IPSJ Transactions on Programming Vol.6 No.1 9–16 (Jan. 2013)

Regular Paper

Just-in-time Compiler for KonohaScript Using LLVM

Masahiro Ide[1,a)] Kimio Kuramitsu[1,†1,b)]

Received: April 20, 2012, Accepted: September 17, 2012

Abstract: In recent years, as a method to improve the language performance of scripting languages has attracted the attention of the Just-In-Time (JIT) compilation techniques for scripting language. The difficulty of JIT compilation for scripting language is its dynamically typed code and in its own language runtime. The purpose of this paper is to evaluate the performance overhead of JIT compilation of runtime library's overhead by using a statically typed scripting language. In this study, we use a statically typed scripting language KonohaScript to analyze language runtime performance impact of the code generated by the JIT compiler.

Keywords: just-in-time compiler, scripting language

1. Introduction

In the past decade, significant advances have been seen in scripting language just-in-time (JIT) compilers for JavaScript [1], [2], as well as for other languages such as Python [3]. JIT compilers can dramatically improve performance by converting application code into machine code in the execution time. Unlike a simple interpreter for scripting language, JIT compiled code is executed directly on the CPU.

The initial implementation of a scripting language JIT compiler is to typically generate native code by translating a higher-level bytecode into native code naively, where applications run slower than equivalent implementations in C and Java. This translation is straightforward, but it leads to the following two overheads being introduced into a scripting language. First, the scripting language JIT compiler needs to take into account the dynamic typed code. Second, this JIT compiler needs to perform code generation with a consideration of collaboration and with runtime for its own language.

Relevance for dynamic typing and performance of code that the JIT compiler generates is well documented by Chang et al. [4]. They evaluate JIT compilers by measuring their performance changes for each full-typed, partially-typed, and un-typed piece of code. As a result, they report that speed improvements of about 60 percent, on average, are gained by adding type information in the benchmark codes.

In this paper, we evaluate an overhead on JIT compiled code except where this is caused by dynamically typing on that interpreted source code. To achieve this goal, we constructed a JIT compiler for a statically typed scripting language, KonohaScript. Furthermore, we describe the design of a JIT compiler for KonohaScript based on a Low Level Virtual Machine (LLVM [5]) as the compiler backend. In addition, we identify three main performance-inhibiting issues in the naive translation of the code in order for it to work together with the scripting language runtime. Finally, we evaluate the execution performance by using a benchmark application with our JIT compiler.

This paper is organized as follows: In Section 2 we give an overview of scripting languages. Section 3 provides an overview of our JIT compiler, followed by a description of our optimization techniques in Section 4. Then, Section 5 evaluates the effectiveness of our JIT compiler and related work is discussed in Section 6. Section 7 concludes this paper.

2. Scripting Language Runtime

In this section, we define the word Scripting Language. We describe the features provided by the runtime system of scripting languages such as Ruby, Python, and Lua. In addition, this section describes the design of KonohaScript and the relevant parts to our JIT compile.

2.1 Definition of Scripting Language

Scripting language is not clearly defined scientifically, but one of the common features of scripting languages is its dynamic type. The classification of the language to perform as a dynamically typed language is included with Lisp and Scheme, but these are not called scripting languages. In recent years, several attempts have been made to integrate static typing features into existing dynamic languages such as ActionScript and Thorn. Another feature for scripting language is an interpreter. Scripting languages require source code (or scripts) at runtime, which need to load and execute scripts.

In recent years, in an effort to improve their performance, scripting languages were often accompanied by a JIT or bytecode compiler. Therefore, a scripting language is not able to be classified simply by the presence of the compiler. The main difference between compiled languages (e.g., C and Java) and scripting languages is that scripting languages can load a script and execute

[1] Graduate School of Yokohama National University, Yokohama, Kanagawa 240–8501, Japan
[†1] Presently with Japan Science and Technology Agency/CREST
[a)] ide@konohascript.org
[b)] kimio@ynu.ac.jp

it at runtime. In this paper, we define a scripting language as a program processing system that has an interpretation engine and an execution engine, which are both integral parts of the system.

2.2 Scripting Language Runtime

Scripting languages are composed from the following four components: an execution engine, a compiler, an extension library, and a memory management system.

