EXTENDING JAVA VIRTUAL MACHINE TO IMPROVE PERFORMANCE OF DYNAMICALLY-TYPED LANGUAGES
動的型付き言語の効率向上のための Java 仮想マシンの拡張

by

Yutaka Oiwa
大岩 寛

A Senior Thesis
卒業論文

Submitted to
the Department of Information Science
the Faculty of Science
the University of Tokyo
on February 16, 1999
in Partial Fulfillment of the Requirements
for the Degree of Bachelor of Science

Thesis Supervisor: Akinori Yonezawa　米澤 明憲
Professor of Information Science

ABSTRACT

Java virtual machine (JVM) is a widely-used code execution environment which does not depend on any architecture, and it is recently used not only with Java language but also with other languages such as Scheme and ML. On JVM, however, all values are statically-typed as either immediate or reference, and its consistency is verified before execution to prove that invalid memory access will never happen. 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 thesis, I extended the JVM by conversions between references and integers and by runtime legality check of the references, without modifying the instruction set. This makes implementation of dynamically-typed language on JVM more efficient, without breaking both binary-compatibility of existing bytecode and safety from invalid memory accesses. I also modified an existing Scheme implementation to use this extension and got about 93% reduction in calculation time of integer recursive functions. Performance penalty for existing code is currently from 0% to about 20% and can be removed by static type information.

論文要旨

Java virtual machine (JVM) は、広く使われているアーキテクチャに依存しないコード実行環境であり、Java だけではなく、最近では Scheme や ML など他の言語の実行環境としても用いられるようになってきている。しかしながら、JVM は実行中に現れるすべての値について静的に整数型と参照型などの区別を決定し、その制約が守られていることを実行前に検証することにより実行時に不正なメモリアクセスが生じないことを保証しているので、他の言語を JVM 上にコンパイルすると非効率的なコードになることがある。特に、型が実行時に動的に定まるような言語では、整数型など頻繁に使われる型についても JVM 上のオブジェクトとして実装する必要があり、整数型の演算の度に実行時にメモリの確保などを伴い効率が著しく悪くなる。

この論文では、JVM の命令セットを変更することなく、セマンティクスのみを拡張し、参照型と整数の相互変換と、参照型に対する動的な正当性チェックを導入する。これにより、既存の Java のバイナリコードとの互換性と、JVM の持つ実行時に不正アクセスを起こさない特性を保ちつつ、動的型を持つ言語の効率の良い実装が可能となる。また、この拡張を用いて、既存の Scheme の処理系を改良し、整数演算のみからなる帰納関数の計算において 93% 程度の実行時間の減少を得た。また、既存のコードにかかるオーバーヘッドは、現状では 0% から 20% 程度であり、これは静的な型情報を利用することでさらに改善できる。

Acknowledgements

I am very grateful to my thesis supervisor Akinori Yonezawa. I also express my gratitude to Dr. Kenjiro Taura for his many helpful advices. I would like to thank all members of Yonezawa Laboratory and Kobayashi Laboratory for valuable daily discussions.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Java virtual machine (JVM) [7] is a widely-used, machine-independent code execution environment. It reads sequence of virtual machine instructions called *bytecode* from file or through user-supplied routine, and execute it with compilation or interpretation. The JVM is designed as stack-based virtual machine, and each instruction takes some operands from the top of operand stack and sometimes takes some from instruction code. I name the latter *instruction operand*. All instructions of JVM are typed statically except for those of simple stack manipulation. The conformance for these type restrictions are checked when the code is loaded, and the program which does not conform to it are rejected before execution. This ensures that all instructions receive the operands which has presumed type, and that references are created only from reference constants without runtime check. To reduce the performance overhead which is caused from interpretation without explicit preparation, many JVM software compiles bytecode on the first time the method is called. This technique is called *Just-in-time Compilation* (JIT). JIT performs register allocation for the stack of virtual machine so that overhead from stack-based execution model is reduced.

Although JVM is designed mainly for supporting Java language [4], as well-tuned softwares for JVM develops, it is used with languages other than Java languages. For example, Kawa Scheme [3], which used for basis of this experiments, is the execution system for Scheme language [6], and Persimmon MLJ [2] is those for ML language. These systems produce JVM bytecodes directly, without through Java language source code.

1.2 Dynamic type and static type

Programming languages can be categorized to three groups. The first group is *dynamically-typed* languages. On these languages, variables has no type associated and accepts any type of values to be stored. Also, these types are disjoint any value have only one type. Most Lisp dialects, Perl, Python, Ruby are in this group. This kind of language need runtime check for type error, which is difficult for low-level language.

The second group is *statically-typed* language. In these languages, all variables have type associated before execution, which is called *static type*. Program can not use one variable for more than two type of unboxed object. The languages falls into this category are Java and ML. Seen as language, JVM's instruction set is also in this group. Languages in this group is executable as same as untyped language once consistency of static type is checked. This is one of the major reasons why JVM employs static type.

