Extending Java Virtual Machine with Integer-Reference Conversion

Yutaka Oiwa, Kenjiro Taura, Akinori Yonezawa
University of Tokyo*

Abstract

Java virtual machine (JVM) is an architecture-independent code execution environment. It is recently used not only for Java language but also for other languages such as Scheme and ML. On JVM, however, all values are statically-typed as either immediate or reference, and types are checked before the execution of a program to prove that invalid memory access will never occur. This property sometimes makes implementation of other languages on JVM inefficient. In particular, implementation of dynamically-typed language is very inefficient because all possible values including frequently-used ones such as integers must be represented by instances of a class.

In this paper, we introduce a new type into JVM, which is a supertype of reference types and a tagged integer type. This allows a more efficient implementation of dynamically-typed language on JVM. It does not require any new instruction, maintains binary-compatibility of existing bytecode, and retains the safety of the original JVM. We modified an existing Scheme system running on JVM to exploit this extension and got factor of 20 speedup for simple integer functions. Our extension imposes little performance penalty on existing JVM code generated from Java; we observed essentially no penalty for Spec JVM benchmarks.

1 Introduction

Java virtual machine (JVM) [9] is a widely-used, machine-independent code execution environment. Although JVM is originally designed for Java language, it is increasingly used as a compilation target of other languages, such as Scheme [5] and ML [2]. Those systems directly generates JVM bytecode without using Java language source code.

Using JVM bytecode as compilation target has several advantages [6]. The high availability of JVM makes the languages more portable. Advanced JIT compilers can easily achieve high execution performance without constructing a dedicated native compiler. A rich set of libraries for graphical user interface, multi-threading, supporting distributed objects, etc., can be used from within those languages.

JVM's static typing, however, prevents efficient implementation of some languages. In JVM, all values are statically-typed as either immediate or reference,

*email: {oiwa, tau, yonezawa}@is.s.u-tokyo.ac.jp

(a) Scheme program with dynamic type

```
1: (define a '(x . y))   ; define variable and store a reference into it
2: (set! a 4)            ; store integer into a
```

(b) invalid JVM program translated from (a)

```
 0 new #3 <Class ConsCell>
 3 dup
 4 ldc #1 <String "x">
 6 ldc #2 <String "y">
 8 invokespecial #9 <Method ConsCell(java.lang.Object,java.lang.Object)>
11 putstatic #10 <Field java.lang.Object a>
14 iconst_4
15 putstatic #10 <Field java.lang.Object a> // invalid operation
```

(c) valid JVM program which expresses the operation of (a)

[*instructions* 0–11 *are same as* (b)]

```
14 new #4 <Class java.lang.Integer>
17 dup
18 iconst_4
19 invokespecial #8 <Method Integer(int)>
22 putstatic #10 <Field java.lang.Object a>
```

Figure 1: Expressing dynamic type on JVM bytecode

and types are checked before the execution of a program to prove that invalid memory access will never occur. This property sometimes makes implementation of other languages on JVM inefficient, particularly of dynamically-typed or polymorphic languages such as Scheme and Smalltalk.

Implementation of those languages (e.g. Scheme) on native CPUs generally represent an immediate value (e.g. integers) in a single word (called *unboxed* representation), so that they can be efficiently manipulated. In Scheme, for example, Program (a) shown in Figure 1 is a valid program. Variable `a` is used twice, once for an reference to a cell and once for an integer value, and most Scheme implementations represent both in a single word. Unfortunately, such representation is not permitted on JVM, since it accepts only statically typed programs. Program (b) in the same figure, which is a direct translation of (a), is not statically typed and rejected by JVM's bytecode verifier. As a result, implementations on the original JVM must represent all dynamically-typed values as Java object—so called *boxed* representation. For example, Program (a) must be translated to Program (c).

In this representation, numerical operation cannot be performed directly. A

Original program is: `(set! c (+ a b))`.

```
void function(Object a, Object b) {
    int a_value = (Integer)a.intValue();
    int b_value = (Integer)b.intValue();
    c = new Integer(a_value + b_value);
}
```

Figure 2: Handling Scheme values with boxed object

single numerical operation is performed as follows:

1. "unbox" the object to get its value,
2. perform the numerical operation to get the result, and
3. allocate a new object and "box" the result value.

