Doctoral Thesis

Study on method-based and trace-based just-in-time compilation for scripting languages

スクリプト言語向けメソッド方式および トレース方式Just-In-Timeコンパイラの研究

Masahiro Ide

Graduate School of Engineering,
Yokohama National University

Supervisor: Associate Professor Kimio Kuramitsu

September 2015

Abstract

Applications written in scripting languages are widely used on web application development. Even on the server side application, programmers are using scripting languages such as Ruby to building complex applications. Since the complexity and the number of the applications written in these scripting languages grown, the performance of these applications is becoming important.

A scripting language's design provides the high productivity and easiness of writing programs to the programmer, while a performance of a scripting language implementation still remains an issue. To improve the performance of the scripting language, many researchers attracted the attention of a JIT (Just-In-Time) compilation technique for scripting language to construct high-performance implementations.

In this thesis, we propose two JIT compilers for the implementation of scripting languages. The difficulty in terms of aiming to improve the performance of JIT compiled code is in the dynamic typing and cooperation with its own runtime. In Chapter 3, we propose a method-based JIT compiler for a statically typed scripting language KonohaScript. We reveal the performance of the portion other than the type checking using a statically typed scripting language. We use a statically typed scripting language KonohaScript to analyze language runtime the performance impact of the JIT compiled code.

Another proposal is a trace-based JIT compiler for a dynamically typed scripting language Ruby. Chapter 4 describes RuJIT, a trace-based JIT compiler for Ruby, which traces program code to determine frequently executed traces (hot paths and loops) in running programs and emits optimized machine code specialized to these traces. We describe the design and implementation of the RuJIT compiler. Then we show that two Ruby's functions and characteristics make difficult to improve the performance of JIT compiled code. To produce better quality code, we describe solutions for these issues as well as implementation of optimization techniques. We also evaluate the application of trace-based JIT compilation technique to Ruby interpreter and explain performance impact of our approach.

要旨

近年，Web アプリケーション開発を中心にスクリプト言語が広く利用されている．特にサーバサイドアプリケーションにおいては複雑なアプリケーションの記述にもスクリプト言語，例えば Ruby 言語が利用されている．そしてスクリプト言語で記述されたアプリケーションの複雑化，大規模化に伴い，アプリケーションの実行パフォーマンスにも注目が集まってきている．

スクリプト言語は，プログラムの書きやすさ，生産性の高さに注力して設計が行われており，その実行パフォーマンスは犠牲となってきた．そこで，これまでに多くの研究者や開発者がスクリプト言語処理系の実行速度向上のために JIT (Just-In-Time) コンパイル技術に取り組んできた．

我々は，2 つのスクリプト言語処理系に対しそれぞれ JIT コンパイラを提案する．JIT コンパイラによって言語処理性能の向上を目指す上で困難な点は動的型付けと独自の言語ランタイムにある．まず Chapter 3 では，静的型付けスクリプト言語 Konoha の上にメソッド方式 JIT コンパイラを構築し，型検査のオーバーヘッド以外の部分の JIT コンパイルの性能を明らかにする． 我々は JIT コンパイラが生成したコードのうち言語ランタイムが性能に与える影響について分析を行い，生成されたコードと言語ランタイムの連携で生じるオーバヘッドを改善する最適化手法を構築し，評価を行った．

次に Chapter 4 では本研究では JIT コンパイル技術の一手法である Trace-based JIT コンパイル技術を Ruby 処理系に適用した．我々は，Ruby 言語の C 言語による処理系 CRuby を対象に，CRuby が生成したバイトコードをトレースし，頻繁に実行された箇所（たとえばループなど) を実行時に機械語に変換する Trace-based JIT コンパイラ RuJIT を構築した．本研究では RuJIT に実行時最適化技術を Ruby プログラムの意味を変更しないよう適用した．とくに Ruby の動的特性や Ruby プログラムに頻出す

る記法の 1 つであるブロック記法に着目した最適化手法を適用した．最後に，構築した JIT コンパイル手法，最適化手法の性能評価を行うため，Ruby 処理系に付属するベンチマークセットを利用し，それぞれの最適化がどの程度性能に寄与しているかを評価した．

Acknowledgments

I am deeply indebted to my supervisor Associate Prof. Kimio Kuramitsu, for leading me to this very exciting area in computer science, programming language design, and implementation. And I am also indebted to him for the financial assistance. He provided me numerous enlightening suggestions and comments. He also mentally encouraged me in my work, and gave me numerous changes to meet many researchers.

I am grateful to Prof. Koichi Shudo, he and IPA adopted me IPA Exploratory IT Human Resources Project (the MITOH youth program). His stimulating suggestions and their financial assistance help to building a foundation of my research.

I am grateful to the professors in the department of physical and information engineering in Yokohama National University. They have stimulated me in positive and important ways since I was an undergraduate student. Also, I deeply thank to students of Kuramitsu laboratory. They gave me much advice and encouragement.

I am more than thankful to Dr. Koichi Sasada and Ruby Association, for their insightful comments on the implementation aspects of my work. They pointed out numerous problems in our system of early stages.

I would like to give my special thanks to Yukihiro Matsumoto, the author of the Ruby language, which is the target language of my thesis. He developed a grate programming language, and his thesis would not have been possible without his team's help.

In addition, I am indebted thank to JST/CREST research grants "Dependable Embedded Operating System for Practical Use" and JST/DEOS R&D Center for their financial assistance. This thesis would not have been possible without their help.

Finally, I am thankful to my family for their advice and financial help.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

One of the oldest high-level programming language that is still in use today is LISP[36]. The features that LISP provided have been inherited by many of the programming language called dynamic language today. The great flexibility of these languages, the ease of development has always made the programmers very attractive.

Due to various reasons, dynamic languages became popular in the last two decades, with several so called *scripting languages* became mainstream. One of the reasons for that is the technological progress of computer hardware. This solved the slowness of implementations of scripting languages and expanded the program domains of the scripting languages can solve.

In addition, to solve the problem of the making scripting languages faster, as [4] nicely illustrates, many researchers have been tackled in a number of different ways over the years. And these approaches can be divided into two categories, writing fast interpreters and writing Just-In-Time(JIT) compilers.

The key idea behind writing fast interpreters is to reduce the cost of interpretation overhead. However this approach works effectively only for the cost of interpretation that dominates the total execution time. On the other hand, JIT compilers have proved to be very effective at optimizing different kinds of overhead introduced by scripting languages. The main characteristic of JIT compilers is that they generate machine code at runtime and only when needed.

In this thesis, we explore and investigate new techniques for JIT compilation. In particular, we propse two JIT compiler implementation that emphasize the two points of performance bottleneck of JIT compiler for scripting languages: type system(dynamic typing or static typing) and runtime overhead of generated native code.

Our choice of the bottleneck point is based on the current design of Ruby, which emphasize the high productivity and easiness of writing rather than execution efficiency. Our goal is to improve the execution performance of the implementations of scripting language using JIT compiler. We firstly tackled to a method-based JIT compiler for a statically typed scripting language, Konoha to reveal the performance of JIT compiled code without any type checking. And second, we studied application of JIT compilation to the Ruby interpreter.

1.2 Overview

This thesis is organized as follows. The following Chapter 2 describe a JIT compilation. In Chapter 3, we develop a efficient method-based JIT compiler for Konoha, a statically typed scripting language. Chapter 4 then describes our trace-based JIT compiler, RuJIT and discuss how well RuJIT performs for a set of benchmark programs. Related work is discussed in Chapter 5. This thesis ends with conclusions and an outlook on future work in Chapter 6.

Chapter 2

Just-in-time Compilation

2.1 Overview of Just-in-time Compilation

In contrast to the ahead-of-time (AOT) compilation, Just-In-Time (JIT) compilation compiled code at runtime. And recently, JIT compilation is employed by a modern implementation of programming languages and programing language environment; for example, Java Virtual Machine[32, 1, 23], implementations of JavaScript[22, 19], and Python[9]. Because a JIT compiler compiles at runtime, JIT compiler may improve the code during optimization phase with the information collected at runtime. For example, an AOT compiler for Java may compiles virtual function call using an indirect branch. This is costly since hardware branch prediction is difficult to predict the target of branches, and it causes pipeline stalls. A JIT compiler, on the other hand, compile each call site to a direct branch to the method of most commonly encountered type, proceeded by a check that the object type is indeed as expected. If this check fails, the execution falls back to the indirect branch technique. JIT compilers for dynamically typed languages such as JavaScript and Python employ this method.

One of the elements that determine the performance of a JIT compiler is to determine what constitutes a compilation unit. The compilation unit of the traditional compiler is a whole file or module. Because the compilation is interleaved with program execution, compiling large part of the program at once can cause a substantial delay in the program. And compiling code also increases the memory usage of the program, which leads to another reason to minimize the total amount of compiled code.

JIT compilers commonly choose a compilation unit from following unit : a method and a trace. And each JIT compiler that adopts these compilation units is called method-based JIT, trace-based JIT, respectively.

Method-based JIT

A method-based JIT uses a single method as compilation unit. Using profile data, the compilers identified hot functions and then compiled and optimized using standard compilation techniques on the control flow graph of the method. The compiler typically has to traverse the program until a fixed-point is reached slowing down compilation speed.

Since AOT compilers include traditional compilers employ a method as compilation unit, the same optimization techniques used for AOT compiler can be used in method-based JIT compiler (if these optimizations run quickly).

A method-based JIT compiler uses a profiling data such as the number of times a function is invoked. In Figure2.1 (A), a method-based JIT compiler compiles two methods (method A and B) separately.

Trace-based JIT

A trace-based JIT compiler uses a trace as compilation unit. A trace is a liner sequence of instructions that has a single entry point and one or more exit points. A trace-based JIT compiler is not bounded by the method boundary, and it often includes part of several functions. And if there are multiple frequently executed paths in the program, these paths may cover by multiple traces.

In Figure2.1 (B), a trace-based JIT compiler creates traces from each execution path, respectively. The generated traces are often contained some replication of code. However, these replicated code increase the opportunities for optimization within each trace.

Following section describes the detail of trace-based JIT compiler.

