of the question cannot in general be determined before it is asked for the first time,[9] the entries in this array have three possible values: YES, NO, and MAYBE. Whenever a MAYBE value is encountered the proper answer is computed and stored into the array. The values of this *implements trits* array is currently stored as a byte.[10]

Assembly code for this case is shown in Figure 4. In the usual (non-MAYBE) case, the answer may be computed with three dependent load instructions. An additional comparison and usually-untaken conditional branch are needed to account for the MAYBE case. As with the superclass ids display, the need for an array bounds check can be eliminated by padding all implement trits array to a minimum size. However, unlike with proper classes, entries in this fast portion are a scarce resource.[11] When a bounds check is needed, it

[6] Alternatively one could allocate a fixed number of short slots in the TIB to hold the first k entries of the superclass ids display. This has the additional benefit of eliminating a load from the common case test sequence. We chose not to do this in Jalapeño because storing non-reference values in a TIB (which is declared as an `Object[]`) would require special-case extensions to Jalapeño's garbage collectors.

[7] 98% of all the proper classes in our benchmark programs (Table 1), the libraries they use, and Jalapeño itself were covered by using a minimum length of 6. A minimum length to 4 would have covered 92% of them, and a minimum length of 2, 54%.

[8] The number of interfaces is much smaller than the number of proper classes. Most proper classes don't implement any interfaces and many of those that nominally do implement interfaces are, in fact, never used as instances of interfaces. For all these reasons, bytecodes that name interfaces are executed far less often than those that name proper classes.

[9] Java interfaces are not leaded with the class that implements them. Furthermore, either a class or an interface may have changed since the source-to-bytecode compiler established that the class implemented the interface.

[10] If space were a more important consideration, each trit could be encoded in two bits. This would require an additional rotate and mask to unpack its value.

[11] In the current implementation, interface entries are assigned in a first-come, first-served fashion (although we ensure that a few commonly used interfaces such as `java.util.Enumeration` receive a fast entry). In the future, online profile data may be used to determine which interfaces should be assigned to the fast portion of the implements trits array.

```
Without array bounds checking
  L    R1,TIBoffset(RHS)       // get TIB adddress from RHS Object
  L    R1,ImplementsTrits(R1)  // get array of trits from TIB
  L    R1,LHSInterfaceId(R2)   // get trit for this interface
  CMPI R1,1                    //  1 => yes   (class implements intrface)
  BLT  NO                      //  0 => no
  BGT  MAYBE                   // >1 => maybe (further checking required)

With array bounds checking
  L    R1,TIBoffset(RHS)       // get TIB adddress from RHS Object
  L    R1,ImplementsTrits(R1)  // get array of trits from TIB
  L    R2,LengthOffset(R1)     // get length of trits array
  CMPI R2,LHSInterfaceId       // can trits array contain this interface?
  BLE  MAYBE                   // trits array too small => maybe
  L    R1,LHSInterfaceId(R2)   // get trit for this interface
  CMPI R1,1                    //  1 => yes   (class implements intrface)
  BLT  NO                      //  0 => no
  BGT  MAYBE                   // >1 => maybe (further checking required)
```

Figure 4: Assembly code for implements interface test.

requires an additional semi-dependent load of the array length, a comparison, and a usually-untaken conditional branch. If the bounds check fails, the implements trits array is extended and filled in with MAYBEs.

The need to handle the MAYBE case allows for a cute space savings. Most classes never end up of the right hand side of a dynamic type checking question with an interface on the left hand side.[12] We want to avoid allocating a big implements trits array for these classes since it will never be used. (Of course, one cannot tell which classes these are ahead of time.) Therefore all TIBs are initialized with their implements trits slots pointing to a single shared trits array filled with MAYBEs. The routine that handles the MAYBE case checks to see if an object's TIB still points to this shared array and if necessary allocates a new non-shared array before changing the MAYBE to a YES or NO.

A further space savings might be realized by exploiting the observation that equivalence classes of classes could share the same implements trits. All members of an equivalence class derive from the same root class and none of them except the root implement any new interfaces.

3.3 Arrays

The final case where LHS type is known is when it is an array. This case has three distinct subcases depending on whether the innermost element type is a primitive type (or a final class), the Object class, or some other non-final class type.

To support type-checking of arrays, each type is assigned a *dimension* (stored in a `short` field of `VM_Type`): primitive types, -1; class types, 0; and arrays, the number of left brackets ("[") in their descriptors. In effect, the dimension of an array is the number of times it can be subscripted.

For an object to be assignable into a variable that is an array whose innermost element type is primitive, or is a *final* proper class, the object's class must be *identical* to that of the variable: it must be an array of the same dimension with the same innermost element type. In this case, the type equality (figure 2) test serves as the dynamic type check.