The last group is untyped or loosely-statically-typed language. Native machine language for most CPUs are typical untyped language. It shares same register to both integer and pointers. Languages like C and C++ are statically-typed in basis, but its reason is not for keep type-independency but only for programmer's ease, and coercion between reference and integers is allows. In both cases, type is not determinable from the value and keeping type consistency is programs' responsibility and system does not care about it.

1.3 Motivation — handling dynamic type

To implement dynamically-typed language, programmer have to construct some dynamic type identification mechanism. In the implementation, frequently used values such as integers must have efficient representation for faster execution. Systems for Lisp language dialects usually use "tagging" to achieve this on loosely-typed C language. Tagging assumes some invariants for reference and use the values which does not meet the invariants for immediate values. In most case, the invariants is that references are aligned to the address which is multiple of 4 or 8. For example, Ruby interpreter uses all odd integers to represent Ruby's integers. Also, SCM Scheme interpreter [5] categorize the machine-native integer as shown in Figure 1.1 and use each for specified types.

However, because JVM is statically-typed language, system does not permit to operate on the value of reference. Therefore, tagging technique cannot be used on JVM. Program (b) in Figure 1.1 is sample program which is not accepted by JVM's type system. In this case, immediate values must be expressed by temporal boxed object. For example, in Program (c) of Figure 1.1 temporal integer object is created by new operator in line 2.

type	**pattern for LSB**
integer	`10`
character	`11110100`
special constants	`101110100`
symbol	`001110100`
special symbols	`00` $\boxed{\text{ID}}$ `100`
location	`11111100`
cell reference	`000`

Table 1.1: tagging scheme used in SCM

(a) Scheme program with dynamic type

```
1: (define a '(x . y))   ; define variable and store a reference into it
2: (set! a 5)            ; store integer into a
```

(b) invalid Java program translated from (a)

```
1: Object a = new ConsCell("x", "y");
2: a = (Object) 5;                          // invalid operation
```

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

```
1: Object a = new ConsCell("x", "y");
2: a = new Integer(5);                      // box integer into object
```

Figure 1.1: Expression of dynamic type on Java language

In this solution, numerical operation cannot be performed directly. The operations are performed as follows:

1. take contained values off from the object
2. perform numerical operation
3. allocate new object and store the result into it

The example code for this operation is shown in Figure 1.2. In these three steps, the final step is problematic. It makes huge numbers of objects allocated throughout the execution. Generally the execution cost of memory allocation is significantly larger than simple operations such as integer addition. In addition to this, this makes many

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 1.2: Handling Scheme values with boxed object

objects discarded which have to be collected by the garbage collection. This causes garbage collection frequently. Both slows the execution of program in a large scale.

To solve the problem above, I extend JVM to handle new type which can be used to handle both integers and references, and to handle integer operations as immediate value provided that other reference values can also be handled. I also implements those extensions on existing systems Kaffe JVM and Kawa Scheme and evaluates the performance improvement.

1.4 Outline of the thesis

This thesis is constructed by four parts. Firstly, chapter 2 discuss the extension to JVM specification which I propose. In this chapter, I explain the extension in the sections from 2.1 to 2.5. Also, I discuss about the safety of this extension in section 2.6. Secondly, I explain the implementation which I have done for experimentation in chapters 3 and 4. The former describes the extension to JVM, and the latter describes the extension to a Scheme environment. Experiment on this extension is done at chapter 5. In this chapter I evaluate the implementation by measuring the execution time for existing test programs. The last part is conclusions and review for related work, in chapters 6 and 7.

Chapter 2

Extending Java virtual machine with descriptor

As discussed in previous chapter, the goal of my extension is to implement dynamically-typed languages efficiently on JVM. To this goal, I loosen type restrictions on JVM and introduce new type which accepts both integers and object references to be stored, provided that invalid memory access through the references is kept to be forbidden.

Another goal of this extension is compatibility with existing virtual machine. Any existing binary codes for JVM should be running well on the extended virtual machine, and changes to the JVM implementation should be small. To hit this objective, I decided not to extend the instruction set of JVM, but only change the semantics of the instructions. Implementation is done on the Kaffe version 1.0.b3.

The brief summary of this extension is:

1. Introduce *descriptor*, the super-type of integer and references
2. Introduce execution-time check at the coercion from descriptor to reference
3. Forbid memory access through descriptors
4. Define the representation of descriptor on Java bytecode
5. Extend garbage collector to handle descriptors

I discuss detail of the extension in following sections.

2.1 Introducing descriptor

On JVM, all locations such as operand stack and local variables have static type. The consistency between static type of the location and the value type which is stored in the location is check by *verifier* using flow analysis. For example, on program in figure 2.1,

(Assume that a boolean value is stored in local variable #1)

```
0: iload_1       Get value from local variable #1 to the stack top
1: ifeq 4        Go to line 4 if stack top is 0 (false)
2: iconst_1      Push integer constant 1 to the stack top
3: goto 5        Go to line 5
4: aconst_null   Push constant reference null to the stack top
5:
```

Figure 2.1: Example code which cause type error on existing JVM

type error is detected on line 5. This program has two flows $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 5$ and $0 \rightarrow 1 \rightarrow 4 \rightarrow 5$ depending on a result of the branch on line 1. The value on the top of the operand stack at line 5 cannot have any static type, because it is integer 1 in the former flow and is reference `null` in the latter flow. Through bytecode verification, original JVM prevents invalid reference to be generated, and guarantees that the reference always points some valid object except for `null`.