2.2 Method-based JIT Compiler

A method-based JIT compiler compiles a single function at a time. Because the natural compilation unit for the implementation of most programming languages is a single method (or a single function) and traditional AOT compilers uses a function as a compilation unit,

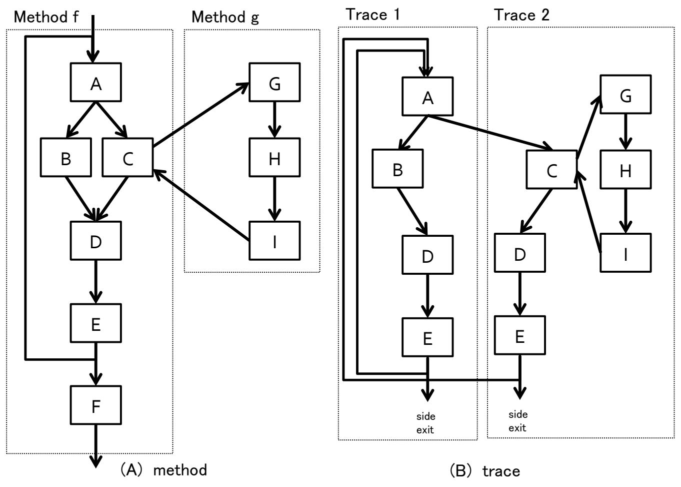


Figure 2.1: Compilation units. (A) compilation units for a method-based JIT compiler, (B) compilation units for a trace-based JIT compiler

method-based JIT compiler can use same techniques that is used in AOT compiler if they can run quickly enough. And it is also easy to connect compiled code with bytecode if both of code use the same interface (or calling convention).

As mentioned above, it is easy for a method-based JIT compiler to interface compiled code with bytecode. However it is difficult to replace a running function to compiled code. To achieve this, a method-based JIT compiler uses on-stack replacement (OSR)[25] technique to replace with an optimized version of the same running function. Using OSR technique, JIT compilers collects the execution state of a particular method. And JIT compiler generates the code that updates the value extracted from the stack frame of the previous version of method and finally, JIT compiler replace old stack frame with new one and restarts execution in the new version of method.

2.3 Trace-based JIT Compiler

A trace-based JIT compiler compiles only a trace. A trace is straight-line codes with no control flow join points except that the last instruction might loop back to the beginning.

If the original code contains conditional branch, a trace contains a guard. A guard is a runtime check instruction to check that the program state is indeed as expected.

When the guard fails, the execution leaves the current trace and fall back to the interpreter or invokes another trace.

Trace-based JIT commonly has three execution modes.

- Interpreter mode
- Record mode
- Native mode

Interpreter mode

In interpreter mode, the interpreter executes bytecode instructions, and the compiler detects the target of compilation. Execution of a trace-based JIT starts from this mode. In interpreter mode, a trace-based JIT compiler finds specific certain bytecode instruction to detect a trace head. A trace head may be an entry point of a trace. When the interpreter executes such instructions, trace-based JIT compiler first look for the traces containing the trace head to the trace from the trace cache that contains already recorded traces. If the trace is found, execution is transferred into the machine code that the trace has. Then the interpreter is switched into native mode. On the other hand, if the trace is not found, the hotness counter is assigned to the trace head, and the hotness counter is incremented. When the hotness counter exceeds the threshold, the interpreter is switched into record mode.

Section 2.3.1 discusses techniques for selecting trace head.

Record mode

In record mode, the interpreter executes bytecode instructions and records which instruction the interpreter executes. In common case, record mode is finished when the following condition is fulfilled.

- a loop is found. i.e., execution reaches a trace head.
- the start of an existing trace is found.
- an exception is thrown.
- the recorded trace has become too long.

When a trace-based JIT recording finished successfully, the recorded trace are compiled into machine code. Then the machine code and the recorded trace are placed into the trace cache, and the interpreter is switched into interpreter mode.

Native mode

In native mode, we execute compiled machine code and the execution continues until a guard in the native code fails. If a guard fails, the execution leaves the machine code and falls back to the interpreter. Then the interpreter is switched into interpreter mode. In this case, the interpreter marked an exit point of the trace to record uncovered execution path.

2.3.1 Trace Selection

The quality of the selected trace is a key feature to determine the performance of a trace-based JIT compiler. If the recorded trace is too short, it cause frequent transitioning between native code and the interpreter and has less possibility of optimization. In contrast, the longer trace means less transition and can apply more optimization.

Because a trace is created speculatively by the path that is frequently performed in a particular code path, the overhead of native code call is increased when a trace-based JIT compiler records the wrong execution path as trace. Therefore, a trace-based JIT compiler needs to detect hot loop such as to occupy most of the execution time of the program, and record the trace.

A trace-based JIT compiler uses a counter to profile the program. When the interpreter executes certain bytecode instruction, trace-based JIT compiler marks these instructions as potential *trace heads*, which is places where a trace may start, and associate a hotness counter with each trace head. Typically, a trace-based JIT compiler uses following the algorithm to detect hot path.

- Next executed tail (NET)[5]
- Last executed iteration(LEI) [24]
- Natural loop first (NLF)[40]

Next executed tail (NET)

Next executed tail (NET) strategy is heuristic to detect potential trace heads. The NET strategy considers all of the targets of backward branches to be potential trace heads. This heuristic is derived from that every loop contains at least one backward branch and the target of a backward branch is very likely to be a loop header, and thus the head of several hot traces in the loop body.

While it is inaccurate in some cases, the NET strategy can detect hot trace without any knowledge about control flow information of the program.

Last Executed Iteration(LEI)

Another strategy to detect hot trace is Last Executed Iteration (LEI) strategy. LEI strategy is a refinement of NET strategy and as same as NET strategy, LEI strategy considers the target of backward branches to be potential trace heads. LEI strategy records branch targets in a branch history buffer, and when a branch target appears twice in the trace recording buffer, the LEI strategy creates a trace corresponding to the path between the repeating branch targets.

Natural loop first

When we can able to access to the information about program structures (such as loops), Natural loop first strategy constructs a trace starting from a loop header and spanning the loop body.

All trace selection schemes also consider the exit point of a trace as potential trace heads. This consideration ensures that a trace-based JIT compiler can compiles uncovered execution path.

2.3.2 Trace Cache

The recorded trace and the associated machine code are placed into the trace cache.

When recording finished successfully, the recorded trace compiled into machine code. Then the machine code and the recorded trace are placed into the trace cache, and the interpreter is switched into interpreter mode.

Note that trace-based JIT compiler compiles traces for the behavior that was observed when the trace was recorded. When the behavior of the program changes, because assumptions that were constructed from the old behavior may not be suitable for the new behavior, the compiler needs to remove old cold trace or re-compile the recorded traces with the new behavior.

For example, Dynamo[5] employs a novel simple heuristic to detect such changes in behavior. When the behavior changes, new parts of the program become hot, and it causes a sharp rise in new trace creation. If the trace creation rate increase, Dynamo flush the trace cache and discards all machine code in the trace cache. Discarded code will quickly re-compiled and restore into the trace cache.

Chapter 3

A Method-based JIT compilation for a statically-typed scripting language

3.1 Introduction

In the past decade, significant advances have been seen in scripting language just-in-time (JIT) compilers for JavaScript[19, 22], as well as for other languages such as Python[9]. 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.[11]. 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 chapter, 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[35]) 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.

3.2 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.

3.2.1 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 it at runtime. In this section, 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.

3.2.2 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.

3.2.3 KonohaScript Runtime System

This section describes the runtime library of KonohaScript in relation to the following sections. KonohaScript[33] 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 [33]. 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.

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

Figure3.1 shows the elements that make up the KonohaScript runtimes.

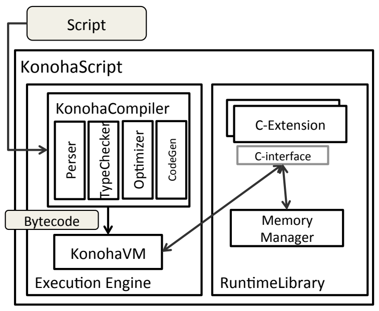


Figure 3.1: Architecture of KonohaScript

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, each object is aligned in 64byte and has a memory layout as shown in Figure3.2. 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.

```
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;
```

Figure 3.2: Memory layout of Object

3.3 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.

3.3.1 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.

3.3.2 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.

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.

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.

3.3.3 Sample LLVMJIT Execution

In this section, we describe the compilation flow of LLVMJIT using an example. Figure3.3 shows the source code used in the following description.

At first, LLVMJIT translates the KonohaScript code in Figure3.3 into KonohaIR in Figure3.4. 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 Figure3.5. 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.

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

Figure 3.3: An example KonohaScript program

3.4 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

3.4.1 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.

As described in Section 3.2, 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

```
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
}
```

Figure 3.4: The KonohaIR sequence for the program in Figure3.3

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

```
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
}
```

Figure 3.5: The LLVM IR sequence for the program in Figure3.4

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.

Figure3.6 shows an example of the factorial method written in KonohaScript and JIT compiled code, with a common method interface and specialized interface in Figure3.7. We noted that we would rewrite JIT compiled code for implementation in C for readability.

3.4.2 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.