Execution Engine and Compiler

The runtime library for a scripting language has an interpreter that is a mechanism to convert a script to its own format in order to execute a script. Execution of a script is carried out to coordinate the execution engine and run-time libraries. In recent years, scripting languages have often provided a bytecode interpreter and JIT compiler to improve performance.

Extension Library

Script language runtime can be enhanced by a library written in C and a function provided by a script. The scripting language user can then use a function not only written in the scripting language but also those provided by the library extension. This runtime has two variations of function extended by the user. One is written in C and is compiled before execution, while the other is written in a script and is compiled as bytecode. From a user's perspective, they can treat these functions equally.

Each function provided by the runtime library has a calling convention to make calls to each other. Furthermore, the scripting language stack structure is used to transfer data both to and from the execution engine and run-time libraries. There are two main methods used when implementing a stack structure. One uses the native stack and the other uses its own stack that is allocated on the heap.

Memory Management

Memory management is one of the most difficult parts when writing a program correctly and efficiently. In order to manage the memory, the scripting language runtime makes use of the garbage collection (GC). When the runtime library allocates new data, the runtime library uses the memory region managed by the GC. Then the runtime library modifies these memory regions based on its own memory layout policy.

The memory layout of the data structure consists of two main memory regions, the header region and the body region. In the header region there are both object type information and flags to indicate the characteristics of each object. For example, the runtime uses the type information for type checking, while the runtime uses the body region to store the actual data.

2.3 KonohaScript Runtime System

This section describes the runtime library of KonohaScript in relation to the following sections. KonohaScript [6] is a statically typed object-oriented scripting language that has been developed from scratch in C. More information about the language design of KonohaScript can be found in Ref. [7]. KonohaScript has adopted a class-based, object-oriented model. This means that the behavior of objects such as methods and fields are defined in the class. In KonohaScript the user can provide attributes for classes and methods, and the following is an attribute that can be granted.

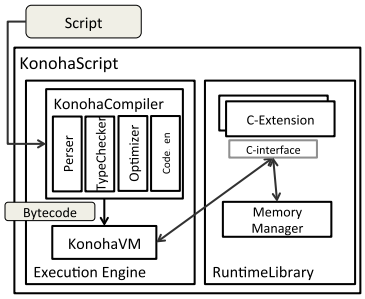

Fig. 1 Architecture of KonohaScript.

- @Final: Class with no inheritable
- @Singleton: Class is guaranteed to be a single instance
- @Native: Method is implemented in C
- @Static: Static method

Figure 1 shows the elements that make up the KonohaScript runtimes.

Execution Engine and Compiler

Based on the compile-time type information included in an annotation, KonohaScript implements the well-typed bytecode and a register-based bytecode interpreter (KonohaVM).

KonohaVM executes the source code in the following order. First, the KonohaScript compiler compiles the source code into bytecode as a single compilation unit. Next, KonohaVM evaluates the different methods in the order to which they are described in a script. Finally, KonohaVM executes the main method.

Extension Library

In KonohaScript, as an interface to calling the extended library functions, all methods are defined in Fmethod type that is defined in C.

```
void (*Fmethod)(CTX ctx, ksfp_t *sfp, long rix)
```

`ctx` (Context) is a pointer to the run-time management information and `sfp` indicates a pointer to the top of the call stack (KonohaStack) that is managed by the runtime of KonohaScript. In addition, `rix` indicates the index on the stack where the return value of the method is to be placed. If the C function matches this interface then the KonohaScript runtime can call this function from a script.

KonohaStack is used to pass data between the methods, such as arguments, and then return values of the method. When KonohaScript's runtime invokes a method, the caller will push the arguments onto the stack pointer in order to call the target method.

With static typing, KonohaScript boxes and unboxes primitive values automatically; such values include integer, float and boolean. Values of primitive types are handled as unboxed values, unless it is necessary to treat them as objects. The data structure of KonohaStack is adopted as being clearly distinguishable between object reference and primitive. The following code is the data structure of KonohaStack.

```
struct ksfp {
  union {int i;float f;} data;
  kObject *optr;
}
```

Memory Management

The memory management system of KonohaScript provides automatic memory management using GC. In 64-bit architecture,

```
typedef struct {
  struct hObject {
    ClassInfo *cinfo; // type infomation
    void *gcinfo; // used by GC
    long flags;
    void *metadata;
  } h; // object header
  struct kObject **fields; //Object body
} kObject;
```

Fig. 2 Memory layout of Object.

each object is aligned in 64 byte and has a memory layout as shown in **Fig. 2**. The header region of the object is composed of a flag which indicates the object state, information type, and a field that GC is used.