The example code for this operation is shown in Figure 2. Of the three steps, the final step is the most problematic, as it causes huge numbers of objects to be allocated throughout the execution of a typical program. Generally the execution cost of a memory allocation is significantly larger than simple operations such as integer additions. Furthermore, it creates many discarded objects which have to be garbage collected, which results in frequent garbage collection. Both of these effects significantly slow down program execution.

To solve the above problem, we extend JVM with a new type which can hold both references and unboxed integers and execute arithmetic on unboxed integers without memory accesses. Our extension has the following advantages:

- It does not introduce any new instruction.
- It does not slow down existing code.
- It retains the same safety properties as the original JVM's.
- It can be implemented efficiently thanks to JIT compiler.
- It is easy to implement based on existing JVMs.

We implemented those extensions based on Kaffe OpenVM 1.0.b3 [16]. We also implemented a Scheme system that exploits our extension based on Kawa Scheme [5], which is a Scheme on JVM. The original Kawa Scheme boxes all values.

Our extension defines only unboxed representation for integer, as most native implementation for dynamically-typed languages uses boxed representation for floats and strings.[1] Also, for representation for "symbol", we can efficiently use Java's `reference` to interned `String`. Consequently, we consider that providing unboxed integer is sufficient enough.

This paper is organized as follows: Section 2 describes the extension to JVM specification which we propose. Section 3 discusses the safety of this extension.

[1] This usually corresponds to Java's `Stringbuffer`, not to `String`.

Section 4 briefly explains the implementation of this extension and Section 5 explains the implementation of Scheme language using this extension. Section 6 evaluates this extension by performance measurement. Section 7 compares this extension with related works. We finally state conclusions in Section 8.

2 Design of the extension

As discussed in the previous section, the main goal of our extension is to allow efficient implementation of dynamically-typed languages on JVM. When we extend JVM, we must pay close attention to three important goals.

- First, the extended VM must not introduce any security hole which does not exist in the original JVM. To satisfy this goal, a naive extension that introduces the casting operation in C language cannot be used. We utilize type information to retain safety, just like the original JVM does.
- Second, the extended JVM must be compatible with the original JVM; it must accept all bytecodes that are valid in the original JVM and retain their semantics. This "backward compatibility" is, we think, the minimum requirement for extension proposals. We recognize that keeping "forward" compatibility is very good idea, but is not absolutely required for extensions. In fact, JDK1.2 is not forward compatible for JDK1.1.
- Finally, changes needed to extend existing JVMs must be small.

Our extension meets these requirements. In particular, we avoid adding new instruction to JVM, as the instruction set is considered to be fixed by many people.

The brief summary of this extension is as follows:

1. Introduce *descriptor*, the super-type of integer and object
2. Represent descriptor in class file format.
3. Extend `CHECKCAST` instruction to allow the coercion from descriptor to an Object or its subtype.
4. Extend bytecode verifier to retain type safety.

2.1 Descriptor type

As previously stated in Section 1, in order to express dynamically-typed values, we need a type which accepts both unboxed integer and reference of any type. That is, it is the supertype of both integer type and Object type. As JVM does not have such a type, we introduce a new type into JVM and integrate it into existing statically-typed system.

The new type is called "descriptor" and is the disjoint union of integer and Object. A variable of descriptor type can hold one of the following values:

1. *non-referencing descriptor*: 31bit-range signed integers.
2. *referencing descriptor*: any reference to an object.

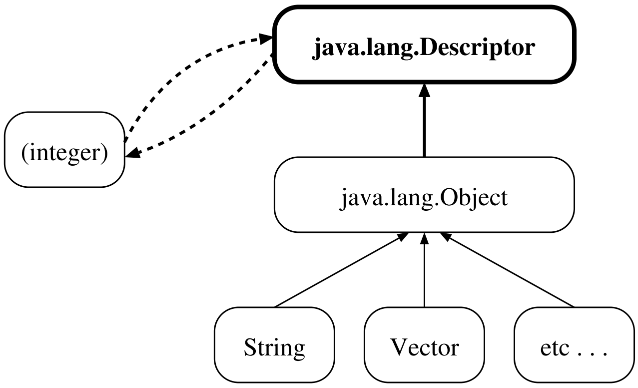

Figure 3: Descriptor type in Java class tree

```
public static native java.lang.Descriptor makeDescriptor(int);
public static native int getDescriptor(java.lang.Descriptor);
```

Figure 4: Type coercing method between integers and descriptors

The internal representation for those values is implementation dependent, but representations of those values must be disjoint, i.e. any non-referencing descriptor must be distinguishable from any referencing descriptor. This property can be easily maintained on most platforms, because references typically do not fully utilize 32 bits. For example, if a reference is represented by the address of the referenced object, it is typically 4 byte aligned, so an integer can be represented by setting its least significant bit (LSB). For another example, a reference might be represented by an index to an object table, which then locates the real object. In this case, the index may not be a multiple of four, but are typically small integers. In this case, a natural way to represent an integer would be to set its most significant bit (MSB). Our extension assumes almost nothing about internal representation for references in original JVM implementation.