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

Figure 3.6: 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);
 }
```

Figure 3.7: The KonohaIR sequence for the program in Figure3.6 with specialized interface

In scripting 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 in-line 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 32KB.
- The length of the wrapper function is equal to, or less than, 64.

The above magic numbers, 32KB 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 32KB 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 32KB. Figure3.8 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.

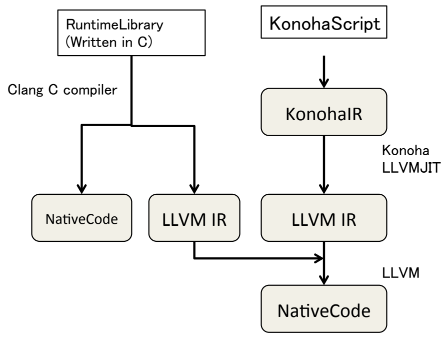


Figure 3.8: inlining native method provied by runtime library

3.4.3 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 Figure3.2) 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 (Figure3.9).

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.

3.5 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.

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

Figure 3.9: The memory layout of the object that has less than 3 fields

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

3.5.1 Benchmark Programs

We evaluated the performance of seven programs ported to KonohaScript (details of the benchmark programs are listed in Table 3.1).

Benchmark	Description
AObench[47]	Ray Tracing (object creation, field access)
DeltaBlue[34]	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[34]	OS kernel simulation(Field access, method call)
Spectralnorm	compute the 2-norm of a square matrix(floating-point arithmetic, array access)

Table 3.1: Description of benchmarks

3.5.2 Effectiveness of LLVMJIT and Optimizations

Figure3.10 shows a comparison of the performance of our JIT compiler with optimization, which is described in Section 3.4. The overall execution time includes the compilation time of our JIT compiler. In Figure3.10, 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 section is only eligible for the object that contains more than three fields. This is because benchmarks such as AOBench and Binarytrees

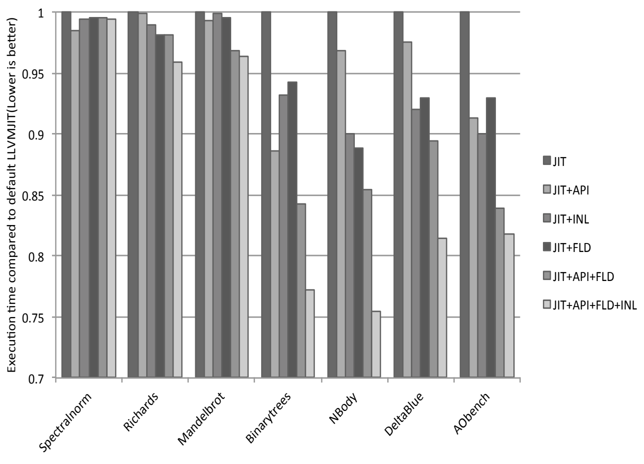


Figure 3.10: Performance improvements of each optimization

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.

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 3.2 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) = (32KB, 64) enables method inlining of the runtime library when the size of the runtime library is less than 32KB 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.

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

Table 3.2: Comparison of the compilation time of the benchmark program

3.5.3 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 3.1 and used equivalent implementations of benchmark programs in C and Java.

Figure3.11 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.

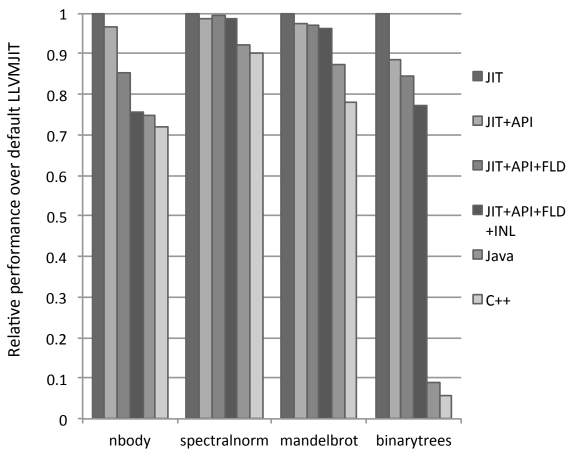


Figure 3.11: Performance comparisons between C++, Java and our JIT compiler

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

Table 3.3: Comparisons of memory usage of a native code generation between Java and our JIT compiler

As described in Table 3.1, 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 3.3 compares memory usage for the compiled code that Java and our JIT compiler generate, and Table 3.4 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.

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

Table 3.4: Comparisons of the compilation time of a native code generation between Java and our JIT compiler

3.6 Summary

In this section, 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. And we show that modification of the object layout and method invocation mechanism can affect the performance of the JIT compiler generated code.

Chapter 4

A trace-based JIT compiler for Ruby

4.1 Introduction

4.1.1 Ruby

The Ruby language is a dynamic typed object-oriented programming language that was designed by Yukihiro Matsumoto. Development of the original implementation of Ruby language is stared from 1993 by Yukihiro Matsumoto. The Ruby programming language[52] is a dynamically-typed object-oriented language. Like Smalltalk, Ruby has no type declarations, allowing programmers to rapidly build and modify systems and this provides the high productivity and easiness of writing programs to the programmer. Also, Ruby is blended parts of other programming languages such as Perl and Lisp, and provides many features such as dynamic typing, meta-programming, high interoperability with other language, or closures, which are common with other scripting languages (e.g. JavaScript, Perl, or Python).

In recent days, Ruby is widely used in web applications, especially used in server-side applications. Also, as a general purpose programming language, Ruby is used in other applications such as GUI applications, game development, and business system. And Ruby community started to implement other Ruby implementation such as JRuby[31] (Java implementation), IronRuby [28](C# implementation), and Rubinius[41] (Ruby implementation).

However, on the other hand, because most of these features are implemented as a part of the interpreter, there are significant performance gap between Ruby and high-performance languages such as Java and C. To fill the performance gap, many researchers and scripting language developers have newly implemented Just-In-Time (JIT) compilers[9, 19, 40, 48, 22]. In Ruby, MacRuby[43], Rubinius[41] and Ishii's VM[29] introduced a method-based JIT compiler to improve an execution speed of the Ruby language.

4.1.2 Challenge of the development of JIT compiler for Ruby

There are two problems to speed up the program written in Ruby language. The first one is that Ruby performance is often bounded by its dynamically typed feature. Moreover, the second one is to support the dynamic features of Ruby such as reflections (the feature that access runtime information at runtime).

The first one, dynamic typing enables the programmer to improve the reusability of the components because the programmers need not to specify the type of variables and functions in the program. The second one, reflection facilitate meta-programming to change the behavior of program depending on the input data at runtime.

These dynamic features provides the flexibility of programming to Ruby programmers, but compilers are not able to do static analysis and make generation of efficient target code difficult. For example, without static information about the types of variables, compilers must emit code to handle all possible combinations of operand types. This causes a large runtime overhead on the generated code.

Also to this problem, to achieve a practical and fast JIT compiler, we need to solve problem describes below, which contains compatibility problem, maintenance ability problem, and portability problem.

In Ruby, we already have been developed many libraries and using these libraries, we can construct complex software in many environments.

And we need to maintain a backward compatibility without modifications as possible to reuse huge program resources. In particular, maintaining compatibility and portability of both the extension library that extends Ruby and the C API are main issues for constructing practical Ruby implementation.

The original Ruby implementation is called CRuby, which is written in the C language. And CRuby is the reference implementation of Ruby.

In this section, we describe the features of the Ruby language. And we will introduce CRuby.

4.1.3 Ruby Feature

Ruby has features listed below. Note that this thesis is not intended to describe the detail of the Ruby language. Details of the language specification are described in ISO/IEC[30]

- Object-oriented feature (class, method invocation, mixin)
- Exception
- Block, closure
- Garbage collection
- Dynamic code loading
- Extension Library

Object-oriented feature

Ruby has a class-based object-oriented feature. Ruby's object system is based on a single inheritance, and we can define the method in classes. The following example (1) defines Human class and instance method **say**. And (2) create the instance of Human class and invoking say method.

```
# (1) class definition
class Human
  def say (name)
    puts "Hello␣" + name
  end
end
# create instance and method call
```

```
naruto = Human.new
naruto.say "Sasuke"
```

Also, Ruby introduced a module, there by enabling flexibility enhancements to a single inheritance-based object system. The module is the feature that can define methods like class, but not be instantiated. A module provides the function of the module to the class by **include** expression. The following example defines one module M and two classes C and D. Module M defines say method and class C and D can utilized say method when includes module M.

```
module M
   def say(name)
      puts "Hello␣" + name
   end
end
class C
  include M
end
class D
  include M
end
```

Also to this feature, Ruby introduced instance-specific method; it is possible to define a specific method for each object. The following example creates an instance of class Human described in the above example, and defines instance-specific method **say_twice** to obj. And using an instance-specific method, we can redefine the methods already defined in the class. This feature is similar to the feature that is contained by prototype-based object-oriented languages, such as SELF[49] and JavaScript[16].

```
obj = Human.new
def obj.say_twice(name)
   self.say name
   self.say name
end
```

Block, closure

In Ruby, we can pass the piece of code, in the form of a block, to a method. This block is called a Proc object; it is as same as the closure in functional languages.

Block is the piece of code that is enclosed in curly braces (or do-end). And the arguments for a block is described as |*a*|. And the block that has been passed to the method is invoked by the yield statement.

Block has two kinds of variables, one is a block local variable, which is defined in the block, and another is a method local variable, which can be accessed during method execution.

In Ruby, the block is treated as a Proc object, and block and Proc object can be interconversion. The block converted from Proc object always accesses both the block-local variables and the method-local variables. In other words, a block can treated as lexical closures. The following example creates Proc object in counter method, and this Proc object binds the argument *num* and bounded environment can be referenced from

Blocks that have been converted into Proc objects can reference block local variables, the method local variables always. That Proc object has a property as lexical closures. For example, in the following program, counter method generates Proc object that binds the argument *num*, constrained environment can be referenced from outside of the method.

```
def counter(num)
  Proc.new {
    num += 1
  }
end
c = counter(0)
c.call # -> 1
c.call # -> 2
c2 = counter(0)
c2.call # -> 1
```

Block is used in various situations in the Ruby program. And a block is used in a variety of scenes in the Ruby program to implement iteration. Ruby programmer prefers using a block to implement iteration to using while statement or until statement. [1] For example, we use a block to implement iteration of the reputation for each element of the array object or range object using each method defined in each class. Each method invokes given block for each element.

```
[1, 2, 3].each{|e| ... }
(1..3).each{|e| ... }
```

The following example shows the definition Range.each method and usage of Range.each method.

```
# Range class and Range.each method
class Range
def each(&block)
  i = self.begin
  last = self.end
  while (i < last)
    block.call(i)
    i+=1
  end
end
end
# the sum from 1 to 20
sum = 0
(1..10).each {|i| sum += i }
(11..20).each {|j| sum += j }
```

Figure 4.1: Definition of Range.each method and a toy program which use block

Garbage collection

In Ruby, explicit memory management is not required. Unused objects are collected automatically by garbage collection.

Dynamic code loading

In Ruby, the definition of classes and the definition of methods are performed at runtime

[1] for statement is available in Ruby. And for statement is a syntactic sugar for invoking each method.

and both definitions can redefine at runtime. Furthermore, Ruby has an **eval** method that evaluate the string as Ruby program at runtime.

In Ruby, there are three types of eval methods. First, the class_eval method is the eval method for classes. The class_eval method evaluates the program in the context of the class expression. The second one is instance_eval method that evaluates the program with specified object. And the third one is the eval method that evaluates the program with specified runtime environment. The following example shows the usage of each three eval methods.