3. Design of KonohaLLVMJIT Compiler

In this section, we describe the design of the KonohaLLVMJIT compiler. LLVMJIT is a JIT compiler that uses LLVM as a compiler backend. LLVMJIT receives an intermediate representation of a script and converts it to LLVM Intermediate Representation (LLVM IR). We describe how to generate LLVM IR in LLVMJIT and devised a point at which we convert scripts to LLVM IR.

3.1 Konoha-LLVMJIT compiler

In order to achieve the generation of highly efficient code, we designed a new intermediate representation KonohaIR and converted KonohaIR to LLVM Intermediate Representation (LLVM IR). With the LLVM backend compiler optimizations such as common sub expression elimination, LLVMJIT finally generated a machine code.

KonohaIR is an intermediate language for the purpose of generating efficient code using LLVM compiler optimizations, and it has made the design of instruction much easier based on the intermediate representation of both the LLVMIR and KonohaVM bytecode.

KonohaIR consists of two elements, expression and annotation. We designed the expression based on a representation of the static single assignment (SSA) form. The operation of each expression has multiple inputs and a single output. The expression and its operand have been given type information, and the LLVMJIT provides higher-level information to generate better code such as a constant value by annotations. Annotations are used and then converted to LLVM IR and used in the LLVM backhand optimization phase.

3.2 LLVM IR Code Generation

In this section, we describe the process of LLVM IR code generation from KonohaIR. All methods are converted to KonohaIR and then converted to LLVM IR by LLVMJIT. The instruction sets of the LLVM IR and KonohaIR take a one-to-one correspondence. LLVM IR is converted to produce the corresponding instructions from KonohaIR. For example, KonohaIR's integer "add" instruction supports the "add" instruction of LLVM IR.

LLVM supports the aggressive optimization of a universal language, yet KonohaScript supports high-level language features such as garbage collection. Both LLVM IR and KonohaIR provide a mechanism for the representation of data structures and function calls. As it could not be expressed in a one-to-one correspondence between LLVM IR and KonohaIR, we provided a mechanism to convert KonohaIR to LLVM IR individually. In the following sections we will compare each type system and function call mechanism, and describe how to convert each correspondence.

```
int f(int n) {
    int v = 0;
    for (int i = 0; i < n; i++)
        v += i;
    return v;
}
```

Fig. 3 An example KonohaScript program.

3.2.1 LLVM Types

LLVM type systems were originally similar to C, supporting boolean (i1), char (i8), long (i32), and float. Now its type system supports structs and arrays.

```
i1, i2, i8, i16, i32, i64
float, double, fp80
array = type [3 x i64] /* array type */
struct = type {i64, float} /*struct type */
```

KonohaScript has three primitive types, boolean, integer, and float. On LLVM IR, these primitive types are made to correspond to the type bit length matches. In order to represent type conversion in LLVM IR, we also built the data structure of the object that is compatible with that used in runtime, and LLVMJIT inserts the LLVM IR express type conversion.

3.2.2 LLVM Functions

In LLVM IR, a function is called based on the calling convention of the architecture. On the other hand, in KonohaScript all methods are defined in a manner tailored to Fmethod type and arguments, and return values are then passed through KonohaStack before and after the method call. When LLVMJIT converts the methods to LLVM IR, we generate the LLVM IR function by adjusting to the Fmethod interface. In addition, LLVMJIT inserts the instruction that is stored in the return value to KonohaStack, and then adds a process to load arguments from the KonohaStack when converting LLVMIR from KonohaIR.

3.3 Sample LLVMJIT Execution

In this section, we describe the compilation flow of LLVMJIT using an example. **Figure 3** shows the source code used in the following description.