To use descriptors in JVM programs, it must be expressed in some way in the bytecode `.class` file. This representation should distinguish descriptors from both integers and references, as it is important for type safety. For the ease of using descriptor from existing Java tools, we should map the descriptor type to something that already exists in current `.class` format.

For this purpose, we define `java.lang.Descriptor` class as a superclass of `java.lang.Object` for the representing descriptors (Figure 3). This class has two static member functions in Figure 4 for converting from/to integer values. `Java.lang.Object` becomes a subclass of `java.lang.Descriptor`. It means that conversion from a reference to a descriptor needs no explicit instructions, and that conversion from a descriptor to a reference needs explicit `CHECKCAST`. It also means that semantics of programs that does not use descriptors is unchanged, because `Descriptor` class stays completely outside the class tree in the original JVM. `Descriptor` class should not be inherited by other than `Object` class. System class archives, for example `classes.zip`, are modified to reflect these changes.

2.2 Coercions among references, integers and descriptors

Rules regarding conversions between references and descriptors follow their subtype relationships.

- A reference is implicitly converted into a descriptor without runtime type check. That is, a reference can be assigned into a variable of descriptor, passed into a formal parameter of descriptor type, and so on.
- On the other hand, conversion from a descriptor to a reference needs a runtime check. For this purpose, we extend `CHECKCAST` instruction, which already exists in JVM, so that it can take a descriptor as its object argument.

`CHECKCAST` in JVM takes two arguments, an object o and a class c and succeeds if the runtime type of o is c or its subtype. We extend `CHECKCAST` instruction so that o can be a descriptor. Given a descriptor, `CHECKCAST` first checks if it is a referencing descriptor. If it is not, the conversion fails. Since the extension requires non-referencing and referencing descriptors to be disjoint in representation, this check prevents any dangling reference from being generated.

We similarly extend `INSTANCEOF` instruction, which essentially checks if `CHECKCAST` would succeed for given arguments. `INSTANCEOF` instruction in our extended JVM can take a descriptor as its object argument. In particular, `INSTANCEOF java.lang.Object` tests whether the descriptor is referencing or non-referencing.

An integer value can be converted to a descriptor by `makeDescriptor` method. `makeDescriptor(`x`)` makes non-referencing descriptor from integer x. This conversion discards the MSB (except for the sign bit) of the integer and tags the value as non-referencing descriptor. A descriptor can be converted to an integer by using `getDescriptor` method. If the descriptor was converted from an integer that fits in 31 bits (including one sign bit), `getDescriptor` returns the original integer. If the descriptor was converted from a reference, on the other hand, the conversion still succeeds, but the return value is undefined. Note that it is not an error to convert a referencing descriptor to an integer, because pointer safety is still maintained. An advantage of this specification is that `getDescriptor` is maximized; it does not require any runtime checks to see if the descriptor was generated from an integer.

Using a static method for conversion from/to integers might seem inefficient because of the method call overhead. However, because dynamic dispatch is not necessary for a static method call, inlining those methods should be fairly easy. Once these methods are inlined, it can achieve the same performance as builtin JVM instructions. This technique can be used universally to extend JVM without altering its instruction set.

2.3 Bytecode verification

Our extension needs some modification to the bytecode verification rules. The operations allowed for descriptors are intersection of those allowed for integers and those allowed for references. Specifically, we allow the following operations to be applied to descriptors on the operand stack:

- duplicate on/discard from stack: `DUP*`, `POP*`
- store value into location: `ASTORE`, `PUTFIELD`, `PUTSTATIC`
- store it into array of descriptors: `AASTORE`
- pass it to another method as argument: `INVOKE*`
- return it to callee method: `ARETURN`
- check the type of the descriptor: `CHECKCAST`, `INSTANCEOF`

Fields, arrays, and local variables can be typed as descriptor and accessed through `GETSTATIC`, `GETFIELD`, `ALOAD`, `AALOAD`, if corresponding location is typed as a descriptor. All other operations are prohibited for descriptors. In particular, invoking instance methods on a descriptor is not allowed.