This type equality test can be performed for the remaining cases as well. If it succeeds, the types match. Otherwise, further tests are required. If the innermost class of the LHS is a class other than Object, then the RHS must

[12] Instances of only 21% of all types in our benchmark programs (Table 1), the libraries they use, and Jalapeño itself actually appeared as the RHS of a dynamic type checking question when an interface was the LHS.

have the same dimension, and its innermost type must be assignable with that of the LHS. If the innermost class of the LHS is Object, any array of the same dimension whose innermost type is not a primitive type is assignable with it. Furthermore, any array with greater dimension is assignable with it, since any array can be assigned to a variable of type Object.

4 When the LHS type is unknown

This section considers two cases where the LHS type of the dynamic type checking question cannot be determined at compile time. These cases arise when an exception is thrown and when an object is stored in an array.

When an exception (or any other `Throwable`) is thrown, Jalapeño walks up the thread's call stack examining stack frames. In each frame, it determines the call site, finds any try blocks that contain it, and checks to see if the exception (the RHS) type matches the type declared for an associated catch clause (the LHS type). This is a constrained instance of the dynamic type checking question. Both types must be proper subclasses of `Throwable`. The depth and id of the LHS type are obtained from its type object (stored in an object associated with the compiled method). There is a match if, and only if, the entry in the RHS's superclass ids display at the LHS's depth equals the LHS's id.[13]

When an object is stored in an array, a check is performed to make sure that the type of the object is compatible with the element type of the array. It may not be immediately obvious why such a dynamic type check is necessary. After all, Java's source to bytecode compiler (and, at class-load time, the JVM's verifier) guarantee that the type of the object is compatible with the element type of the declared array. The problem is that this declared element type may be less restrictive than the runtime element type.

Consider Figure 5. Variable `x` is declared to be an array of `Object` and variable `y` is declared to be a `Petunia`. At compile time (and classload time), the assignment `x[0] = y;` looks fine because `Petunia` is a subclass of `Object`. However, suppose that `makeObjectArray` returns an array of `Truck`. Unless `Petunia` is a subclass of `Truck` (or `y` is null), the assignment is illegal. This can only be determined at runtime.

```
Object [] x    = makeObjectArray();
Petunia   y    = makePetunia();
          x[0] = y;
```

Figure 5: Object array stores require checking.

This is unfortunate. In the first place, in almost any Java program, although object array stores may be fairly frequent, they fail extremely rarely, if at all. Furthermore, in a typical Java program, almost all of such stores are perfectly innocuous: either the type of the object being assigned is the element type of the array being assigned into, or the runtime type of the array is the same as the declared type of the variable that holds it. To make the former case easy to check, a slot in the TIB is devoted to the element type TIB. (This entry is null in TIBs for non-array types.) To make the short-circuit test the TIBs of both the LHS and the RHS are loaded. Then the LHS's element type TIB is loaded. If this is equal to the RHS's TIB, the match succeeds.

The latter case is even easier, *if* the declared type of the variable containing the array is available at compile-time. The TIB for the LHS's declared type is loaded at a fixed offset from the JTOC. If this is equal to the LHS's runtime TIB, the match succeeds. This short-circuit test is especially nice in that it is oblivious to the type of the RHS value. Thus, the test can be hoisted out of a loop in which some or all of the entries of an array are assigned values.

There are two ways the declared type of an array could inadvertently get estranged from the bytecode. If the array is stored in a pure local variable (as opposed to a parameter or a field), Java's source-to-bytecode compiler has thrown away the declared type (that it must be a subtype of `Object[]` can be inferred). More prosaically, an array of a particular class could get upcast to an array of a more general type (usually `Object`) as a side-effect of

[13] This scheme works even if the LHS type has not been loaded. The depth of such a type appears to be zero (because it has not yet been computed), but the type already has a unique id. Since this id is not the id of Object (slot 0 of the RHS's display) the match fails as it should.

being passed to some generic service method. A prime example is the `arraycopy` method of java.lang.System. Sometimes, such situations can be ameliorated by inlining the offending service method. Notice, that even if the information about the declared type of the LHS has been lost, this test degenerates into a short-circuit test for the runtime LHS type being `Object[]`, an important subtest in its own right.

In the cases that these two short-circuit tests fail, the most prevalent situation is that the LHS array is an array of some proper class (rather than an array of an interface or an array of arrays). A type match can be detected fairly quickly. The LHS element type is loaded from its TIB (available from the failed short-circuit tests). The depth and id fields can be loaded from this type. If the depth'th entry in the RHS's superclass ids display is the element type id, the types match. Otherwise, either the types don't match, or the LHS is an array of interfaces or an array of arrays. These final cases so rare that they can be safely be handled by a helper method without measurable impact on performance.