At first, LLVMJIT translates the KonohaScript code in Fig. 3 into KonohaIR in **Fig. 4**. At this time, each block of a script is converted to basic blocks and each part of the code is translated to SSA form. When LLVMJIT converts code to SSA form, to be unique of usage and declaration of a variable, LLVMJIT inserts the PHI instruction. The PHI instruction is represented by a list of pairs of variables and basic blocks. Next, LLVMJIT performs the translation from KonohaIR into LLVM IR in **Fig. 5**. With the exception of those mentioned in the previous section, the conversion to LLVM IR is a one-to-one transformation from KonohaIR. Finally, LLVMJIT converts from LLVM IR to a machine code using the LLVM backend.

```
def Int Script.f ( Int i1_0 ) {
bb0:
  i2_0 = @const Int 0
  i3_0 = @const Int 0
  _ = jmp bb1
bb1: @loop:cond
  i1_1 = mov Int i1_0
  i2_1 = phi [(i2_0, bb0), (i4_0, bb3)]
  i3_1 = phi [(i3_0, bb0), (i4_1, bb3)]
  b5_0 = lt Boolean i3_1 i1_1
  _ = cond Boolean b5_0 bb2 bb4
bb2: @loop:body
  i4_0 = add Int i2_1 i3_1
  _ = jmp bb3
bb3:
  i6_0 = @const Int 1
  i4_1 = add Int i3_1 i6_0
  _ = jmp bb1
bb4
  _ = return i2_1
}
```

Fig. 4 The KonohaIR sequence for the program in Fig. 3.

```
define void @f(%ctx* %ctx, %sfp* %sfp, i64 _rix){
bb0:
  %0   = getelementptr %sfp* %1, i32 1, i32 1
  %arg = load i64* %0
  ;; Konoha to load the arguments from the stack
  br label %bb1
bb1:
  %3 = phi i64 [ 0, %bb0 ], [ %6, % bb3 ]
  %4 = phi i64 [ 0, %bb0 ], [ %7, % bb3 ]
  %5 = icmp slt i64 %4, %arg
  br i1 %5, label %bb2, label %bb4
bb2:
  %6 = add i64 %3, %4
  br label % bb3
bb3:
  %7 = add i64 %4, 1
  br label %bb1
bb4:
  %8 = getelementptr %sfp* %1, i32 _rix, i32 1
  store i64 %3, i64* %6
  ;; Make the store of arguments from KonohaStack
  ret void
}
```

Fig. 5 The LLVM IR sequence for the program in Fig. 4.

4. Implementation

In this section, we describe the optimization applied to the LLVMJIT compiler. LLVMJIT optimizes for the following items to be considered in the code, which it generates for the JIT compiler to work with language runtime, and is associated with high execution performance.

- Method interface
- Calling runtime library function
- Memory layout of objects

4.1 Direct Invocation Between Compiled Methods

In KonohaScript, various kinds of methods co-exist simultaneously, such as methods represented with bytecode and executed by an interpreter, native methods written in C and provided by an extension library, and methods compiled by a LLVMJIT compiler. To agree a calling convention to call each other between different kinds of methods, KonohaScript provides a single common calling convention. LLVMJIT generates that code and is responsible for matching the common method interface.

```
/* factorial method */
int factorial(int n) {
  if (n < 2) return 1;
  else return n * factorial(n-1);
}
```

Fig. 6 Factorial program written in KonohaScript.

```
/* Common calling interface version */
void fact(CTX ctx, ksfp_t *sfp, long rix)  {
    int n = sfp[1].ivalue;
    int ret = fact_opt(ctx, n);
    sfp[rix].ivalue = ret;
    }
/* specialized interface version */
int fact_opt(CTX ctx, int x) {
    if (n < 2) return 1;
    return n * fact_opt(ctx, n-1);
 }
```

Fig. 7 The KonohaIR sequence for the program in Fig. 6 with specialized interface.

As described in Section 2, all methods for KonohaScript are defined with the common method interface Fmethod. Therefore, the code of method invocation that LLVMJIT generates is not only a simple function call instruction but also contains the operation of the arguments and return values to KonohaStack. However, in a case where both the caller method and callee method are compiled by LLVMJIT, the common interface does not have to be involved and it can eliminate load and store operations to KonohaStack for arguments and return values. In addition to the common method interface, when both the caller method and callee method are compiled by LLVMJIT, LLVMJIT provides a specialized interface that eliminates the operation of the return value and arguments through KonohaStack, and the compiler generates a call instruction directly.

When LLVMJIT compiles a method it generates code that conforms to two method interfaces, a specialized interface and a common calling interface. If the method is called from a non-compiled script, it makes a call via the common calling interface. On the other hand, the method is called from the compiled code calling convention, switches to a specialized interface, and the methods are called but not allowed to pass through the KonohaStack. As a side note, with dynamic binding the LLVMJIT compiler cannot guarantee that both the callee and caller methods will always be compiled code. Therefore, LLVMJIT is restricted to adapt the ideas when the callee method is a final method or static method so that dynamic binding does not occur.