3 Safety

"Safe execution" is one of the JVM's important characteristics. To make our extension upper-compatible to the original JVM, the safety properties must be maintained. Although the formal proof is far beyond the scope of this paper, we informally claim that our extension does not invalidate any of the original Java safety properties.

JVM's safety properties can be summarized as follows:

Pointer Safety: Every value used as a pointer is in fact a valid pointer to a Java object. The type of pointer must agree with the type of object.

Data Privacy: One can access an object only when it obtains the reference to the object through the regular dataflow defined by JVM (assignments, parameter passing, etc.). In other words, one cannot access an arbitrary address in the heap to find secret information.

Access Control: Accesses to system resources are properly controlled. (e.g., applets denied to access local file.)

In this section, we assume that original JVM specification satisfies those properties. [11] has shown that a subset of JVM specification meets those properties.

Pointer safety is maintained by the following two properties.

1. If a value is statically typed as a descriptor at a program point, and it happens to be a non-referencing descriptor at runtime, then it is not used as a pointer at that point.
2. If a value is statically typed as reference at a program point, then the value is in fact a valid pointer to a Java object.

We maintain the first property simply by disallowing all pointer-dereferencing instructions (except for `CHECKCAST` and `INSTANCEOF`) to take a descriptor as the argument; Specifically, our extension prohibits all instructions which access instance, including `PUTFIELD`, `GETFIELD`, `MONITORENTER`, and `INVOKEVIRTUAL`, to take a value statically typed as descriptor. Exceptions are `CHECKCAST` and

INSTANCEOF, which first check if the argument is a valid Java reference before dereferencing the pointer.

The second property is maintained by a proper subtyping relationship; descriptor type is a supertype of reference (`java.lang.Object`), therefore a variable of reference type cannot hold a descriptor.

By "data privacy", we specifically mean that a program can obtain a reference to an object only through the regular dataflow (assignments, parameter passing, etc.) permitted by JVM; if a module wishes to make an object x inaccessible from another module M, it can do so by making sure that x never reaches M through method parameters, return values, or other objects' fields. Our extension clearly maintains data privacy because we did not add any operation that produces a reference that has not been reached the same module. In particular, if a CHECKCAST returns a reference r at runtime, r must have previously been assigned to the descriptor argument of that instruction, which means that r has already reached the module before. Therefore, one cannot use CHECKCAST to "fabricate" a reference to an object that would otherwise be inaccessible.

Note that pointer safety alone could be maintained without tagging descriptors, but tagging is mandatory for data privacy. To maintain pointer safety, we could insert runtime checks before every pointer dereference, or even better, these runtime checks can be eliminated if CHECKCAST succeeds only when the descriptor is a valid pointer; CHECKCAST could conservatively assume that every data which appears to be a valid pointer is in fact a valid pointer, even if it may originally be an integer. This is the same technique as conservative garbage collectors [4, 3]. This tag-less descriptor might be beneficial for applications that need 32 bit integers. However, it would violate data privacy, because in this scheme an arbitrary reference could be generated by converting an integer to a descriptor and then CHECKCASTing it to a reference. A malicious program might in this way access every object in the system.

The specification of our extension allows converting a referencing descriptor to an integer using `getDescriptor` method. An alternative specification would be to disallow this conversion by adding runtime check to `getDescriptor`. However, converting a referencing descriptor to an integer does not introduce any security problem because the converted value is not usable for memory operations. Clearly, this conversion makes the behavior of bytecode program implementation-dependent, but Java already has some implementation-dependent methods (e.g., `makeHash()`). In short, we chose this unchecked conversion for better performance.

4 Implementations

To evaluate the performance of our extension, we have made an experimental implementation based on Kaffe OpenVM version 1.0.3.

In our implementation, descriptors have the same bit width as references, which is the natural pointer width. Our implementation uses LSB as the tag bit to distinguish referencing descriptors from non-referencing ones. Odd values (LSB = 1) represent non-referencing descriptors, and even values (LSB = 0) referencing descriptors. Because all objects are aligned to 4 bytes boundary, converting a reference into a descriptor is no-op.