```
puts 10 + 20 # -> 30
# re-define Fixnum.+ method using class_eval
Fixnum.class_eval <<-TEXT
def +(n)
  self - n
end
TEXT
puts 10 + 20 # -> -10

n = "Hello"; m = "World"
# define String.foo method using eval
eval <<-TEXT
def String.foo(n)
  self + n.to_i
end
TEXT

puts n.foo 2 # -> Hello2
puts m.foo 2 # -> Hello2
# define instance-specific method n.foo
# using instance_eval
n.instance_eval <<-TEXT
def self.+(n)
  self * n.to_i
```

```
end
TEXT
puts n.foo 2 # -> HelloHello
puts m.foo 2 # -> Hello2
```

These toy programs will not be used in a practical program. However, it is necessary to implementing meta-programming feature such as rewriting already defined methods and adding logging function.

Extension library
Ruby has many libraries and using these libraries, the programmers easily implement complex software. CRuby has a feature to use a library written in the C language from Ruby. This feature enables the programmer to implement connecting database or handling images easily.

4.1.4 Ruby implementation

The development of the implementation of Ruby is started from 1993 by Yukihiro Matsumoto. Also, many developers and researchers have been tackled to speed up the Ruby implementation. In this section, we describe the Ruby implementations to improve the performance.

CRuby

The reference Ruby implementation is called CRuby, which is written in C language. Current stable version is version 2.2.

In an early stage of development, CRuby used an interpreter that traverse abstract syntax tree and evaluates each node. And CRuby uses its conservative mark-sweep garbage collector for memory management.

In addition to this, CRuby has high portability and the implementation of CRuby considered to use assembler not as much as possible. CRuby uses assembler to implement garbage collector, and this assembler code is used to support the architecture-depended feature.

Ruby can run on Linux, Mac OSX, Windows, and FreeBSD. However, the feature that strongly depends on the environment is not available or limited.

CRuby is also characterized to be able to write the extension library written in C language easily. To utilize the capabilities of Ruby implementation, CRuby has Ruby C API. Using Ruby C API, the functions written in C language can invoke from Ruby program as a Ruby method. This method that has been written in the C language and calls from Ruby program is called the native method.

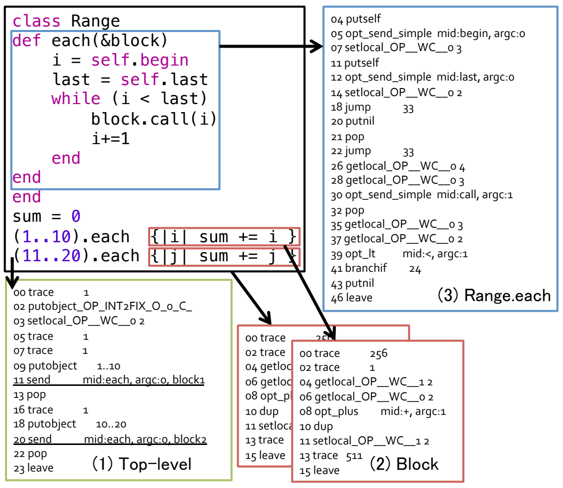
Libraries for extending Ruby using Ruby C API is called C extension library. In CRuby, as mentioned above, there are many C extension libraries (including handling image processing library, database) to implement image processing system or database management system.

Since the program size of Ruby program has been increases, the speed of the interpreter becomes the bottleneck. To solve this problem, the some developers implement the bytecode interpreter[51, 44], and other developers implement another implementation of Ruby to tackle this problem[31, 28, 41, 42].

And since CRuby 1.9, CRuby uses a stack-based virtual machine interpreter, called YARV. YARV compiles the Ruby program into YARV bytecode at runtime. To compile the Ruby program, YARV divides the program into four code fragments (top-level expressions, class definitions, methods, and blocks) and each code fragments are treated as the basic unit for compilation. For example, YARV compiles the Ruby program shown in Figure4.1 1 to bytecode shown in Figure4.2. In Figure4.2, each bytecode indicates (1) top-level expression, (2) two blocks, (3) each method. Each bytecode is executed in order from the bytecode of top-level expression. And bytecode for "Range.each" method is executed by "send" instruction at the top-level expression. At this point, each block is passed as an argument to a method.

Other Ruby Implementation

In recent days, to improve the execution speed of CRuby, JRuby uses Java languages to construct the Ruby implementation. JRuby is built on Java Virtual Machine[32] and can work in the environment that works other Java applications. And IronRuby[28] is another Ruby implementation for .NET framework. IronRuby uses the Dynamic Language



```
class Range
def each(&block)
    i = self.begin
    last = self.last
    while (i < last)
        block.call(i)
        i+=1
    end
end
end
sum = 0
(1..10).each  {|i| sum += i }
(11..20).each {|j| sum += j }
```

```
00 trace        1
02 putobject_OP_INT2FIX_O_0_C_
03 setlocal_OP__WC__0 2
05 trace        1
07 trace        1
09 putobject     1..10
11 send          mid:each, argc:0, block1
13 pop
16 trace        1
18 putobject     10..20
20 send          mid:each, argc:0, block2
22 pop
23 leave
```

```
00 trace        256
02 trace        1
04 getlocal_OP__WC__1 2
06 getlocal_OP__WC__0 2
08 opt_plus      mid:+, argc:1
10 dup
11 setlocal_OP__WC__1 2
13 trace   511
15 leave
```

```
04 putself
05 opt_send_simple  mid:begin, argc:0
07 setlocal_OP__WC__0 3
11 putself
12 opt_send_simple  mid:last, argc:0
14 setlocal_OP__WC__0 2
18 jump         33
20 putnil
21 pop
22 jump         33
26 getlocal_OP__WC__0 4
28 getlocal_OP__WC__0 3
30 opt_send_simple  mid:call, argc:1
32 pop
35 getlocal_OP__WC__0 3
37 getlocal_OP__WC__0 2
39 opt_lt        mid:<, argc:1
41 branchif       24
43 putnil
46 leave
```

Figure 4.2: Bytecode for Figure4.1

Runtime[14] that is the library to support to implement the dynamic-typed language that is built on Common Language Runtime[13].

The advantages of these implementations are that they can utilize these existing programming language environment and can be utilized rich program resources available in Java or C#. Moreover, because the features of Ruby such as threads and garbage collection are available in these language environments, there is no need to implement newly.

On the other hand, in this implementation, there is a disadvantage point such as the resolution of the gap between sthese programming environments and the Ruby language is difficult or impossible.

For example, Java virtual machine provides functions to enough to run the Java language. However, it requires a significant cost to implement the feature that is far away from the feature that can be written in Java language. In addition to these implementations, Rubinius[41] and Topaz[21] have been developed independently, respectively. Rubinius is a meta-circular interpreter of Ruby language and is mainly written in Ruby. Rubinius convert Ruby program into own intermediate language and convert it to native code using LLVM compiler framework[35]. And Topaz is another implementation of Ruby language written in RPython, a subset of Python programming language. Topaz project uses PyPy[9] to derive a trace-based JIT compiler.

4.2 Problem

In this section, we describe the problems to building a high-performance Ruby implementation.

As mentioned above, Ruby is a dynamically typed object-oriented programming language. In the Ruby program, because of the feature of dynamic typing, the programmers need not to specify the type of variables, methods, and expression. And this is why Ruby program is difficult to analyze the types precisely in a short time. The dynamic type system needs runtime overheads include a runtime type checking and dispatch the execution by the types.

In addition to these overhead, in Ruby, the definition of classes and methods are executed at runtime and as described in Section 4.1.3, the Ruby programmer can re-define classes and methods at runtime. This *eval* feature makes it difficult to analyze before executing the Ruby program. Because there is no guarantee that will not be re-defined methods, we cannot simply adopt the traditional compiler optimization techniques (e.g. constant folding and loop invariant code removal).

The next chapter discusses the techniques used in this work to implement a high-performance Ruby implementation.

4.3 Implementation

In this chapter, we provide the implementation details of RuJIT, our trace-based JIT compiler for CRuby. the source code of RuJIT can be found at `https://github.com/imasahiro/rujit`. RuJIT is based on existing implementation of YARV, the interpreter of CRuby, and we use CRuby's runtime other than YARV without any modification. RuJIT records the execution path that YARV performed and constructed a trace from the path. RuJIT does not use YARV bytecode but using newly designed low-level intermediate representation, called LIR to construct the trace. And RuJIT adopts compiler optimizations and after that, RuJIT generates machine code.

4.3.1 Overview of RuJIT

We implemented a new software component, a trace-based JIT compiler into YARV. Figure4.3 is an overview of the entire system. RuJIT contains four components; (1) trace selection engine that forms and dispatches a trace. (2) IR generator, which convert from YARV bytecode into an IR (intermediate representation) of RuJIT. (3) Optimizer that optimize LIR instructions. (4) Native code generator. (5) Trace cache that manages compiled traces.

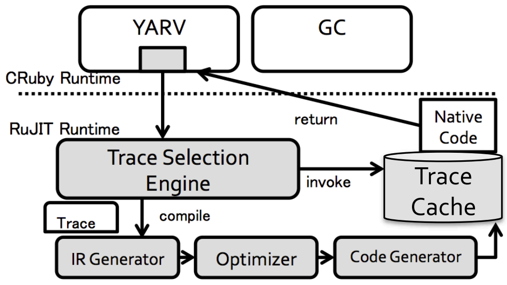


Figure 4.3: A overview of RuJIT

The tracing selection engine is driven by the execution of bytecode at YARV. We modified YARV to call the tracing selection engine at each bytecode execution. The trace selection engine uses NET strategy to selecting which sequence of bytecode compiles, record execution path, and invoke compiled traces.

When RuJIT detects potential trace head by using NET strategy, RuJIT is switched into record mode. In the recording mode, the tracing selection engine records all execution information includes a result of method search and type checking. This information is stored as a part of an intermediate representation.

When the trace selection engine formed a trace, we optimize a trace. We use following optimization set to optimize a trace, constant folding, and invariant code hoisting, and optimization described in Section 4.4. After an optimizer optimizes a trace, RuJIT compiles a trace, it puts the entry point address of the compiled code in the trace cache and the process of the compiling a trace is completed.

Next section shows a concrete example of the trace compilation.

4.3.2 Compiling YARV to RuJIT IR

When RuJIT is on the record mode, RuJIT records the execution path of YARV bytecode instructions and translates to LIR instructions. In this section, we describe how to translate YARV bytecode instructions to LIR instructions and translate LIR instructions to machine code using a running example. Our example program computes the 1st-100th number in Fibonacci sequence using a loop.

```
a, b = 0, 1
while b < 100
    a, b = b, a+b
end
```

Figure 4.4: Toy program that computes the 1st - 100th number in Fibonacci sequence using loop.

```
0002 putobject 0
0003 putobject 1
0004 setlocal a
0006 setlocal b
0010 putobject 100
0012 setlocal x
0016 jump 36
0018 putnil
0019 pop
0020 jump 36
*0024 getlocal a
*0026 getlocal b
*0028 getlocal a
*0030 opt_plus <callinfo!mid:+, argc:1>
*0032 setlocal a
*0034 setlocal b
*0036 getlocal a
```

```
*0038 getlocal x
*0040 opt_lt <callinfo!mid:<, argc:1>
*0042 branchif 24
0044 putnil
0045 leave
```

At first, RuJIT compiles this program into YARV bytecode instructions.

During the execution of these bytecode instructions, RuJIT detects that the bytecode instruction on line 11 as a potential trace head. In record mode, RuJIT records the execution path of the bytecode on line 11-20 and generate the following trace.