Figure 6 shows an example of the factorial method written in KonohaScript and JIT compiled code, with a common method interface and specialized interface in **Fig. 7**. We noted that we would rewrite JIT compiled code for implementation in C for readability.

4.2 Inlining Runtime Library

Inlining is a well-known optimization technique to reduce the number and cost of method calls, and it introduces more opportunities of inter-procedural analysis and optimization. In script-

ing languages, there are many functions provided by extension libraries. To adjust the difference between the common interface and calling convention of the native method, wrapping functions are provided in these libraries.

Expanding the scope of JIT compiler optimization by inlining to the runtime library can be expected to lead to a reduction in the number and cost of method calls, including wrapper functions. However, in the existing JIT compiler, it is not easy to inline the native method provided by the runtime library because it is necessary to disassemble the runtime library and analyze the behavior of each native method at runtime operation.

Our approach is to convert the runtime library code to LLVM IR at the same time the user compiles the runtime library to native code. The native function expressed as a LLVM IR is expanded to the main memory at runtime, and LLVMJIT can easily inline native function code into LLVM IR that is converted from KonohaIR.

We assume that native functions are composed of two parts, the function body and the wrapper to adjust with the common interface. As mentioned in the previous section, the inlining wrapper function of the native function can allow the removal of the operation relevant to KonohaStack and thus remove the cost of method calls.

With inlining the wrapper function, LLVMJIT will remove operations relevant to KonohaStack in the following steps. At first, LLVMJIT determines the layout of KonohaStack at the time the wrapper function is called. The layout can be analyzed from the type information of the method. Next, LLVMJIT performs the inline wrapper function to the caller code. Finally, based on the layout of the KonohaStack, LLVMJIT transforms the values placed on KonohaStack, such as arguments and return values, to local variables of the caller functions.

We perform inlining with the following rules:

- File size of the runtime library is less than 32 KB.
- The length of the wrapper function is equal to, or less than, 64.

The above magic numbers, 32 KB and 64, are the parameters to determine how often inlining happens. Inlining can make performance levels worse by filling up the instruction cache if it is applied excessively. We determined the above default parameters of 64 and 32 KB heuristically. 64 is a number for inline wrapper functions to prevent over inlining. When the LLVMJIT inline native function is provided by a large scale library, we cannot ignore the startup and execution time of the LLVM inline optimizer, which includes loading the LLVM IR of the library into the main memory. To reduce the compile time, we set the threshold to 32 KB. **Figure 8** shows the conversion process of the runtime library into LLVM IR, written in C. Before the user executes the application code, we compile runtime libraries with an LLVM-based C compiler, named clang, into the LLVM IR. At the same time, we compiled native code for these libraries. In carrying out the generation of the LLVM IR from KonohaIR, LLVMJIT performs inlining native methods using LLVM IR that clang generates. Finally, LLVMJIT removes the stack operations relevant to KonohaStack and the procedure of inlining is completed.

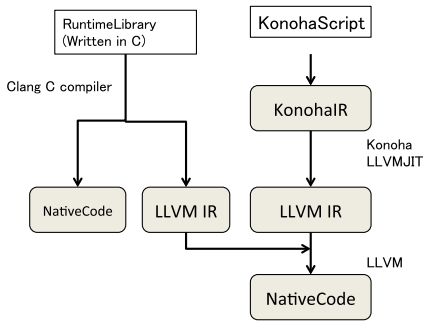

Fig. 8 inlining native method provied by runtime library.

```
typedef struct {
  hObject h;
  kObject **fields; /* fields == &fields*/
  kObject *fieldsI[3];
} kSmallObject;
```

Fig. 9 The memory layout of the object that has less than 3 fields.

4.3 Field Access and Memory Layout

Field access is one of the functions frequently used in an object-oriented programming language. We can expect that accelerating field access will have a high impact on performance.

How to access fields of the object will be determined by a memory layout of the object that the runtime has been established. As a native way, when the JIT compiler adopts its own layout to represent the object, the compiler needs to track all parts of the runtime that handle the objects.