Our implementation inlines two type coercing methods described in Section 2.1. We modified the Just-In-Time compiler so that `INVOKESPECIAL` generates an inlined sequences for `makeDescriptor` and `getDescriptor`.

In our current implementation `CHECKCAST` instruction always check the LSB even if the argument's static type is a reference. The effect of this change will be evaluated later.

As of writing, our implementation is still a subset of this extension. Specifically, it uses the type `java.lang.Object`, instead of `java.lang.Descriptor`, for the representation of descriptor type. Therefore our implementation is still not secure, although the specification is. This does not affect performance.

5 Application: Kawa Scheme with unboxed integers

Kawa Scheme is the almost-full-featured Scheme implementation on the JVM. For efficiency, Kawa has internal compiler which compiles Scheme closures (functions) into Java bytecode at definition time. Based on Kawa Scheme, we implemented Scheme which uses unboxed descriptor representation for integers.

Original Kawa scheme gives boxed representations to all Scheme values by defining its own class for integers and other numerics, and therefore it performs unbox and box operation upon every arithmetic. Extended implementation uses descriptor for small integers that fit in 31 bits (including one sign bit). We call those integers *fixnum*. Range overflow of fixnum is checked at every operation, and overflowed values are represented by boxed objects, just as in the original Kawa Scheme.

Routines for numeric operation procedures are re-implemented with descriptor extension to handle unboxed (descriptor) integer directly. Extended routines inline operations on fixnum arguments directly into compiled code. Inlined code first checks whether arguments are fixnum using `INSTANCEOF` instruction. If both arguments for binary operator are fixnum, the operation is performed inline. Otherwise, it delegates it to an auxiliary routine which is written in Java language. The auxiliary routine handles both fixnum and boxed arguments correctly but is slower. Example output of the compiler is shown in Appendix A.

6 Performance evaluation

In this section, We evaluate the performance of the descriptor extension. All experiments are performed on Sun Ultra Enterprise 4000 (Ultra SPARC 168MHz) with Solaris 2.6. In these tests, Kaffe's runtime stack overflow detection is disabled, which is expensive enough to mask the differences.

6.1 Performance of a dynamically typed language

First, we evaluate the performance gain on Scheme code. We use four simple tests that perform many integer arithmetic operations and procedure calls.

1. the program which calculates the 25th element of Fibonacci sequence. (Figure 5).

```
(define (fib x)
  (if (< x 2)
      1
      (+ (fib (- x 1)) (fib (- x 2)))))
```

Figure 5: Fibonacci function used in evaluation

test program	original	modified	ratio
Fibonacci	53	2.0	26.5
Coins	94	3.0	31.3
Prime	148	9.0	16.4
Pi	1029	28	36.7

(unit: sec.)

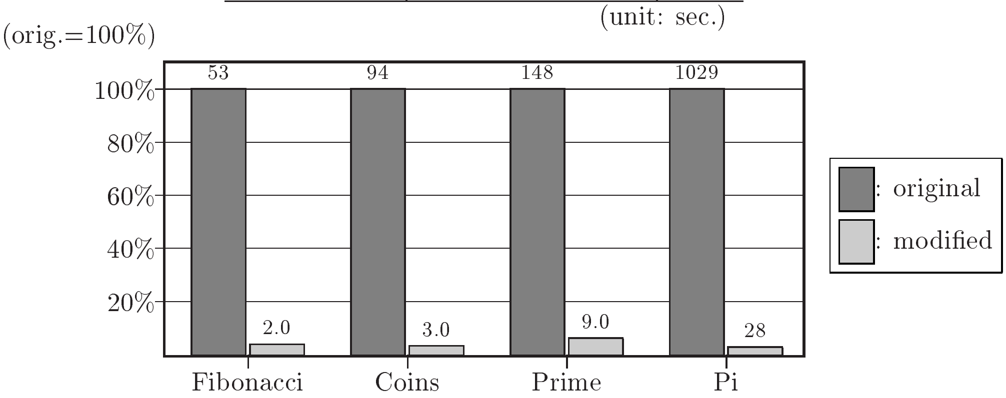

Table 1: Performance improvement of Kawa Scheme with descriptor extension

2. the program which enumerates all possible way to pay 200 cents by the combination of 5 coins (50, 25, 10, 5, 1)

3. the program which build the list of prime numbers up to 40 000.