In this thesis, I introduce *descriptor* as the super-type of integer type and all reference type. Descriptors have 32bit significance and can be used to integer arithmetics. In addition, descriptors can be coerced to reference using the instructions `INSTANCEOF` and `CHECKCAST`. Other reference operations, such as virtual function invocation and memory access are not defined on descriptors. With this extension, The stack-top value at line 5 on figure 2.1 has the descriptor type.

In the real use of descriptor extension, some values guaranteed never to reference no object is needed, e.g. for tagging in chapter 1. To satisfy this demand, I restrict that reference always coerce to the descriptor which is multiple of 4.

2.2 Coercing descriptor to reference

In this section, I discuss about the operation on the coercion from a descriptor to a reference. To coerce a descriptor to reference, `CHECKCAST` instruction is needed as same as to coerce some reference to reference of its superclass. `CHECKCAST` instruction takes class type as an instruction operand and check whether the reference at the stack top points an object of some subclass of specified class. The verifier in JVM assumes the stack-top reference has a reference type to specified class after `CHECKCAST` instruction. I extend `CHECKCAST` instruction to accept descriptor as the stack-top value and check whether the descriptor points to valid object and its type is the subclass of specified class.

The `INSTANCEOF` instruction is similar to `CHECKCAST`. This instruction is used to implement `instanceof` operator in Java language. `INSTANCEOF` checks the dynamic type of stack-top value just like CHECKCAST, and replace stack-top reference with the boolean value which shows whether the coercion is available of not.[1] I also extend INSTANCEOF instruction to handle descriptor.

Many existing JVM assume that references are either null or pointing valid Java object. With this assumption runtime type check can be simply done by checking the pointer to virtual function table contained in the object. This method, however, cannot be applied to descriptor because the assumption is false for descriptor. Descriptors must be checked first whether they point to valid object.

Because this additional existence check works properly even when it is applied to references, implementations applying this check to all CHECKCAST instructions is still proper. However, in those implementations execution performance is lost. To keep performance to existing Java code, this check should be omitted for values which are known to be references.

We can use static type information available from verifier stage to solve this problem. If the stack-top operand of some CHECKCAST instructions have descriptor as its static type, verifier marks these instructions to notify that existence check is needed. Just-in-time code generator (JIT) or bytecode interpreter can change the operation of CHECKCAST instructions which is marked by the verifier. One way to do this is to add new virtual instruction CHECKCAST_DESC internally. This instruction is never appear in bytecode, and only generated through verification stage. The bytecode verifier replaces CHECKCAST instructions with this instruction is it is applied to descriptor. This simplifies JIT and interpreters, and also needs no additional memory space. However, this functionality is not yet implemented because the verifier in Kaffe 1.0.b3 is incomplete and cannot deriver informations needed for this technique.

2.3 Representation of descriptors on Java bytecode

With extensions already described in past section, descriptors can be used inside one function. However, to use descriptor effectively, it should be usable beyond the function regions; it should be used for function arguments and return values; and it should be stored inside other data structures. To achieve this, descriptor type must have some representation on the Java bytecode syntax. This representation should:

1. be distinguishable from either integers and references, to diminish dynamic existence checks.
2. reflect the one-way convertibleness between descriptors and references
3. reflect that memory access through descriptor is invalid

[1]These two instruction differ in the handling of null reference, but this difference is not essential for this extension.

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

Figure 2.2: Helper functions to handle descriptors

I define a virtual superclass of `java.lang.Object` class as the representation of descriptor. This is named `java.lang.Descriptor` and has only two static member functions described later. `Java.lang.Object` is changed to be a subclass of `java.lang.Descriptor`.

Because this representation reflect the convertibleness between descriptors and references, those coercion can be expressed naturally on Java language. Existing Java compiler can detect the location where the `CHECKCAST` instructions are needed. However, with this representation coercion between descriptors and integers cannot be written by existing Java language. To solve this problem, two helper functions in figure 2.2 are defined. These are effectively a simple identity function but changes its type between descriptor and integer for Java language.

This representation has one more important property. This virtual class has no instance functions and instance variables. Because of this, invalid code which access memory through descriptors is never produced from Java language compilers. This makes those invalid memory access detectable at compiler stage, to make debug easier.

There is no special representation for descriptor constants, both in the program and in the initial value of variables. For program constants, descriptor can be represented simply by integer constants or reference constants. For initial values, In JVM they are represented as program in special initializer function and fall into former case.

In current implementation, I use `java.lang.Object` as descriptor representation to make changes to Kaffe smaller. This makes some unnecessary problem which is described in later section.

2.4 Garbage Collector

Garbage Collector (GC) walks and marks all active references in JVM recursively to find all object not pointed by any reference and collect their memory regions for later reuse. Handling of descriptor on GC is ideally desired that object pointed by descriptor is marked if the descriptor currently holds reference, not an integer. However, because descriptor can hold any integers, it is unable to decide which a descriptor currently holds reference or integer perfectly. In fact, not collecting memory region not pointed by any reference is non-desirable but acceptable, though collecting memory region pointed by reference is unacceptable and non-desirable. Therefore, any object which seems to be pointed by descriptor is temporally marked and not collected. This is called *conservative GC* technique.