5 Performance

Jalapeño can be deployed in a multitude of variations. Key discriminates include: the garbage collector (copying or not, generational or not), the compiler (baseline or optimizing, level of optimization) to be used on methods of classes included in Jalapeño's boot image, the compiler to be used on methods of classes loaded dynamically, whether such methods should be compiled immediately when their class is loaded, and whether, under what circumstances, and how methods should be recompiled.

For our experiments, we used a configuration that currently obtains the best Jalapeño performance on the SPECjvm98 benchmarks. It uses a copying, non-generational garbage collector. The optimizing compiler (at optimization level 2) is used to statically compile all methods in the boot image. The first time a dynamically loaded method is invoked, it is compiled using the baseline compiler. Methods observed via online profiling to be computationally intensive or frequently called are selected for recompilation with the optimizing compiler by Jalapeño's adaptive optimization system [3].

The performance impact of the changes to dynamic type checking were evaluated using the SPECjvm98 [7] benchmarks and the Jalapeño optimizing compiler [4]. (Table 1 provides a description of each benchmark, the number of classes that comprise the benchmark, and the size, in bytes, of its class files.) The best elapsed time from 10 runs *during a single JVM execution* of each benchmark is reported. The size 100 (large) inputs were used for the SPECjvm98 benchmarks. The time for the optimizing compiler to compile itself (roughly 80,000 lines of Java source code) is shown as the `opt-compiler` benchmark. Results were obtained on an IBM 43P Model 140 with one 333MHz PPC604e processor and 512MB of main memory running AIX v4.3.

Not surprisingly, the new dynamic type checking implementation only improves the execution times of those benchmarks that perform significant amounts of dynamic type checking: `opt-compiler`, `javac`, and `jess`. Interestingly, these are also the three largest (and arguably the most object-oriented) programs in our benchmark suite. Overall, the largest improvement comes from efficient dynamic type checking for proper classes, although array store checks and interfaces are also factors on some benchmarks.

6 Related work

We are unaware of other work that attempts to exploit the particularities of Java's type system to expedite dynamic type checking. Krall *et.al.* [11] review, and supercede, earlier work on dynamic type checking in a general multiple-inheritance environment.

Oberon [14] is an object oriented language with a simple inheritance model based on type extensions [13]. Cohen [6] presents a display-based technique for constant-time dynamic type checking in this setting. This work did not pad displays to avoid the array bounds checks. Pfister, et. al. [8] pad displays (to eight elements) but restricts the maximum depth of inheritance. Jalapeño does not have such a maximum, rather it uses a minimum display size (6) to eliminate the bounds check in almost all cases.

Benchmarks	Description	Number of Classes	Class File Size (in bytes)
`compress`	Lempel-Ziv compression algorithm	12	17,821
`jess`	Java expert shell system	150	396,536
`db`	Simple memory resident database	3	10,156
`javac`	JDK 1.0.2 Java compiler	175	561,463
`mpegaudio`	Decompression of audio files	54	120,182
`mtrt`	Two-thread ray-tracing algorithm	25	57,859
`jack`	Java parser generator	55	130,889
`opt-compiler`	Jalapeño optimizing compiler	393	1,378,292

Table 1: The benchmark suite. The first seven rows are the SPECjvm98 benchmarks.

Benchmarks	Prior DTC	New DTC	Only Classes	Only Interfaces	Only Arrays	Only AAStore
`compress`	41.16	40.95 (1.01)	41.17 (1.00)	41.82 (0.98)	41.60 (0.99)	42.51 (0.97)
`jess`	24.66	23.09 (1.07)	24.63 (1.00)	23.35 (1.06)	24.53 (1.01)	24.58 (1.00)
`db`	66.66	63.66 (1.05)	67.13 (1.00)	66.70 (1.00)	67.97 (0.98)	63.79 (1.04)
`javac`	42.63	35.33 (1.21)	38.80 (1.10)	41.81 (1.02)	42.91 (1.00)	42.17 (1.01)
`mpegaudio`	22.55	22.24 (1.01)	22.00 (1.03)	23.33 (0.97)	23.89 (0.94)	22.57 (1.00)
`mtrt`	19.42	19.12 (1.02)	19.66 (0.99)	19.04 (1.02)	18.81 (1.03)	19.20 (1.01)
`jack`	35.82	35.43 (1.01)	36.66 (0.98)	35.66 (1.00)	35.88 (1.00)	37.12 (0.96)
`opt-compiler`	146.29	95.76 (1.53)	119.47 (1.22)	153.14 (0.96)	148.37 (0.99)	124.42 (1.18)

Table 2: **Execution times in seconds.** The second column gives the execution times using Jalapeño's prior implementation of dynamic type checking. The third column shows the total impact of the new dynamic type checking implementation. The last four columns show the performance obtained by selectively enabling the new dynamic type checking implementations only for proper classes, interfaces, arrays, and array store checks. The number in parenthesis is the speed relative to the `Prior DTC` configuration.