```
Trace1:
 *BasicBlock BB0
 local variables:
     []
 code:
  0000 Jump TargetBB:bb:1

*BasicBlock BB1
 local variables:
     [(0, 3)=0019,(0, 4)=0018,(0, 2)=0021]
 code:
  ## getlocal a
  0002 EnvLoad Level:0 Index:3
  0003 StackPush Val:0002
  ## getlocal b
  0004 EnvLoad Level:0 Index:4
  ## getlocal x
  0021 EnvLoad Level:0 Index:2
  ## opt_plus <callinfo!mid:+, argc:1>
  0005 StackPush Val:0004
  0007 StackPush Val:0002
  0008 GuardTypeFixnum pc:0x7feee35630a0 R:0004
  0009 GuardTypeFixnum pc:0x7feee35630a0 R:0002
  0010 GuardMethodRedefine pc:0x7feee35630a0 Fixnum.+
  0025 GuardMethodRedefine pc:0x7feee35630a0 Fixnum.<
  0011 StackPop
  0012 StackPop
  0013 FixnumAddOverflow LHS:0004 RHS:0002
  ## setlocal a
  0016 EnvStore Level:0 Index:3 Val:0013
  0017 StackPop
  ## setlocal b
  0018 EnvStore Level:0 Index:4 Val:0002
  ## opt_lt <callinfo!mid:<, argc:1>
  0020 StackPush Val:0016
  0022 StackPush Val:0021
  0023 GuardTypeFixnum pc:0x7feee35630f0 R:0016
  0024 GuardTypeFixnum pc:0x7feee35630f0 R:0021
  0026 StackPop
  0027 StackPop
  0028 FixnumLt LHS:0016 RHS:0021
  ## branchif 24
  0031 GuardTypeNil pc:0x7feee3563110 R:0028
  0032 Jump TargetBB:bb:1
 snapshot:
    VM stack snapshot at pc=0x7feee35630a0: [0002, 0004, 0006];
    VM stack snapshot at pc=0x7feee35630f0: [0019; 0021];
    VM stack snapshot at pc=0x7feee3563110: [];
---------------
```

The recorded trace (Trace1) consists from two basic blocks (BasicBlock BB0 and BB1) and contains three exit points and VM stack snapshots. And we realize that this trace only covers the execution path of integer addition and integer comparison, and when we execute any other execution path, guards fail.

After optimization, the trace is translated to machine code. As mentioned above, RuJIT uses C compiler as a native code generator. RuJIT first translates the trace to C code and then translate to machine code. The following code show how the LIR is translated to C code.

```
trace_side_exit_handler_t *
ruby_jit_0(rb_thread_t *th, rb_control_frame_t *reg_cfp)
{
  VALUE v2, v4, v21, v11, v12, v13;
  VALUE v16, v18, v28;
L_0:
  // 0000 Jump TargetBB:bb:1
  goto L_1;
L_1:
  // 0002 EnvLoad Level:0 Index:3
  v2 = *(GET_EP() - 3);
  // 0004 EnvLoad Level:0 Index:4
  v4 = *(GET_EP() - 4);
  // 0021 EnvLoad Level:0 Index:2
  v21 = *(GET_EP() - 2);
  // 0008 GuardTypeFixnum pc:0x7feee35630a0 R:0004
  if(!(FIXNUM_P(v4))) goto L_exit0;
  // 0009 GuardTypeFixnum pc:0x7feee35630a0 R:0002
  if(!(FIXNUM_P(v2))) goto L_exit0;
  // 0010 GuardMethodRedefine pc:0x7feee35630a0 Fixnum.+
  if (!JIT_OP_UNREDEFINED_P(0, 1)) goto L_exit0;
  // 0025 GuardMethodRedefine pc:0x7feee35630a0 Fixnum.<
  if (!JIT_OP_UNREDEFINED_P(7, 1)) goto L_exit0;

  // 0013 FixnumAddOverflow LHS:0004 RHS:0002
  v13 = rb_jit_exec_IFixnumAddOverflow(v4, v2);
  // 0016 EnvStore Level:0 Index:3 Val:0013
  *(GET_EP() - 3) = v13;
  v16 = v13;
  // 0018 EnvStore Level:0 Index:4 Val:0002
  *(GET_EP() − 4) = v2;
  v18 = v2;

  // 0023 GuardTypeFixnum pc:0x7feee35630f0 R:0016
  if(!(FIXNUM_P(v16))) goto L_exit1;
  // 0024 GuardTypeFixnum pc:0x7feee35630f0 R:0021
  if(!(FIXNUM_P(v21))) goto L_exit1;
  // 0028 FixnumLt LHS:0016 RHS:0021
  v28 = rb_jit_exec_IFixnumLt(v16, v21);
  // 0031 GuardTypeNil pc:0x7feee3563110 R:0028
  if(!(RTEST(v28))) goto L_exit2;
  // 0032 Jump TargetBB:bb:1
  goto L_1;
// side exits
L_exit0:; // pc=0x007ff6f352a4c0
   SET_PC(0x007ff6f352a4c0);
    PUSH(v4);
    PUSH(v2);
    return TRACE_GUARD_FAIL;
L_exit1:; // pc=0x007ff6f352a510
   SET_PC(0x007ff6f352a510);
    PUSH(v16);
    PUSH(v21);
    return TRACE_GUARD_FAIL;
L_exit2:; // pc=0x007ff6f352a530
   SET_PC(0x007ff6f352a530);
    return TRACE_GUARD_FAIL;
}
```

Finally the generated machine code and the recorded trace are stored in the trace cache, and the compilation process is completed.

4.3.3 Trace Selection Engine

The trace selection engine is driven by the execution of each bytecode at YARV. Along the algorithm shown in Figure4.5, a trace selection engine selects and invokes a trace. Our trace selection algorithm uses NET strategy. It first determines potential trace head and then records the next execution starting from the trace head as trace.

A trace selection engine first selects a candidate of a trace head with following two conditions. (Figure4.5 Lines 15-16, 11-12)

- backward branch
- the exit point of an already formed trace

The first condition indicates that each time a backward branch instruction is executed, the target of a taken backward branch is nominated as a trace head. This condition approximately detects a loop without building a control flow graph. The second condition indicates that when an execution of a compiled trace is aborted because guards (checks) for validation for the speculation that is constructed at compile time, is failed the resume point from a trace is nominated as a trace header. In this way, the trace selection engine can record uncovered execution path and which achieve high coverage for the JIT compiled code.

The trace selection engine assigns a counter for each nominated trace heads and records execution count of each potential trace heads. When this counter reaches a predefined threshold, the trace selection engine starts to record a trace. (Figure4.5. Lines 17-21)

4.3.4 Trace Recording

In the recording mode, the trace selection engine records all bytecode instructions executed in YARV, including type checking, method invoking, and branch instruction. This information is recorded to the trace as IR and type checking and invoking method is converted to a guard instruction. Figure4.6 shows how the trace selection engine convert opt_plus instruction into IR.

To reduce the overhead of driving the trace execution engine at non-recoding mode, we add new YARV instruction, which embedded the process for recording the execution path. (Figure4.6 lines 12-30) And when the trace selection engine becomes the recording mode, YARV switches from the interpreter mode (Figure4.6 lines 1-11) to the record mode to record a trace. Note that R_recv, R_obj are variables for RuJIT's IR.

```
void trace_selection(Program Counter PC) {
    if (mode == Record) {
        if (Satisfy dtrace termination condition) {
            mode = Default;
            Compile trace
        }
        return;
    }
    if (code = FindCompiledCache(PC)) {
        Invoke compiled trace
        if (Exit from side exit)
            Mark PC as trace head
        return;
    }
    if (isBackwardBranch(e))
        Mark PC as trace head
    if (counter = FindTraceCounter(PC)) {
        counter += 1;
        if (counter > Threshold)
            mode = Record;
    }
    return;
}
```

Figure 4.5: Trace selection algorithm

Trace Termination Condition

The trace selection engine records all YARV bytecode until one of the following trace termination conditions is satisfied.

- a loop is found. i.e., execution reaches a trace head.
- the start of an existing trace is found.
- an exception is thrown.
- the recorded trace has become too long.

```
case opt_plus:
   VALUE recv = POP();
   VALUE obj = POP();
   if(typeof recv is Fixnum && typeof obj is Fixnum) {
      if(Fixnum.+ is not redefined)
         val = recv + obj;
      else
         Invoke Fixnum.+
   } else { ... }
   PUSH(val);
   Dispatch next instruction
case opt_plus_on_record:
   VALUE recv = POP();
   VALUE obj = POP();
   VALUE val;
   if(typeof recv is Fixnum && typeof obj is Fixnum) {
      EMIT(GuardFixnum, R_recv);
      EMIT(GuardFixnum, R_obj);
      if(Fixnum.+ is not redefined) {
         EMIT(GuardRedefine, Fixnum.+)
         val = recv + obj;
         EMIT(FixnumAddOverflow, R_recv, R_obj);
      }
      else
         EMIT(GuardMethod, type of recv, R_recv)
         EMIT(InvokeMethod, recv.+, R_recv, R_obj)
         Invoke Fixnum.+
   } else { ... }
   PUSH(val);
   Dispatch next instruction
```

Figure 4.6: Bytecode instruction handler for opt_plus

- call native method that we cannot include a trace

First condition stops the recording when a cyclic path is detected in the recorded trace. The trace selection engine detects repetition when current program counter (PC) is already recorded. And second condition stops the recording when current PC is same as the trace head of already recorded traces.

From the point of view of performance, our trace selection engine uses a fixed length buffer to record execution path. Third condition ended the recording because this buffer is overflowed.

Third and Forth conditions stop the recording because YARV executes the instruction that the trace selection engine cannot trace throwing an exception or calling native methods. Note that to improve the coverage of the JIT-compiled code, we allow to include a part of native method in traces. See Section 4.4.2 for details.

After the trace selection engine finish recording, code generator compiles a recorded trace into native code (Figure4.5. line 5). And a compiled trace is putted in the trace cache with the entry point address of the instruction of the compiled trace. Once the address of trace head of the compiled trace becomes available in the trace cache, the trace selection engine invokes the compiled trace when the execution reaches the instruction of trace head (Figure4.5 lines 9-14).

4.3.5 RuJIT Low-level IR (LIR)

RuJIT uses the simple register-based low-level IR and does not use YARV bytecode directly.

RuJIT uses low-level IR, called LIR, to record each instruction that the trace recorder recorded. YARV uses a stack-based high-level IR to represent Ruby program and focus to ease of implementation and memory usage. But RuJIT newly designed register-based low-level IR, called LIR, to represents the behavior of each executed YARV bytecode instructions. Each LIR instructions store the opcode, the result type, and one or more operands.

Table 4.1 lists all LIR instruction that RuJIT supported. The first group of instruction (e.g. GuardTypeFixnum, GuardClassMethod) represents guards, and these instructions hold the snapshot of the VM stack at the time of the source YARV bytecode instruction executed. And if the guards fail, RuJIT restores the VM stack using the snapshot and execution leaves the trace and falls back to the interpreter.

The second group of instruction (e.g. FixnumAdd, FloatSub, StringAppend) represents basic operation such as simple arithmetic and string operations. Because LIR instruction set is based on YARV bytecode instruction set, LIR instruction contains the fine-grained version of YARV bytecode instruction.

The third group of instructions represents the constant values. And the fourth group of instructions represents method call and unconditional branch. Method invocation in

CRuby consists of two parts, the dynamic lookup and invoking method. RuJIT performs the dynamic lookup at the time of recording the trace and emits the result as a guard instruction; then it emits the instruction for method invocation.

The fifth group of instructions represents the VM stack operation. In general, we avoid writing the value back to the VM stack. On the other hand, when we invoke a method that CRuby from a trace, we need to maintain the consistency of the VM stack between the interpreter and a trace. We use StackPush and StackPop instructions to maintain the consistency of the VM stack.

The final group of instruction is meta instructions that correspond to the optimization. PHI instructions correspond to phi instructions from static single assignment (SSA) form[15]. Because some optimizations such as loop-invariant code motion, RuJIT use, are assumed that the target of optimization is SSA form, we use PHI instruction to encode the semantics in SSA form.

4.4 Trace Optimizations

After RuJIT completes the recording of the trace, RuJIT performs optimizations that include constant folding, duplicated guard removal, dead code elimination, and heap allocation removal using escape analysis[12]. In addition to this optimization, to optimize the trace more aggressively, RuJIT performs type inference to provide the concrete types and shapes of program variables.

The following section describes two optimizations that we adapt to RuJIT and our type propagation system.

Constant Folding

In RuJIT, we use constant propagation to optimize the expression that can be computed at compile time. RuJIT has prepared patterns of instruction sequences, and RuJIT compare these patterns and the instruction sequence, to perform the optimization. Table 4.2 shows the part of the pattern of instruction sequences that RuJIT perform constant propagation.

4.4.1 Redundant guard removal

In common case, a recorded trace contains many guards. Redundant guard removal removes guards that have already appeared and thus redundant do not to be checked again.

Following example code shows redundant guard removal can remove a guard.

The guard *g3* on line 6 can be removed since the type of v3 is known. And the guard *g2* on line 5 can be removed since it is identical to the guard on line 2.