In KonohaScript, the object type has an array field to store the member variables (see Fig. 2) and the fields are allocated in a heap and managed by GC. By improving the data locality of the object's field, we store member values directly in the region of the object, as long as the object has less than three fields (**Fig. 9**).

We also modified the data layout of the object instance of the class annotated with @Final, which has less than three fields to examine whether the changes in memory layout affect the performance of the accessing fields.

5. Performance Evaluation

In this section, we describe the effects derived from the LLVMJIT compiler and our optimization techniques. Benchmarks run on the following computing environment.

- CPU: Intel(R) CoreTM i7 2.2 GHz
- Memory: 8 GB, 1333 MHz
- OS: MacOSX 10.7.2
- C compiler: GCC 4.4.5, G++ 4.4.5
- LLVM: version 3.0
- Java: HotSpot 64 Bit 1.6.0_33

We use a set of optimization configurations provided by LLVM. We use StandardFunctionPass for the function's optimization pass and StandardModulePass for the entire program optimization.

As described in Section 4, we developed three different optimization techniques to improve performance of the JIT compiled code: API, INL, and FLD short. We selected these techniques to make it clear how much improved these techniques are:

- LLVMJIT without our optimizer (JIT)

Table 1 Description of benchmarks.

Benchmark	Description
AObench [8]	Ray Tracing (object creation, field access)
DeltaBlue [9]	Constraint solver benchmark (field access)
NBody	Simulation of N-body problem of planetary NBody (field access)
Binarytrees	A large amount of the product a binary tree of depth n (object creation)
Mandelbrot	Compute the Mandelbrot set (floating-point arithmetic)
Richards [9]	OS kernel simulation (Field access, method call)
Spectralnorm	compute the 2-norm of a square matrix (floating-point arithmetic, array access)

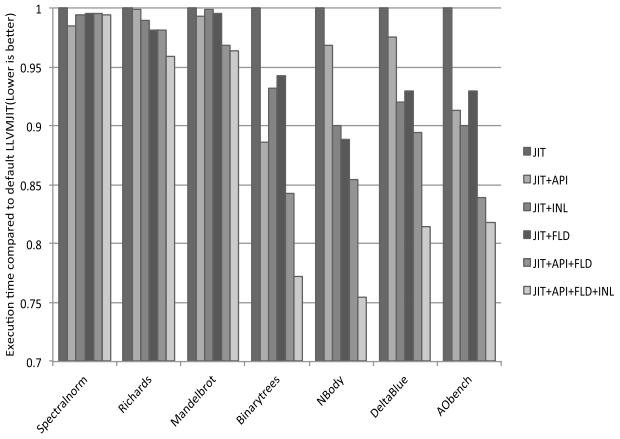

Fig. 10 Performance improvements of each optimization.

- LLVMJIT with a specialized interface (JIT+API)
- LLVMJIT with inlining native function (JIT+ILN)
- LLVMJIT with optimizing field accessing (JIT+FLD)
- LLVMJIT with a specialized interface and optimizing field accessing (JIT+API+FLD)
- LLVMJIT with all optimizations (JIT+API+FLD+ILN)

5.1 Benchmark Programs

We evaluated the performance of seven programs ported to KonohaScript (details of the benchmark programs are listed in **Table 1**).

5.2 Effectiveness of LLVMJIT and Optimizations

Figure 10 shows a comparison of the performance of our JIT compiler with optimization, which is described in Section 4. The overall execution time includes the compilation time of our JIT compiler. In Fig. 10, the vertical axis shows the execution time that is normalized by the default LLVMJIT.

We observed about 20 percent performance improvements in the fully optimized code for AOBench, DeltaBlue, Nbody, and Binarytrees. In addition, these performance improvements indicate that LLVMJIT reduced the overhead of using runtime libraries. Our optimization technique implemented in this paper is only eligible for the object that contains more than three fields. This is because benchmarks such as AOBench and Binarytrees create the object with small sized fields, and we observed that this optimization reduced the number of memory references by fixing the memory layout of an object.

Table 2 Comparison of the compilation time of the benchmark program.