4. the program which calculates the first 1 000 digits of π by explicit multiple-precision arithmetic. This program is distributed with SCM[8] distribution.

Table 1 shows the result. Performance gain varies for each program, but the extended Scheme with unboxed integers is about 15 to 30 times faster than the original Kawa Scheme. This shows that unboxed integers are vitally important for dynamically-typed languages.

6.2 Performance for integer arithmetic

Second, we compare the descriptor arithmetic with primitive integer arithmetic. We wrote simple Fibonacci function in Java language with three styles. The first version (a) is the simplest version that uses regular Java integers. The second version (b) represents every integer by a descriptor. Every arithmetic first converts a descriptor into an integer and then converts the result back to a

test program	time	ratio
(a) integer	658.048	1.000
(b) descriptor	950.795	1.445
(c) optimized	874.420	1.328

(unit: sec.)

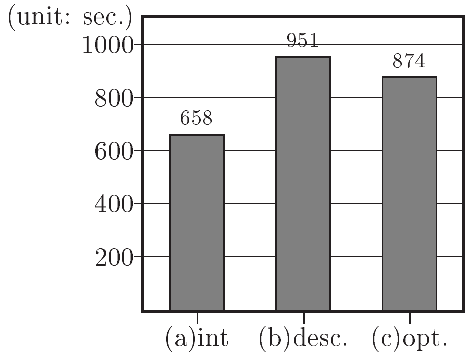

Table 2: Performance of descriptor handling

descriptor. This code is intended to show the performance of dynamically-typed languages with a simple compiler that represents every value by a descriptor. The third version (c) represents method arguments and return values by descriptors and represents all intermediate values by the regular Java integer. This code is, in contrast to (b), intended to show the performance of dynamically-typed languages with a compiler that performs a local type inference to omit unnecessary conversions. Many compilers of dynamically typed languages perform such an optimization.

Results are shown in Table 2. In this test, program (b) runs approximately 40% slower than Program (a). Also, optimized program (c) is about 10% faster than non-optimized (b). 40% performance penalty in (b) seems to be large, but the optimization decreased the penalty significantly. Note that test program is function-call dominant so that the effect of the optimization is limited. In realistic programs, optimization with local type inference will be more effective and make performance of programs which utilizes descriptors closer to that of programs with native integer.

6.3 Performance of Java programs

Finally, we compare the execution speed of Java programs on the regular and our extended JVM. We use five test programs from Spec JVM benchmark suite [14],[2] specifying data set size to 10% (option `-s 10`).

Results are shown in Table 3. In this table, "original" shows the execution time on the unmodified Kaffe VM, and "extended" shows that on the VM with descriptor extension. The values in the graph are the total execution time of 5 trials (i.e., lower value is better). No performance penalties are observed

[2]Unfortunately, only five SpecJVM programs runs correctly on Kaffe OpenVM 1.0.b3. Other programs in the test-suite are therefore not used.

test program	original	modified
201 compress	97.582	92.200
202 jess	66.012	62.007
213 javac	128.044	114.709
222 mpegaudio	155.363	155.889
228 jack	448.631	413.598

(unit: sec.)

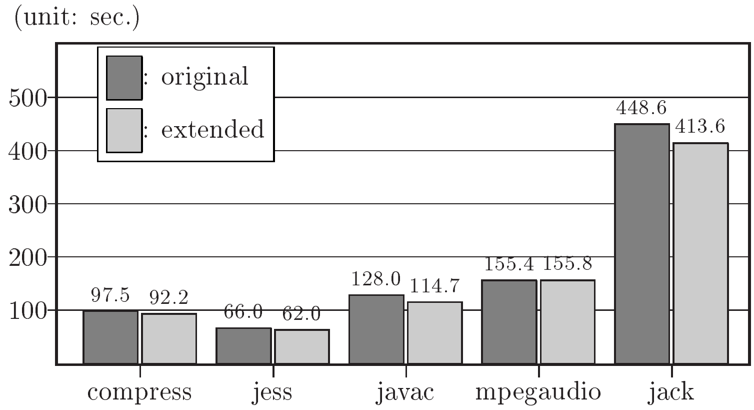

Table 3: Performance comparison with existing Java benchmark

with this experiments. We conjecture that the overhead posed by `CHECKCAST` is overwhelmed by an optimization of `AASTORE` in our experimental VM, that has to be implemented for using `Object` class as a representation of descriptors.[3]