In general, changing normal GC into conservative GC is difficult, because it is hard to know whether some memory space is object or not. However, in the case of JVM, conservative GC is generally needed even without this extension. This is caused by the properties which JVM's operand stacks and local variable frame have. Each of these locations may be shared by more than two variables with different types, provided that type inconsistency is not caused under all control flow. Types of those are not determinable even if the function currently executing is known, and therefore conservative GC is required. This is the one of the reason that I suppose conservative GC is acceptable.

Some GC routines performs more complex operation to achieve better performance. Because GC have information about all alive objects and references, when an object is moved, GC can repair all reference to the moved object properly. This can be used to gather unallocated memory regions to one continuous area and to improve memory allocation performance. Copying GC is one of the algorithms which uses this property. However, if some object is referenced by descriptor, object cannot be moved. Otherwise, if referencing descriptor is used as reference in future, this value must be rewritten to point new location, but if the descriptor is used as integer in future, the value must not rewritten anyway. These two situations are not determinable from GC and the operation is exclusive. To conclude, an object which is referenced from at least one descriptor must not moved from existing location. Researches for copying GC with those untrusted references are mentioned in other places[1]. In this thesis, this problem is not mentioned any more because the GC routine in Kaffe is Mark & Sweep method and does not move object anyway.

2.5 Array handling

On JVM, array of references have transitional properties as same as those of simple references in the sense of coercion. For example shown in figure 2.3, reference to an array of `String`s can be stored to a variable whose type is an array of `Object`s. This makes that reference to the object of some type cannot simply stored into the a array which is pointed by a reference to an array of the that class. In the above example, if an reference to `Integer` is stored into a array through the reference to an array of `Object`s which actually points an array of `String`s, and if the array is also pointed by another reference whose type is an array to `String`s, user can later get the reference stored as an reference to `String`, which is a violation of type restriction. To prevent this to happen, JVM dynamically check the type of reference against the dynamic type of the array in store time.

If the system forbids assignments of a reference to an array of descriptors into an variable with type of array of descriptor, invalid reference will never created with

```
{
  Integer ip = new Integer(0);
  String[] as = new String[1];
  Object[] ao = as;              This is valid
  ao[0] = ip;                    If this is allowed,
  String s = as[0];              this get an reference ip as String
}
```

Figure 2.3: Example code which needs dynamic type check of array

array handling. This solution is restrictive, but simple and complete. Otherwise, system can allow such assignments with object existence check at runtime. In this approach, existence check is required when both two conditions are met:

1. static type of a reference to the array is array of descriptor, in according with type inducted at verifier stage.
2. dynamic type of the array, to which value is to stored, is an array of object.

Although second condition is only determinable on runtime, first one is fully determinable at verifier stage. Therefore, verifier can limit this check to the instructions which meet the first conditions, in the way same as `CHECKCAST` and `INSTANCEOF` instructions.

2.6 Safety

2.6.1 About type safety

Type safety of references is very important to guarantee the safety of JVM. JVM assumes that any codes which passes the verifier does not make any reference which does not point to any object, and that any object pointed by the reference is an instance of some sub-class of the reference's type. If this assumption fails, safety is completely compromised and user may perform any operation which is invalid to do. To proof the type safety of references before execution, JVM must check following conditions statically.

1. For each instructions, depth of the stack is always same, independent from execution path.
2. For each instructions, the types of operands on the stack must be subtype of the types of the instructions accepts.
3. All (static/dynamic) member of some class must typed statically to one type.

4. For each local locations such as stack or local variables, it stored value is used at some instruction as some type, it must be stored as some subtype of the type in all execution path which reaches the instruction.

If these conditions are met, all operands of all instruction in the bytecode properly have the type which the instruction accept.

The descriptor extension breaks the separation between integers and references which the original JVM assumes. Therefore, I extend the type system for the descriptors and assure that all references are either (1) proved to be valid reference, in the way the verifier of original JVM does; or (2) constructed from descriptor and its validity is checked explicitly with `CHECKCAST` instruction.

2.6.2 Higher order security

Although type safety is most important part of JVM's security, not all security can be saved with it. There is aspects of security such as information privacy, which is more higher aspects than type safety. This extension assures the proper execution of bytecode and keeps the world outside JVM safe from attack, but data which is inside JVM is not secured. User can guess the address of the secret object as integer and cast it to reference through descriptor. Also, any class object may be acquired in the same way, so that some secret class may be accessible through Reflection API of JVM 1.1. This problem is exist when more than one programs are running on same Virtual Machine (including system tasks implemented in Java or JVM language), for example on AWT, applets, and mobile agents.

To solve this, I guess that some access managements is needed. Further detailed work is needed for this problem, but I propose one possible solution. We can partitioning each program apart into each "execution region", and add the identification of execution region to all objects it allocates. When some program is to coerce descriptor to reference, matching of identification is checked in addition to existence check and type check. In my opinion, this check is not needed for other instructions, since ordinary operations on references are already safe to this attack.