```
g0 = GuardTypeString v1
g1 = GuardTypeString v2
v3 = StringAdd v1, v2

g2 = GuardTypeString v1
g3 = GuardTypeString v3
v4 = StringAdd v1, v3
```

In RuJIT, we adopt this optimization to type checking guard and object property guard to remove the redundant guard.

Escape Analysis

Reducing the number of the objects allocating on heap leads lower runtime overhead of garbage collection. If the objects are used only the trace and there is no other outside reference to them, we can remove the allocation of these objects. This optimization is based on escape analysis[12].

As a result, we can assign the object as a stack-allocated variable or scalar value, and we obtain the advantage such as increasing the opportunity to perform other optimization.

We implement this optimization using a standard data flow analysis. We optimistically removing every memory allocations and only perform them when leaving the trace or the point that the trace needs to create a reference to the object.

The following code illustrates how we rewrite the trace. If an object is allocated on the trace, but may escape from the trace because the object is referred at side exits, and then we inserts an object allocation at side exits.

```
# (A) before escape analysis and stack allocation
bb0:
  c0 = LoadConstFixnum 10
  c1 = LoadConstFixnum 20
  v0 = AllocArray c0, c1
  ...
  g2 = GuardMethod Array.length # goto side_exit:
  v1 = ArrayLength v0

side_exit:
  v2 = StackPush v0 # escape v0
  TraceExit

# (B) after escape analysis and stack allocation
bb0:
  c0 = LoadConstFixnum 10
  c1 = LoadConstFixnum 20
  v0 = AllocArray c0, c1
  ...
  g2 = GuardMethod Array.length # goto side_exit:
  v1 = ArrayLength v0

side_exit:
  v2 = AllocArray v0 # escape v0
  TraceExit
```

Type Propagation

To detect non-trivial case that redundant guard removal can remove and to support additional type specialization, we adopt type inference to determine the types of various variables in the trace.

We performed type propagation using a standard forward data flow formulation. This type propagation is achieved by predicting what types of LIR instructions will produce, type checks based on guard instructions in two steps.

Figure4.7 shows the type lattice used by the type inference algorithm.

The least precise type is Object, which represents the Ruby Object type, and all symbols belong before type inference. The most precise type is *untyped* but unsound. We first initialize the return type of each instruction to untyped. And because the global variables are not supported in the current implementation, the types of global variables are initialized to Object.

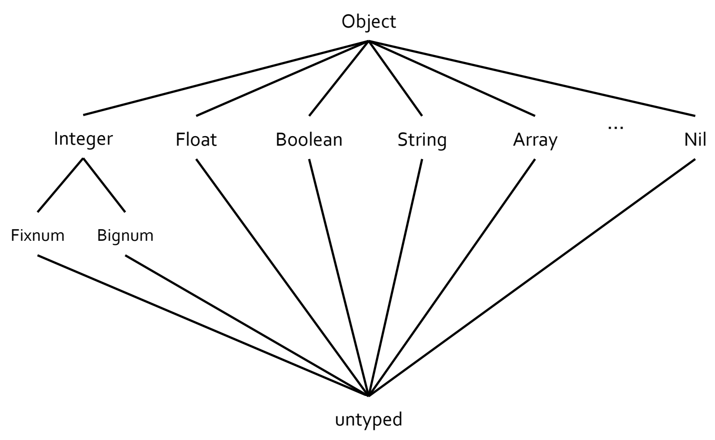


Figure 4.7: Type lattice used by our type inference algorithm

Next, we propagate the type of each instruction using a standard forward iterative dataflow algorithm that stops when we have reached a fixed point. We compute a new type of the output of each instruction based on the type of its inputs, such as guard, function call, load/store operation of local variables, among others. The algorithm completes when the type of each instruction has not changed. Using the computed type information, RuJIT can eliminate redundant guard and support additional type specialization.

4.4.2 Trace Recording and Specialization

In this section, we describe two techniques that we introduce to reduce the runtime overhead in RuJIT.

Speculative code motion for guard for block

As mentioned in Section 2.1, Ruby programmers often use a block to implement iteration by invoking the block itself. The iteration is likely formed to a trace. To illustrate our techniques, we demonstrate how to construct traces by using Figure4.8, a simplified version of Figure4.1.

In Figure4.1, following the algorithm depicted in Figure4.5, the trace selection engine forms a trace that contains the loop between lines 2-6 and the block B1 on line 8.