Benchamrk	CompileTime (0, 0) (msec)	CompileTime (32, 64) (msec)	CompileTime (∞, ∞) (msec)
AOBench	240	281	3106
DeltaBlue	328	336	1584
NBody	198	203	3375
BinaryTrees	101	136	1702
Mandelbrot	120	140	1562
Richards	255	266	1877
Spectralnorm	140	151	1340

In contrast, we no longer see any performance improvements on Mandelbrot, Richards and Spectralnorm. With the observation of the benchmark code and LLMV IR generated by LLVMJIT, we consider there to be three reasons why the performance did not improve. First, Mandelbrot and Spectralnorm consist of floating-point arithmetic operations and operations of an array, and each operation does not use the runtime libraries. Therefore, we observed no performance improvement compared to the no optimization option. Second, because Richards uses the method call with dynamic bindings in many places, we are not able to perform optimization for the specializing method interface of the JIT compiled code. The third reason is that Mandelbrot and Richards use the object with more than four fields and that means our optimization technique is not available. In the main kernel of each benchmark, we are not able to improve speed field access.

Table 2 shows the compilation time for different optimization levels of native method inlining. Each configuration for optimization is classified by the size of the extension library file and the number of instructions in the native function.

- (Filesize, Instructions) = (0, 0) disables method inlining for the runtime library. In this configuration, LLVMJIT performs inlining of only user-defined methods.
- (Filesize, Instructions) = (32 KB, 64) enables method inlining of the runtime library when the size of the runtime library is less than 32 KB and the length of the native function is equal to, or less than, 64. In this case, the native method provided by Math library is inlined but IO library is not.
- (Filesize, Instructions) = (∞, ∞) preforms inlining of all the methods provided by the runtime libraries.

The compilation time performing inlining on all native methods results in 20–40 percent of execution time and takes up a lot of execution time. On the other hand, with two other configurations, the compile time accounted for about 5 percent of the execution time and we observed that performing inlining of the native method does not adversely affect execution time.

5.3 Comparison to C++, and Java

In this section, we compare the performance of our JIT compiler with C++ and Java. We have chosen four benchmarks from the benchmarks shown in Table 1 and used equivalent implementations of benchmark programs in C and Java.

Figure 11 compares the execution time of our JIT compiler with C++ and Java. We observed that our JIT compiler has an approximate equivalence to the performance of C++ and Java in NBody, Spectralnorm, and Mandelbrot. On the other hand, our LLVMJIT with all optimizations is inferior to C++ and Java on Binarytrees.

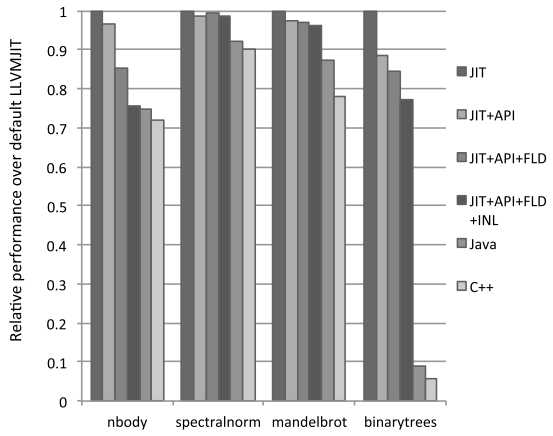

Fig. 11 Performance comparisons between C++, Java and our JIT compiler.

Table 3 Comparisons of memory usage of a native code generation between Java and our JIT compiler.

Benchamrk	LLVMJIT (bytes)	Java (bytes)
Nbody	7700	3233
Spectralnorm	5591	3915
Mandelbrot	6223	3272
Binarytrees	7521	4211

Table 4 Comparisons of the compilation time of a native code generation between Java and our JIT compiler.

Benchmark	LLVMJIT (sec)	Java (sec)
Nbody	0.203	0.083
Spectralnorm	0.151	0.095
Mandelbrot	0.14	0.087
Binarytrees	0.136	0.091

As described in Table 1, Binarytrees composes a large part of the computation with object creation. We observed that garbage collection is devoted to about 70 percent of the execution time. We now use a simple mark-and-sweep algorithm for garbage collection, and we will use a smart GC algorithm to fill the performance gap.

Table 3 compares memory usage for the compiled code that Java and our JIT compiler generate, and **Table 4** compares the compilation time of our JIT compiler and Java's JIT compiler. Because our JIT compiler does not yet preform compilation selectively, and Java's JIT compiler does, our JIT compiler is inferior to both memory usage and the compilation time when compared with Java's.