2.7 Required modification to implement descriptor extension

One property for extension which I regard as important is ease of implementation. This extension is carefully designed to keep modification to existing system smaller and easier. Required work for implementing this extension on usual existing JVM is as follows:

1. implement some object identify mechanism for existence check (see section 2.2)

2. modify GC to handle descriptor correctly (see section 2.4)
3. modify `CHECKCAST` instruction handler to check object existence
4. modify array store handler to accept descriptor as an operand
5. add primitive `Descriptor` type and support methods (see section 2.3)
6. modify verifier to analyze data flow of descriptors

I suppose these modifications are not too hard to implement on any existing JVM systems.

Chapter 3

Implementing descriptor extension

In this chapter I express the implementation for the descriptor extension. I implemented subset of the extension on Kaffe 1.0.b3.

3.1 Implemented features

In my implementation, because verifier of Kaffe 1.0.b3 is incomplete, I have not modified the verifier.[1] In addition to this, in spite of the discussion in section 2.3, to minimize the modification to Kaffe and Kawa Scheme in test phase, current implementation uses existing `java.lang.Object` class as the representation of descriptors. This makes following additional changes essential:

1. Because virtual method invocation through the reference of type `java.lang.Object`, additional runtime check of object existence is needed to be inserted before those invocations by JIT.

2. Special handling for the array of `java.lang.Object` is needed.

3. All native methods which takes `Java.lang.Object` as arguments needed to be modified.

4. Checks for object existence is needed for value of frequently-used `java.lang.Object` at GC and `CHECKCAST`.

Especially, 3. makes modification for the system very huge. Currently, I bypassed such methods temporally, but for type safeness this can not be neglected. In addition to this, existence check introduced by 1. and 4. slows execution of existing Java code, which is

[1]Of course, this shows that Kaffe 1.0.b3 is not type-safe.

shown later. Knowledge about these problems are acquired through experiment, and to resolve these, I decided to implement complete part of the extension described in this thesis in near future.

3.2 Object management and garbage collector

To implement the extension, system firstly must be able to know whether pointer is referring some object or not, and it is difficult for general languages. However, as described in the section 2.4, it is likely for JVM to have the way to perform similar tests, and therefore this requirement is not too unrealistic. In Kaffe, memory management routines keep track of the address and usage of all allocated memory including objects, and system can determine the usage of memory blocks only from memory location address. In my implementation object existence check is performed in three step:

1. Check whether reference points the top of the some allocated memory block actually

2. Check the memory block found is currently marked in use

3. Check recored usage of the memory block is one of four types which is correct Java object type, out of eighteen

First two step is already implemented for conservative GC, and only the last step is implemented additionally. If and only if these three conditions are met, some object is at the location pointed by the reference.

Secondly, garbage collector must be able to handle descriptor like conservative GC. Because of the reason described in section 2.4, Kaffe's original GC routine walks Java operand stack with conservative GC technique, and therefore no change is required for descriptor extension and operand stack. On the other hand, Kaffe does not perform conservative GC on class instances. Once the type of the object is determined from virtual function table pointer, locations where references are exist in the object memory space is fixed. Kaffe keeps track of this information in the class information structures. During class loading, Kaffe copies the bitmap from the class information of its superclass and construct it for the additional members which is not inherited from superclass.

With my extension, conservative GC is needed for descriptor member. I created second bitmap in class information, which records whether each member is descriptor or not. Combined with first bitmap, these bitmap keeps 2 bits of information for each member. Descriptor member have 1 for corresponding position of both bitmap, and Reference have 1 in only original bitmap. Immediate members such as integers and booleans have 0 in both. Action of the garbage collector is defined as follows:

(0-0): value is a immediate GC ignores this value.

(0-1): value is a reference GC marks and traces the object referred by this reference without any dynamic checking.

(1-1): value is a descriptor GC first checks object existence and marks only if object is exist there.

3.3 Modified instructions

Modification for JIT is needed for four instructions. First two is `CHECKCAST` and `INSTANCEOF`. These two functions are implemented as C routine in Kaffe and JIT emits function call in output native code for these instruction. Therefore, modification for these instructions are done at C-language level. I added object existence check for the routine of both instructions. If check fails, `INSTANCEOF` simply returns 0 and return to JIT-generated code. For the failure case of `CHECKCAST` instruction, I generate special diagnostic message and throws new `ClassCastException` object.

Next one is for `AASTORE`. This instruction stores reference (or descriptor) into array, and implemented in C language also. In my extended implementation, class check is bypassed when dynamic type of the array is `Object`. Because class check required by JVM specification is done by using same routine as `INSTANCEOF`, modification for object existence check is already done.

Last one is for `INVOKEVIRTUAL`. This instruction is processed by JIT level and no corresponding C language routine which is called in execution time is not exist. Extended implementation inserts native code for `CHECKCAST` instruction in JIT time if called method is of `Object` class.