```
1 def each(&block)
2   while (A)
3     B
4     block.call(i)
5     C
6   end
7 end
8 each { B1; }
9 each { B2; }
```

Figure 4.8: simplified version of Figure4.1

The formed trace has a block guard that checks two blocks equality; one block comes from an argument of each method, and another is B1, which is inlined into this trace. When this guard fails, RuJIT transfers control to YARV (Trace 1A in Figure4.9).

Because the method "each" (on line 8 and line 9) takes different blocks (including B1 and B2, respectively), executing the trace with block B2 is always aborted by the block guard for block B1. As the algorithm mentioned in Figure4.5, RuJIT forms a new trace, Trace 1B, which includes the execution path from the aborted point to the invocation of the block B2, and then links Trace 1B to Trace 1A.

The simplified example suggests that the transition between traces cause a significant overhead at runtime. To reduce this overhead, we focus on the mostly immutability of CRuby's compiled code. Note that the "mostly" means the immutability is broken if the block is modified at runtime.

In RuJIT, when a block guard aborts a trace, we form a child trace from the exit point of the aborted trace (referred to as parent). We assume that blocks contained in the parent trace are immutable. We apply an invariant code hoisting technique into a block guard to reduce the transition between a parent trace and a child trace.

We use this code hoisting technique as a part of the link-time optimization for traces. Our Invariant code hoisting is performed as the following steps. First, from blocks commonly appearing in parent and child trace, we select blocks for code hoisting that satisfy the following conditions to ensure the immutability of the instruction sequence.

1. a block which has same name lexically and same start point of the live range of block between traces.

2. parent trace, child trace, and blocks do not contain eval methods.

3. a block is not a target by rewriting from other objects when the trace is formed.

Each condition show (1) each traces inlined invocation of blocks, (2) block's sequence of instructions is not able to rewrite from others, (3) block is not rewrite from Binding which is the object can rewrite runtime context and local variables at runtime.

When we found a nominated trace, we first duplicate each trace for the purpose of deoptimization that is described later. And we replicate a parent trace from the entry point to block guard. (Figure4.9 Trace2)

Second, we apply code hoisting to block guard and hoist this guard to the start point of the live range of a block. (Figure4.9 Trace3). Finally, we obtain traces that reduce the runtime overhead of block guards from the original traces.

Note that as mentioned above, this optimization assumes that block and its instructions are immutable at runtime. We need to de-optimize these traces when an assumption is no longer satisfied. For de-optimization, we replace these traces to the original version of traces (Figure4.9 Trace 1A, Trace 1B)

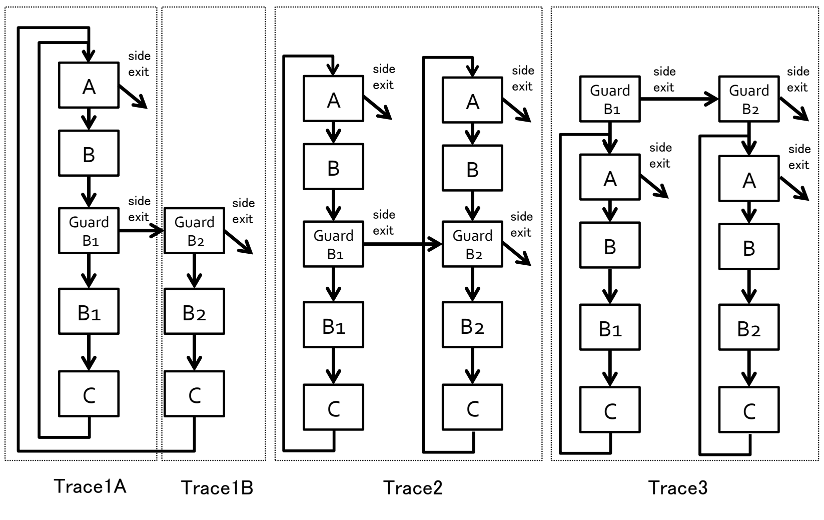


Figure 4.9: Traces with block

Native method inclusion

In CRuby, to improve the performance, some methods are written in C language and compiled to native code before runtime. Especially, in CRuby, method of core classes such as String and Array are written in C. Because these methods are not traceable by the trace selection engine, we frequently form short traces and these traces causes runtime overhead and low coverage for the JIT compiled code.

To improve the quality of the trace, we include native methods and convert iteration methods to the Ruby program.

First, we include the native method call into the traces. We modified our trace termination condition to do not stop trace recording when YARV calls native methods that are shown in Table 4.3. We select several native methods, which do not throw an exception and do not write global variables. By allowing these native method calls in the traces, we can reduce the overhead by generating longer traces. If it is possible to include this method calls to trace a longer more interval trace covers, also can reduce runtime overhead due to the transition between the trace and the interpreter.

In Ruby, to improve performance, iteration methods such as Range.each and Array.map is written in C language. We need to include these iteration methods into the traces because these methods frequently execute loops and in most case, these methods contain a trace head. For this purpose, we write native methods shown in Table 4.4in Ruby to be able to form a trace by the trace selection engine. Note that we re-implement some methods because rewriting native method often causes the performance degradation. We choose methods that are empirically frequently invoked.

4.4.3 Code Generation

After RuJIT performs optimizations, the last step of compiling process of RuJIT is to translate LIR instructions to C code. We chose C language and C compiler as our compiler backend for the following reasons. First, C language is known as one of the fastest programming languages, and it has a well-known highly optimized compiler. RuJIT uses GCC[17] to compile the generated C code and uses many optimizations that GCC has. And the second reason is that CRuby uses C language to implement itself, and we can share CRuby and

RuJIT implementations.

RuJIT translates LIR instructions to C code. Generated C code contains one function for initializer and one or more functions for trace body.

- Initializer: Initializer establishes a connection with the trace and CRuby runtime. Compiled traces share CRuby runtime with RuJIT. This runtime includes native functions, methods, classes, and constants.

- Implementation: This function has the implementation of a converted trace. Each of generated code contains one function that is translated from LIR instruction directly and one or more functions that encode trace side exits.

The initializer function first initializes runtime information about RuJIT. The LIR instructions are represented as the C function. This function takes the runtime context of YARV. If a guard fails, it writes back local variables and temporary variables appear in the trace, and the execution falls back to YARV.

Most LIR instructions use the functions and the macros that are used in the implementation of YARV. Sharing implementation of LIR and YARV leads the improvement of maintainability, and it reduces the development costs.

Note that at the time the early stage of development of RuJIT, we use only one thread to record the trace and generate the machine code. However, when RuJIT construct a large number of traces, because RuJIT generates the machine code using C compiler and invoking C compiler leads huge overhead, we found that the machine code generation cause a significant overhead at runtime.

To avoid this overhead, we run the machine code generation in parallel with the interpreter. That is, we prepared a dedicated thread for the machine code generation, and we avoid the overhead due to the invocation of the C compiler.

When we are compiling a large number of recorded traces, it had been five times slower than YARV. After the introduction of the dedicated thread for code generation, the ratio of performance degradation comes to 1.2-2 times. We need to reduce more compilation overhead because our JIT compiler still has performance degradation. As future work, to reduce this overhead, we will plan to replace current code generator to our native code generator described in Chapter 3.

4.5 Evaluation

This section evaluates RuJIT by comparing it to YARV. We evaluate the performance of Ruby applications from the CRuby benchmark suite.

4.5.1 Experimental Setup

We ran benchmark application with the experimental environment listed below.

- CPU : Intel(R) CoreTM i7 2.2GHz
- Memory : 8GB, 1333MHz
- OS : MacOSX 10.9.5
- C compiler : Clang 3.5

We compare RuJIT to CRuby version 2.2-preview, and RuJIT uses the same version of CRuby. To study the impact of the optimizations that are described above, we use clang c compiler to translate IR from native code. Note that we use optimization level O3 for clang.

To evaluate the performance, we use 36 micro-benchmarks from the CRuby benchmark suite.

We first compare the performance of RuJIT and YARV. Next, we evaluate the performance improvement by optimizations described in previous chapter. And finally, we compare the performance improvement by type inference and the effectiveness of type inference.

We use elapsed time that took to run the benchmark program by each system and the elapsed time for RuJIT contains execution time for selecting trace and the execution time of compilation.

4.5.2 Performance comparison

Figure4.10 shows the relative speed up of different RuJIT configuration with respect to YARV. Throughout this section, we abbreviate the optimization techniques mentioned at

Section 4.4.2 as BLK, mentioned at Section 4.4.2 as MTD and type inference as IT. We indicate the optimizations are enabled using +.

In Figure4.10, we see a speedup of 1.5 times through 5 times. And In app_strconcat, so_benchmark of so_random, our optimization techniques improve the coverage of JIT compiled code and most code are compiled into native code and we see a speedup of 2 times through 5 times.

In contrast, in so_reverse, so_complement, and app_lc_fizzbuzz, RuJIT do not speedup these benchmarks because these benchmarks heavily rely on operations that are not able to contain to a traces such as file operation or closures. In addition to this, in some benchmarks such as app_fibo, RuJIT is slower than YARV because these benchmarks rely on recursive call, and RuJIT currently do not support these features.

Note that in this experiment, we use clang for a native code generator of RuJIT. Compiling with clang contains the overhead such as translation IR to C program, read header files, etc. Execution time of the benchmark programs used in this experiment contains these overheads, and it is about 3% of the execution time on average. However, when we use a tiny program for benchmark, we cannot ignore the overhead due to invoking clang.