If a program's type hierarchy were available when the JVM started executing, type checking information could be encoded as a two-dimensional *typecheck* array, the i,j-th entry being on if, and only if, the i-th type were a subtype of the j-th. It would be convenient to keep a row of this array in a slot in i's TIB. Where the LHS type of a type check is known at compile time, the tests would require three dependent loads, a rotate and mask (to unpack the j-th bit), and a comparison. Object array store checks would require one or two extra loads to obtain j's index from its VM_Type object. A major drawback to this approach is that the size of the type check array is N^2 in the number of types.

The typecheck array is mostly sparse. A significant compression arises as a by product of a different type check procedure. Krall *et.al.* [11] propose associating with each type a small set of integers such that RHS is a subtype of LHS if, and only if, the set associated with LHS is a subset of the set associated with RHS. They consider various methods of constructing such sets. The two-dimensional array of answers to these subset questions encodes the type-check array at an impressive space savings.

An obvious drawback to both these schemes is that a Java program's type hierarchy is *not* available *a priori*. Thus, a three-valued array element (as with the implements trits array used above for interfaces) seems to be required. And, as even the number of types is not known in advanced, an array bounds check may also be needed with every type check. Krall *et.al.* give an incremental algorithm for computing their type sets, but only if the sets for a type's immediate parents are known when the type is loaded. Because the interfaces a class implements are not loaded with it, this condition does not hold for Java.

7 Conclusions

Dynamic type checking can contribute a significant component to the overall runtime of a Java program. It is, therefore, important that the most common instances of such tests be executed as efficiently as possible when they are executed frequently. This paper presents techniques for exploiting the peculiarities of the Java language, and its type system in particular, to perform these tests efficiently.

It suggests maintaining three data structures within easy striking distance of an object.

The first, and most important, of these is a display of the ids of the superclasses of a type. In most cases where the prospective supertype is known at compile time, this allows a dynamic type check to be performed in four instructions. (Padding this array out to a known minimum size eliminates most array bounds checks.) This array is also useful when the prospective supertype is not known at compile time (*e.g.* the `aastore` bytecode).

The second is an array that tells whether the object's class implements a particular interface. Since this question cannot be definitively answered before the first time it is asked, entries in the array must be three valued. Dynamic type checking questions where the prospective supertype is an interface type are answered efficiently using this array.

The third data structure is only used to determine whether it is legitimate to store a particular object in a particular array. It allows quick access to information about the element type of an array from the array object. Jalapeño uses the TIB of the element type, but the class object of the element type could be used instead.

The paper also suggests that testing for the equality of a runtime array with the declared type of the variable that contains it (where possible) is an important short-circuit test for object array stores.

The resultant dynamic type checking system is much more space efficient than those based on two-dimensional typecheck arrays, is more friendly to dynamic class loading (and more space efficient) than those based on compressed typecheck arrays, and, on some benchmarks, performs significantly better than the earlier Jalapeño system.

Acknowledgments

Without the ongoing support of the entire Jalapeño group, this work would not have been possible.

References

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

[2] Bowen Alpern, Dick Attanasio, John J. Barton, Anthony Cocchi, Derek Lieber, Stephen Smith, and Ton Ngo. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 314–324, 1999.

[3] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 2000.

[4] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño dynamic optimizing compiler for Java. In *ACM 1999 Java Grande Conference*, pages 129–141, June 1999.

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

[6] Norman H. Cohen. Type-extension type tests can be performed in constant time. *ACM Transactions on Programming Languages and Systems*, 13(4):626–629, October 1991.

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

[8] Beat Heeb Cuno Pfister and Josef Templ. Oberon technical notes. Research Report 156, Eidgenossische Technischle Hochschule Zurich- Departement Informatik, March 1991.

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

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

[11] Andreas Krall, Jan Vitek, and R. Nigel Horspool. Near optimal hierarchical encoding of types. In *Proceedings ECOOP '97*, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[12] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification Second Edition.* The Java Series. Addison-Wesley, 1999.

[13] Niklaus Wirth. Type extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, April 1988.

[14] Niklaus Wirth and Jurg Gutknecht. *ProjectOberon: The Design of an Operating System and Compiler.* Addison-Wesley, 1992.