6. Related Work

Due to the popularity of JavaScript and Ruby on the Web, scripting languages have received both industrial and academic attention more recently. Several attempts have been made towards a JIT compiler of the existing scripting languages based on LLVM. Rubinius [10] is one implementation on Ruby that uses LLVM as a JIT compiler backend. The main parts of the runtime library of Rubinius are written in Ruby and the JIT compiler compiles the application code into machine code at runtime.

Compilation techniques for dynamic language have a long history from Smalltalk and Self, to more recent languages like Python and JavaScript. Today, the trace-based JIT for dynamically typed languages are TraceMonkey [1] (JavaScript) and PyPy [3] (Python). This approach of tracing through the runtime lends itself well to specialization due to the collected trace capture type information.

As these compilers collect trace information using the interpreter, it is difficult to make a comparison with the runtime library written in other languages such as C. In this work, we compile both application code and the runtime library to LLVM IR, and our JIT compiler provides the opportunity to optimize the runtime library.

As with the inlining of the native method described in Section 4, VMKit [11] compiles the runtime libraries into LLVM IR, and VMKit performs the inlining of the native method at runtime. VMKit only prepares LLVM IR of the runtime library, but we generate both LLVM IR and machine code in the library. In addition, our JIT compiler performs inlining selectively, depending on the size of the library and length of native methods.

7. Conclusion

In this paper, we have described the design and implementation of our JIT compiler, LLVMJIT and three optimization techniques to help reduce language runtime overhead. We evaluated the performance by using a benchmark application built on a JIT compiler. We conclude that modification of the object layout and method invocation mechanism can affect the performance of the JIT compiler generated code.

Acknowledgments This work is done in part by JST/CREST research grant "Dependable Operating System for Practical Use".

References

[1] Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M.R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E.W., Reitmaier, R., Bebenita, M., Chang, M. and Franz, M.: Trace-based just-in-time type specialization for dynamic languages, *Proc. 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, *PLDI '09*, New York, NY, USA, pp.465–478, ACM (online), DOI: http://doi.acm.org/10.1145/1542476.1542528 (2009).

[2] Google: V8 JavaScript Engine, available from ⟨http://code.google.com/p/v8/⟩.

[3] Bolz, C.F., Cuni, A., Fijalkowski, M. and Rigo, A.: Tracing the meta-level: PyPy's tracing JIT compiler, *ICOOOLPS '09: Proc. 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, New York, NY, USA, pp.18–25, ACM (online), DOI: http://doi.acm.org/10.1145/1565824.1565827 (2009).

[4] Chang, M., Mathiske, B., Smith, E., Chaudhuri, A., Gal, A., Bebenita, M., Wimmer, C. and Franz, M.: The impact of optional type information on jit compilation of dynamically typed languages, *Proc. 7th Symposium on Dynamic Languages*, DLS '11, New York, NY, USA, pp.13–24, ACM (online), DOI: http://doi.acm.org/10.1145/2047849.2047853 (2011).

[5] Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *Proc. 2004 International Symposium on Code Generation and Optimization* (*CGO'04*), Palo Alto, California (2004).

[6] Kuramitsu, K.: Konoha — Implementing a Static Scripting Language with Dynamic Behaviors, *Workshop on Self-sustaining Systems 2010*, S3, The University of Tokyo, Japan (2010).

[7] Kuramitsu, K.: KonohaScript The design and evolution of statically typed scripting language (in Japanese), Japan Society for Software Science and Technology (2011).

[8] Syoyo Fujita: Ambient Occlusion Benchmark, available from ⟨http://code.google.com/p/aobench/⟩.

[9] Laboratories, S.M.: Benchmarking Java with Richards and DeltaBlue, available from ⟨http://labs.oracle.com/people/mario/

java_benchmarking/index.html⟩.
[10] Evan Phoenix: available from ⟨http://rubini.us/⟩.
[11] Geoffray, N., Thomas, G., Lawall, J., Muller, G. and Folliot, B.: VMKit: A substrate for managed runtime environments, *Proc. 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10, New York, NY, USA, pp.51–62, ACM (online), DOI: 10.1145/1735997.1736006 (2010).

Masahiro Ide was born in 1988. He received his M.S. degree from Yokohama National University in 2012. He is a member of IPSJ, ACM.

Kimio Kuramitsu was born in 1972. He has been Associate Professor of Yokohama National University since 2007. He received Yamashita Memorial Research Award 2008. He is a member of JSSST, IPJS and ACM.