3.4 Modification for Java library

Support functions described in section 2.3 is added. In this implementation these routines are added to class `java.lang.Object`. I added the prototype definition for the source code of Java library and compiled it into `.class` file. Substance for these method is in dynamically-linked library (DLL) and linked dynamically by name identification. I added small two identity function to DLL.

Chapter 4

Implementing Scheme with descriptor extension

Kawa Scheme is the almost-full-featured Scheme implementation on the JVM. Kawa have closure compiler and Scheme closures are compiled into Java bytecode at definition time to provide fast execution. The compiler in Kawa produces JVM instructions directly, which is fully JVM specification compliant. I extend this system with descriptor extension and solved the problem mentioned in Section 1.3.

4.1 Representation of Scheme value

In my new implementation, Scheme's values are represented as descriptor. Signed integers which can be represented by 30bit are called fixnum and represented as descriptor coerces from integer. Its encoding is $4n + 1$ for integer n, which is always odd number. Other objects such as cons cells, vectors, big numbers etc. are represented as boxed objects (as same as original Kawa Scheme) and handled by the descriptors coerced from the references. By the encoding of fixnums and the rule described in section 2.1, those two groups of descriptors are easily determinable by checking least significant two bits of the descriptor.

4.2 Modification for procedure interface

In Kawa scheme, all procedures are represented as an instance which is subclass of `kawa.lang.Procedure`. Most procedure have corresponding class in which actual job is defined as method. JVM does not have first-class functions, but this technique emulates the first-class functions in small cost and with extendibility. The class `kawa.lang.Procedure` have six abstract method, from `apply0` to `apply4` for invocation with each 0 to 4 arguments, and `applyN` for call with any number of arguments.

Compiled code calls specific routine for pre-determined number of arguments for efficiency, and interpreter calls `applyN` for generality. To make implementation easier, some glue classes are defined and mediation between those functions are done automatically. For example, `kawa.lang.Procedure2` class have only one abstract method `apply2`, and `apply{0,1,3,4}` are defined to generate runtime error. `applyN` is defined to first check number of the arguments and then calls `apply2` only if just 2 arguments are given. All two-argument procedures are defined by inheriting `Procedure2` and defines own `apply2` routines.

I extend this interface to keep interchangeability in some extent and use descriptors effectively. I first defined helper function which converts descriptor-expressed integer into original-compatible boxed object. I then defined more six interfaces `dapply`*X* (*X* is one of `{0,1,2,3,4,N}`) in `Procedure` class. In `Procedure` class these methods are defined that they converts all arguments with the helper function and the calls corresponding `apply`*X* with converted arguments. Results are back-converted to descriptor representation. Then I define new interface `DProcedure` (Descriptor Procedure) which is subclass of `Procedure`. It has `dapply`*X* as abstract member and all `apply`*X* are defined to call corresponding `dapply`*X*. `DProcedure`*X* is also defined as analog to `Procedure`*X*.

Because descriptors and references share almost same syntactical properties except for member access, All methods which does not care about the numeric arguments are not needed to be re-implemented. For example, the Scheme function `car` simply casts the arguments to `Pair` class and take `.car` member from it. Because casting non-reference descriptor to any class always fails, original routine also works with descriptor representation. So, all changes needed are change its superclass from `Procedure1` to `DProcedure1`. Also, numerical functions which is not used frequently at least with integer arguments are not changed. They pass arguments through the bridge routine defined in `Procedure` class and perform operations as same as original Kawa Scheme. Some I/O procedures such as `read` and `display` are also modified to handle fixnum correctly.

Finally, frequently-used numeric operations are re-implemented with descriptor extension. These operations are defined as the sequence of routines which emits Java bytecode and inlined into body of the function which calls such procedures. The inlined routine first check whether the given descriptor is fixnum and handles both type of values appropriately. If both arguments are fixnum, calculation is done directly. For example, adding two integers represented by descriptors can be performed by adding two fixnum-descriptors and decrement it by 1. If the calculation overflows or one or more arguments are not fixnum, the operation is performed by a support routine which is implemented by Java language. It is defined as static method in class of each procedure and handles both fixnums and references correctly but slowly. Example are shown in Appendix A.

4.3 Changes for compiler, etc.

Function compiler also need some modification. Firstly, it changed to emit call to `dapply`*X*, to use extended interface. Next, handling of constants in the program are changed. Because fixnum constants cannot be used with Java's standard methods for `Object`, it must be handled specially. This change needed investigation on the structure of Kawa's compiler, but modification is still not so large. Also, compiler for bridging routine to native Java class is also needed to be changed.

Other changes required for Kawa to work is on following areas:

- Initial environment and procedure class autoloader: to clarify function's ability to be inlined before loading
- Bytecode handling routines: because of incompleteness of original routine

Chapter 5

Experiments

5.1 Performance on Scheme code

To evaluate performance of modified Kawa Scheme, I measured the execution time of two integer calculation. One is the Fibonacci function fib(25), which defined as follows.

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

The other is the decimal 1000 decimal places of the value π, calculated by every 4 digits with the program contained in `scm` distribution. The evaluation is performed on Sun Ultra Enterprise 4000 (UltraSPARC 168MHz) and Solaris 2.5.1. Kaffe normally performs stack-overflow checking on runtime, but it is disabled to evaluate pure calculation and garbage-collection time.