7 Related work

Shivers' proposal [13] is closest to ours. He introduce `DirectDescriptor` class as a subtype of `java.lang.Object` (Figure 6). A variable of type `DirectDescriptor` can hold a tagged integer. That is, `DirectDescriptor` is very similar to our non-referencing descriptor. The critical difference of these extensions is that he introduces `DirectDescriptor` as a *subtype* of `java.lang.Object`, whereas we introduces `Descriptor` as a *supertype*. As a consequence, in Shivers' proposal, it is possible that an integer value held by a `DirectDescriptor` variable is then assigned into a `java.lang.Object` variable, like `Descriptor` variable in our extension. It implies that JVMs can no longer assume that every value of `java.lang.Object` type is a valid pointer. Changes necessary for implementing his extension will be significant. Our subtype relationship is more natural and does not break the assumption that every `java.lang.Object` value is a valid pointer. It does not seem to have been implemented and its performance has not been published.

Virtual machines that are closer to native machines than JVM, such as Omniware [10, 1] and MIC [12], can be used as intermediate languages for dynamically typed languages. These system perform a run-time check for every

[3] We plan to measure the performance of an original Kaffe VM with the same optimization.

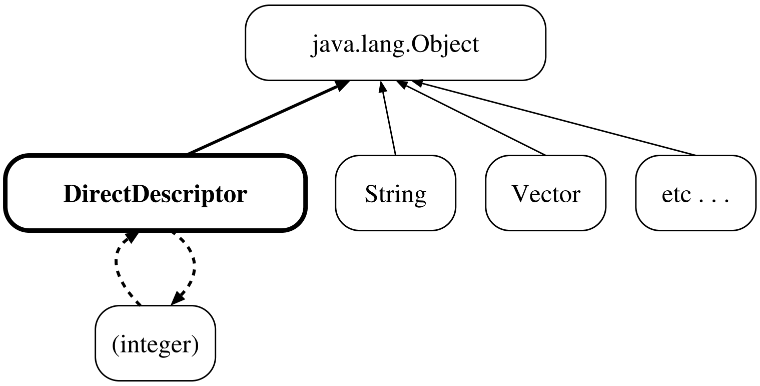

Figure 6: DirectDescriptor type in Shivers' extension

```
void example(void) {
  DirectDescriptor desc
           = ⟨some code to generate descriptor⟩;
  Object o = desc; // o does not refer any object
  int    i = o.makeHash();       // must be re-implemented
  String s = String.valueOf(o); // must be re-implemented
  ⋮
}
```

Figure 7: Problem on DirectDescriptor

memory access in order to ensure validity of the access (so called *sandboxing*). Our system does not perform most of the runtime checks since the static type system ensures validity of most memory accesses. Also, we can exploit many existing JVM implementations and libraries for Java language.

Persimmon ML is a compiler of standard ML for JVM. ML is statically-typed, but it has polymorphic types; for example, `(fn x => x)` is the identity function which can be applied to values of any types. Generally, implementation of polymorphic types has the same problems as those of dynamic type. Persimmon ML solves this problem by generating monomorphic functions from a polymorphic functions. For example, when a static analysis determines that a polymorphic function takes a string value or an integer value as its argument, the compiler generates two Java methods for this function, one for string arguments and the other for integer arguments. Since the analysis requires the entire program, supporting separate compilation is hardly supported with this optimization. With our extension, a more generic solution, which is used in many ML compilers, can be implemented.

There is many attempt to improve the performance of bytecode program through optimizations at JIT compiler stage. For example, escape analysis[15] checks whether object can be alive beyond the current method's scope. If some objects are known to be alive only inside the method, these can be allocated on stack, not in heap storage, to eliminate garbage collection costs. Semantic expansion makes operations for some standard classes such as `java.lang.Integer` inlined into native codes. With these optimizations, performance loss caused by *intra-method* object handling could be reduced. However, *inter-method* overhead, i.e. cost on passing an integer to some method or returning it from a method to callee, cannot be eliminated. Figure 2 in the previous section indirectly tells that *inter-method* overhead cannot be ignored.

In some cases inter-method optimization is also available [7], but it seems difficult to perform such optimization for functions on dynamically-typed language, because function's return value has generic type such as `Object` (or `Descriptor` with our extention). In our extended Kawa Scheme, for example, even simple integer arithmetic functions may return both fixnum unboxed integer and "bignum" boxed integer, depending on the value's magnitude.