Table 5.1 shows that this extension improves the performance of integer operation with Scheme code on the large scale. For both example, execution speed is about twenty times as the original implementation.

5.2 Performance on existing Java code

To test the performance on existing Java code, I chose five test from benchmark test suite “Spec JVM98”[9]. Other tests are omitted because they are unfortunately not properly runnable on Kaffe. For each test, execution time is measured five times and the total time is compared between the original JVM and the modified one. Table 5.2 shows the result of the evaluation. Environment used to evaluation is the same as the test of the previous section.

test program	original	modified	ratio
Fibonacci	53	2.9	−94.5%
Pi	1029	42	−95.9%

(unit: sec.)

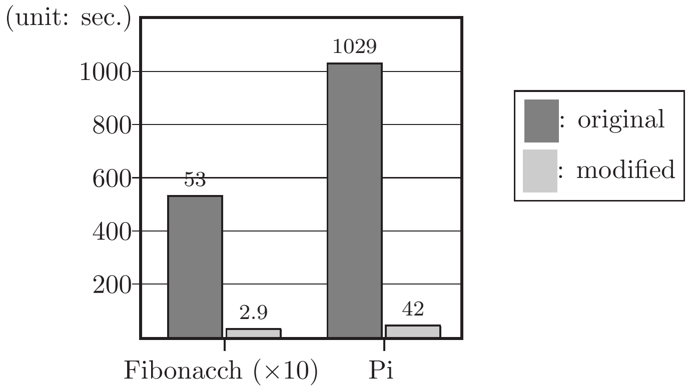
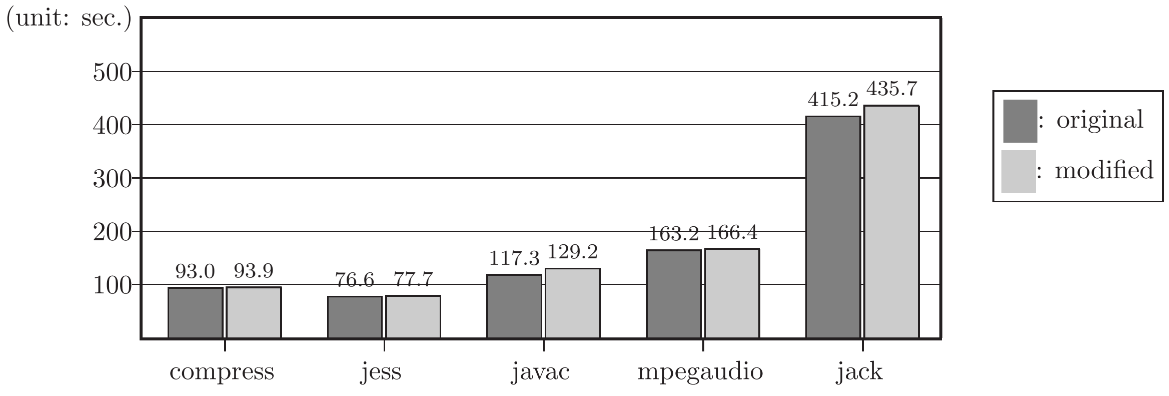


Table 5.1: Performance improvement of Kawa Scheme with the descriptor extension

The penalty shown in the table 5.2 is supposed not to be a substantial problem of the extension but to be the implementation problem described in section 3. It is caused by the unessential runtime check for object existence with invoking method defined in `java.lang.Object`, and with applying the `CHECKCAST/INSTANCEOF` instructions to references. The variation of the performance penalty is appeared because the frequency of the instructions which cause those unessential check vary dramatically among those programs, along with the properties of the programs. Those penalty might be removed with the importation of new descriptor class, with implementation of verifier and with optimization using static type informations.

test program	original	modified	ratio
201 compress	93.013	93.921	+1.0%
202 jess	76.559	77.717	+21.1%
213 javac	117.308	129.222	+10.2%
222 mpegaudio	163.226	166.376	+1.9%
228 jack	415.189	435.654	+4.9%

(unit: sec.)



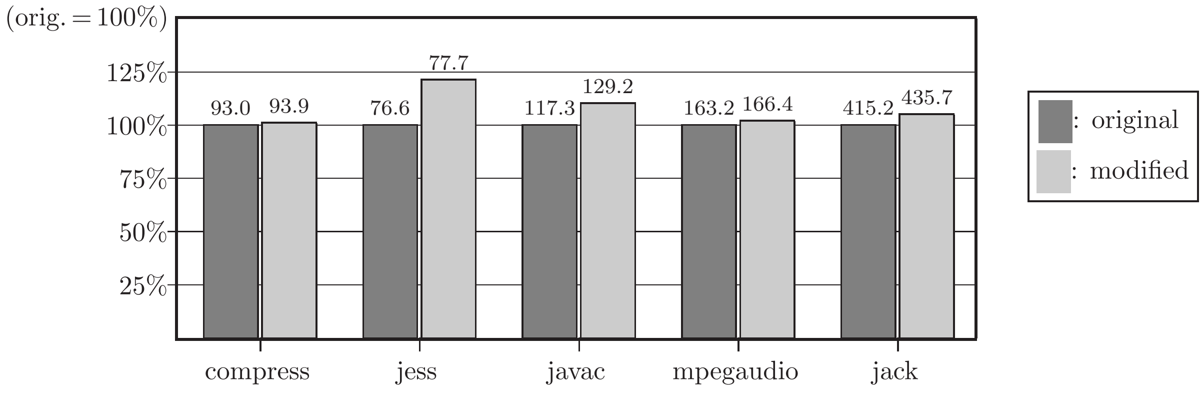


Table 5.2: Performance penalty for existing Java code with the descriptor extension