8 Conclusion

We extended JVM for supporting efficient implementation of dynamically-typed languages by adding new type called descriptor, which are the union of integers and references. It retains JVM's safety properties, including pointer safety and data privacy. We implemented extended virtual machine with a Scheme system running on top of it, and evaluated their performance. Scheme programs that frequently use integer operations became twenty times faster, while regular Java programs did not show any noticeable performance loss. These results show that JVM can be a much more general execution platform than it is today, if a very modest proposal would be considered.

Acknowledgement

We are grateful to Dr. Masahiro Yasugi at Kyoto University for his valuable suggestion on this research.

References

[1] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and language-independent mobile programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, pages 127–136, May 1996.

[2] Nick Benton, Andrew Kennedy, and George Russel. Compiling standard ML to Java bytecodes. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pages 129–140, January 1999.

[3] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 197–206, 1993. See `http://reality.sgi.com/boehm_mti/gc.html`.

[4] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[5] Per Bothner. Kawa: the Java-based Scheme system. In *Lisp Users Conference*, January 1999. See `http://www.cygnus.com/~bothner/kawa/`.

[6] Jonathan C. Hardwick and Jay Sipelstein. Java as an intermediate language. Technical Report CMU-CS-96-161, School of Computer Science, Carnegie Mellon University, August 1996. See `http://www.cs.cmu.edu/~scandal/html-papers/javanesl/`.

[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 *Proceedings of the 1999 ACM Javagrande Conference*, June 1999.

[8] Aubrey Jaffer. Scm. Interpreter is available from `http://www-swiss.ai.mit.edu/~jaffer/SCM.html`.

[9] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[10] Steven Lucco, Oliver Sharp, and Robert Wahbe. Omniware: A universal substrate for Web programming. In *Proceedings of the 4th International World Wide Web Conference*, December 1995. See `http://www.w3.org/pub/Conferences/WWW4/Papers/165/`.

[11] Tobias Nipkow and David von Oheimb. Java-light is type-safe—definitely. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages (POPL '98)*, pages 161–170, 1998.

[12] Tatsuro Sekiguchi. MIC (machine independent code). Not yet published. contact address: `cocoa@is.s.u-toyo.ac.jp`.

[13] Olin Shivers. Supporting dynamic languages on the java virtual machine, April 1996. See `http://www.ai.mit.edu/~shivers/javaScheme.html`.

[14] Standard Performance Evaluation Corporation. SPEC JVM98 benchmarks, 1998. Information is available online from `http://www.spec.org/osg/jvm98/`.

[15] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. Presented at ACM 1999 Conference on Object-Oriented Programming, Systems, Languages, and Applications, 1999.

[16] Tim Wilkinson. Kaffe—a free virtual machine to run Java code. Information and implementation are available from `http://www.transvirtual.com/kaffe.html`.

A Compiled code from descriptor-enabled Kawa Scheme

Our descriptor-enable Kawa Scheme produces following bytecode for function `(define (f x y) (+ x y)))`. This code is for the full-set of descriptor extension. For subset implementation described in Section 4, change all `Descriptor` to "`Object`".

Firstly, check whether both arguments are fixnum.

```
 0: aload_1
 1: aload_2
 2: dup2
 3: instanceof #16 <Class java.lang.Object>
 6: ifne 42
 9: instanceof #16 <Class java.lang.Object>
12: ifne 43
```

Get two integers from descriptor and add it.

```
15: invokestatic #20
          <Method java.lang.Descriptor.getDescriptor(java.lang.Descriptor)>
18: swap
19: invokestatic #20
          <Method java.lang.Descriptor.getDescriptor(java.lang.Descriptor)>
22: iadd
```

Check whether the result overflows. If overflow occurs, method which creates boxed integer is called. otherwise, `makeDescriptor` creates unboxed integer.

```
23: dup
24: ldc #21 <Integer 1073741824>
26: iadd
27: iflt 36
30: invokestatic #25 <Method java.lang.Descriptor.makeDescriptor(int)>
33: goto 39
36: invokestatic #31 <Method gnu.math.IntNum.make(int)>
39: goto 46
```

Control reaches here from the branch at the first step. Call Java-written fallback routine.

```
42: pop
43: invokestatic #37 <Method kawa.standard.plus_oper.addTwo
                        (java.lang.Descriptor,java.lang.Descriptor)>
46: areturn
```