Chapter 6

Related work

Olin Shivers [8] describes his `DirectDescriptor` extension to the same means very briefly. His extension rules all references to have odd numbers as native representation. Also, `DirectDescriptor`, which is virtually a subclass of `Object`, holds 31bit of state and expressed by even numbers. Integers are converted to `DirectDescriptor` by one bit shift, and `DirectDescriptor` to integers by compile time change of view. Superior point of his extention is that higher order security mentioned in Section 2.6.2 is kept. This is because unsecure reference is never generated by those restrictions. However, even the discussion in his note is very brief, I suppose that the modification for JVM which is needed to implement the extension is large, and also performance penalty for existing code is difficult to eliminate on recent CPU architectures.

Andrew W. Appel and David R. Hanson [1] describes one method for implementing copying garbage collection with ambiguous reference (i.e. descriptor) briefly. They assume that object is self-identify, so that existence check is done by small cost. Applying copying garbage collection for Kaffe implementation is under experimentation by Tanaka [10], concurrently with my experiment.

Chapter 7

Conclusion and future work

I extended JVM to handle new type named *descriptor* which can be handled as both integers and references, without breaking JVM's type safety, to support dynamically-typed languages efficiently on the virtual machine. I implemented the extension to existing virtual machine, and I also implemented Scheme system on it and evaluated the speed of calculation. Usage of descriptor makes integer operation, which is one of the most frequently-used operation in Scheme language, about twenty times faster. I also constructed the frame work to make this extension accepts existing Java code without any performance penalty.

Further work is needed to implement the verifier which proofs the type safety of this virtual machine. More performance improvements are available with the information generated by the verifier. In addition to this, more security-related features might be needed to use this extension with network applications such as applets and agents.

References

[1] Andrew W. Appel and David R. Hanson. Copying garbage collection in the presence of ambiguous references. Research Report CS-TR-162-88, Department of Computer Science, Princeton University. June 1988.

[2] Nick Benton et. al. Compiling standard ML to Java bytecode. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. January 1999.

[3] Per Bothner. Kawa: the Java-based Scheme system. in *Lisp Users Conference.* November, 1988. Available from `http://www.cygnus.com/~bothner/kawa/` .

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

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

[6] Richard Kelsey, William Clinger, and Jonathan Rees (editors). Revised^5 report on the algorithmic language Scheme. February 1998.

[7] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification. Addison-Wesley, 1996.

[8] Olin Shivers. Supporting dynamic languages on the Java virtual machine. April 1996. Available from `http://www.ai.mit.edu/~shivers/javaScheme.html`.

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

[10] Yoshizumi Tanaka. Copying garbage collection in the presence of uncertain pointers. Senior Thesis. University of Tokyo. February 1999 (to be published).

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

Appendix A

Example output code produced by extended Kawa compiler

This is output of the compiler with the following simple Scheme code.

```
(define (f x) (+ x 5))
```

Push the argument x and 5 onto stack. Integer 21 in instruction 1 is tagged value of fixnum 5.

```
  0: aload_1
  1: bipush 21
```

Check tag of the arguments on stack before calculation. Value is fixnum if v `and` $3 = 1$. If one of the arguments are not fixnum, jump to instruction 50.

```
  3: dup2
  4: iconst_3
  5: iand
  6: iconst_1
  7: if_icmpne 50
 10: dup
 11: iconst_3
 12: iand
 13: iconst_1
 14: if_icmpne 50
```

Branch according to the sign of the first operand, for overflow check.

```
 17: iflt 35
```

The routine for positive operand. Add operands and subtract 1.

```
20: dup2
21: dup_x1
22: iadd
23: iconst_1
24: isub
25: dup_x1
```

If result is smaller than second operand, overflow is occurred. Then jump to instruction 50.

```
26: if_icmpgt 50
```

Throw away dust on the stack and go to instruction 54.

```
29: dup_x2
30: pop2
31: pop
32: goto 54
```

The routine for negative operand.

```
35: dup2
36: dup_x1
37: iadd
38: iconst_1
39: isub
40: dup_x1
41: if_icmplt 50
44: dup_x2
45: pop2
46: pop
47: goto 54
```

If type check is failed or overflow is occurred, control reaches here. Call Java-implemented routine to handle.

```
50: pop
51: invokestatic #20=<Method kawa.standard.plus_oper.addTwo
        (java.lang.Object java.lang.Object) java.lang.Object>
```

The final instruction returns result to caller.

```
54: areturn
```