4.5.3 Trace Length and Transition

Table 4.5 shows that RuJIT with MTD and RuJIT with MTD+BLK configurations with the average of the length of traces and Table 4.6 shows the average of number of transition between YARV and traces. Note that the length of traces is the number of instructions contained in traces.

We see that, with RuJIT +MTD configuration, we take longer traces as compared to the default configuration of RuJIT in benchmarks that contain invoking block and invoking native method such as so_nested_loop, so_matrix. When we see the number of transition in these benchmarks, we observed that some benchmarks are increased the number of transition and others are decreased. This is because, applying MTD obtains longer trace but introduce more transitions between interpreter and traces.

With RuJIT +MTD+BLK configuration, we see that benchmarks that contain invoking blocks reduce the number of transition. On the other hand, the averages of trace length do

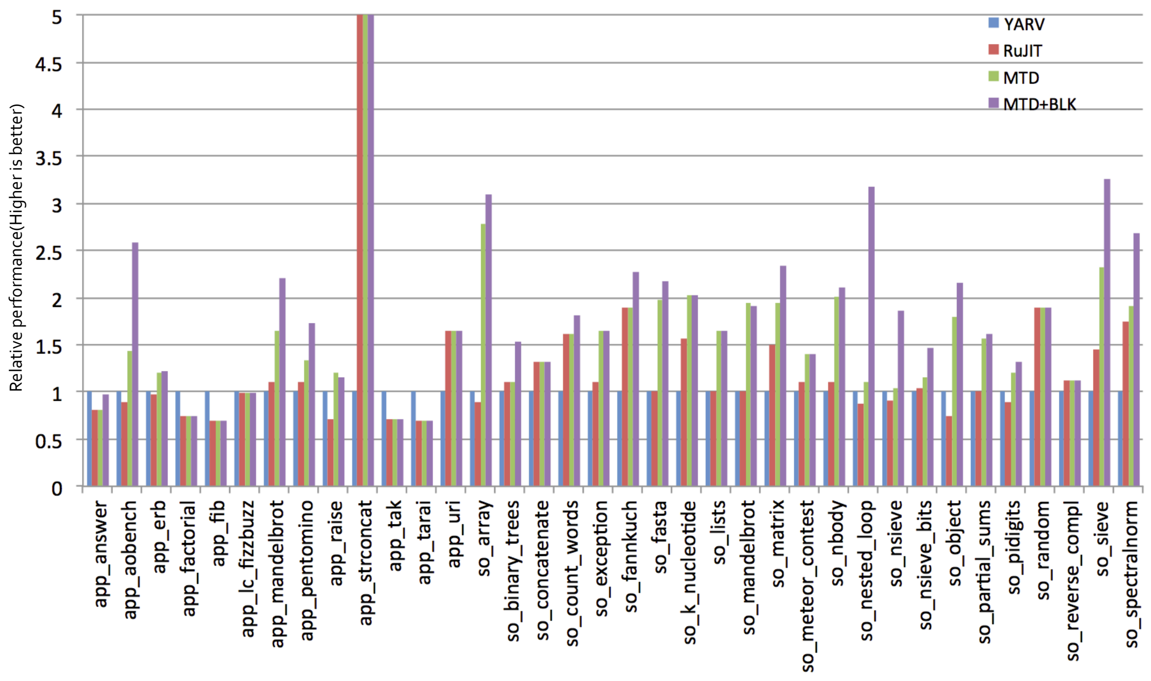


Figure 4.10: Performance comparisons between RuJIT and YARV

not change.

Note that RuJIT uses simple counters to detect hot spots from a program. And a threshold for hotness counter determines when a particular region of code is considered hot. In this section we show the relationship between the thresholds, program performance, and the number of traces that are created.

In RuJIT, we uses two hotness thresholds. One for detecting hot trace and one for detecting cold trace. Using the hot trace threshold, when the number of execution of potential trace head is exceeded the threshold and RuJIT promotes a potential trace head to trace head and starts trace recording. The cold trace threshold determines when the behavior of the program changes and cold trace can detach.

Figure4.11 compares the relative speed up for each of our benchmarks compared to our default thresholds. We select following benchmarks (so_nbody, app_aobench, nested_loop, erb, so_list and nested_loop , and mandelbrot) from Ruby benchmark suite and all benchmarks were compiled with our optimizations. We tested three hot trace threshold Th_h = (1, 2, 16) and three cold trace thresholds Th_c = (0, 32, 50). We normalized performance of each benchmark to our default configuration $(Th_h, Th_h) = (2, 32)$.

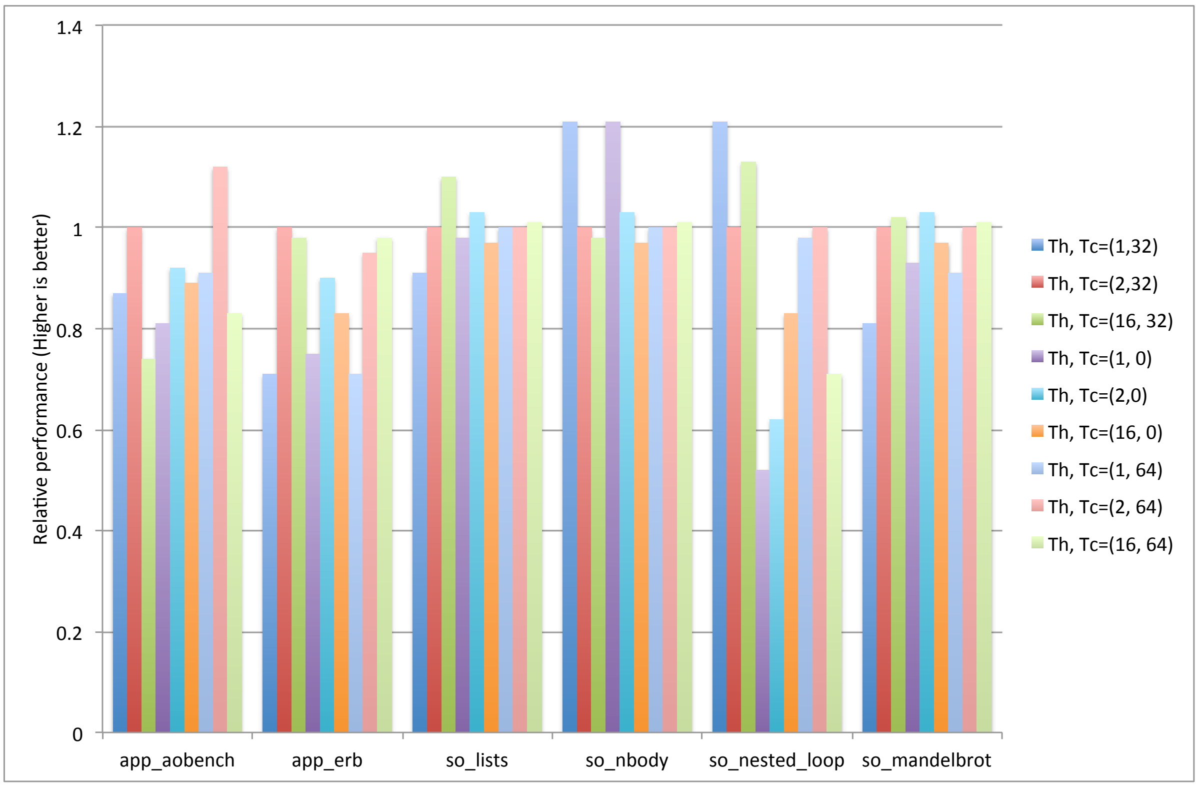


Figure 4.11: Performance comparisons between different hot trace and cold trace thresholds

The benchmarks are sensitive to the hot trace threshold. And most of benchmarks in Ruby benchmark suite are not sensitive to cold trace threshold because these benchmarks construct only few traces and these traces cover most of the execution. And the performance of nested_loop benchmarks is heavily affected by both thresholds because If the hot trace threshold is small, RuJIT generates more trace speculatively and more transition between compiled trace and the interpreter.

Note that we need to clear optimal configuration of both thresholds and how to implement each methods because these configuration determine the performance of each application. However, there are no clearly optimal thresholds. Our default of configurations including method implementation are appears from a reasonable choice but additional benchmarks may suggest better configurations in the future.

4.6 Summary

In this chapter, we showed the design and implementation of our trace-based JIT, called RuJIT built on top of CRuby interpreter. We show RuJIT achived on average about 2 times faster than CRuby interpreter. RuJIT achived better compiled code performance using two new optimization techniques to improve the quality of compiled trace. We also showed that our techniques to reduce runtime overhead improved the performance of the trace-based JIT.

Opcode	return type	Description
GuardTypeSymbol	void	Guard. The type of the given operand is Symbol.
GuardTypeFixnum	void	Guard. The type of the given operand is Fixnum.
GuardTypeNil	void	Guard. The given operand is nil.
GuardBlockEqual	void	Guard. The block given to the method currently running is equal to the block which is already recorded by RuJIT.
GuardMethodCache	void	Guard. The callee of method invocation is equal to the result of a method lookup.
FixnumAdd	Fixnum	Fixnum + Fixnum
FixnumSubOverflow	Fixnum or Bignum	Fixnum - Fixnum. This instruction consider the integer overflow/underflow.
StringAdd	String	String + String
ObjectEq	Boolean	Object == Object
GetProperty	Object	Load the property of a object.
SetProperty	Object	Store the property of a object.
LoadConstNil	Nil	Load nil.
LoadConstFixnum	Fixnum	Load fixnum constant.
LoadConstString	String	Load string constant.
InvokeNative	Object	Invoking the native method.
InvokeMethod	Object	Invoking the method.
InvokeBlock	Object	Invoking the block.
Exit	void	Exit from this trace and fall back YARV interpreter.
FramePush	Object	Push the call frame to the VM stack.
FramePop	Object	Pop the call frame to the VM stack.
StackPush	Object	Push the VM stack.
StackPop	Object	Pop the VM stack.
PHI	Object	Denotes a phi node.
NOP	void	NOP

Table 4.1: Part of the list of LIR instructions

Rule	Transformed LIR
v_1 = FixnumAddOverflow c_1 c_2	v_1 = LoadConstFixnum (c_1 + c_2)
v_1 = FloatSub c_1 c_2	v_1 = LoadConstFloat (c_1 - c_2)
c_2 = LoadConstString c_1 v_1 = StringToFixnum c_2	c_2 = LoadConstFixnum c_1.to_i
v_2 = StringAdd v_1 c_1 v_3 = StringAdd v_2 c_2	c_3 = (LoadConstFixnum (c_1 + c_2)) v_2 = StringAdd v_1 c_3
v_2 = GuardTypeFixnum *PC* c_1 Type of c_1 is Fixnum	Remove v_2.
v_2 = GuardTypeString *PC* c_1 Type of c_1 is Fixnum	Exit *PC*

Table 4.2: Part of the list of LIR constant folding rules. Constants *c* only match the instruction for constants and PC represents the exit point of the trace . Most instructions for arithmetic and comparison can optimize with constant folding. But we cannot optimize operations which have side effects including throwing exception and the VM stack operations.

Fixnum	-@, to_f, +, -, *, <, <=, >, >=, !=, ==, &, |, ^, >>, <<
Float	-@, to_i, to_f, +, -, *, /, %,>, >=, <, <=, !=, ==
Array	[], []=, <<, +, -, &, |, uniq, to_s, to_a, at, fetch, first, last, concat, push, pop, shift, unshift, insert, length, join, reverse, sort, clear, fill
String	+, *, %, [], length, size, empty?, succ, next, upcase, downcase, split, inculude?
Object	new

Table 4.3: List of Ruby methods which RuJIT includes into trace

Enumerable	find, find_all, collect, map, reduce, all?, any?, one?, none? each_with_index, inject, detect
Array	each, each_index, reverse_each, index, rindex
Fixnum	times, upto, downto

Table 4.4: List of Ruby methods which RuJIT reimplements

Name	Trace Length RuJIT	Trace Length MTD	Trace Length MTD+BLK
app_answer	0.0	23.0	23.0
app_aobench	0.0	47.6	48.4
app_erb	0.0	12.0	11.0
app_factorial	0.0	0.0	0.0
app_fibo	0.0	0.0	0.0
app_lc_fizzbuzz	0.0	0.0	0.0
app_mandelbrot	20.0	34.2	35.9
app_pentomino	8.0	17.7	18.1
app_raise	11.0	11.0	11.0
app_strconcat	14.0	14.0	14.0
app_tak	0.0	0.0	0.0
app_tarai	0.0	0.0	0.0
app_uri	14.3	65.0	70.0
so_array	0.0	37.5	37.5
so_binary_trees	0.0	84.7	84.7
so_concatenate	14.0	24.5	24.5
so_count_words	8.0	38.0	38.0
so_exception	38.0	61.0	61.0
so_fannkuch	15.0	26.0	26.0
so_fasta	0.0	41.0	42.2
so_k_nucleotide	17.0	25.2	25.3
so_lists	15.0	31.0	31.0
so_mandelbrot	0.0	57.5	59.0
so_matrix	0.0	21.6	23.1
so_meteor_contest	12.0	47.9	51.1
so_nbody	34.0	60.8	60.8
so_nested_loop	0.0	28.2	28.2
so_nsieve	11.0	27.0	27.0
so_nsieve_bits	0.0	37.1	38.5
so_object	0.0	29.3	29.8
so_partial_sums	0.0	116.0	116.0
so_pidigits	0.0	83.0	81.3
so_random	43.0	43.0	43.0
so_reverse_complement	5.0	31.3	31.3
so_sieve	0.0	30.5	30.1
so_spectralnorm	0.0	79.9	80.1

Table 4.5: Trace Length

Name	# of transition RuJIT	# of transition MTD	# of transition MTD+BLK
app_answer	0	2	2
app_aobench	0	629146	201327
app_erb	0	198	198
app_factorial	0	0	0
app_fibo	0	0	0
app_lc_fizzbuzz	0	0	0
app_mandelbrot	2209661	591884	31875
app_pentomino	1833360	833360	533360
app_raise	0	0	0
app_strconcat	2	2	2
app_tak	0	0	0
app_tarai	0	0	0
app_uri	101016	60610	59397
so_array	0	996	996
so_binary_trees	0	14286980	14286980
so_concatenate	10	10	10
so_count_words	486381	632295	632295
so_exception	249995	249995	249995
so_fannkuch	650155	845202	845202
so_fasta	0	2032838	1580781
so_k_nucleotide	32	179151	170049
so_lists	401	451873	451873
so_mandelbrot	0	2379060	1979060
so_matrix	0	26403167	1240316
so_meteor_contest	38	9500356	8935619
so_nbody	0	16801052	14801052
so_nested_loop	0	36909858	5913770
so_nsieve	8257703	9601980	9601980
so_nsieve_bits	0	13453946	8745065
so_object	0	2980035	2961110
so_partial_sums	0	1	1
so_pidigits	0	910581	611763
so_random	1	1	1
so_reverse_complement	940269	625007	625007
so_sieve	0	4903982	3286996
so_spectralnorm	0	16977280	9908118

Table 4.6: A number of transition between interprer and traces

Chapter 5

Related work

5.1 Just-In-Time Compilation

After the advent of Java, many researchers and implementor have been tackled in a number of different ways over the years. A popular research compiler for Java is Jikes virtual machine[1] and HotSpot VM[39]. It uses dynamic profiling to optimize code at runtime. Since this compiler uses a method as compilation units, this compiler uses OSR technique to replace the currently running method to new version of code.

On the other hand, it became a problem in terms of maintenance of the development of JIT compiler. SableVM[18], ShuJIT[46], OpenJIT[38] use a very simple template-based code generator to reduce the maintenance cost of developing JIT compiler.

Dynamo[5] was the first trace-based compiler to optimize programs at runtime. Dynamo formed a trace from machine code using a machine code interpreter. Dynamo only works on the PA-RISC architecture. DynamoRIO[10] adapted Dynamo's ideas and built an optimization framework for Intel hardware. RuJIT uses similar trace selection algorithm and trace management system by Dynamo.

Gal et al.[20] introduced a trace-based compiler for Java using trace trees. Their implementation is focused on low memory environment and is aimed at low memory usage than execution performance. Their Trace formation technique, called trace trees, was adapted to TraceMonkey (a JIT compiler for JavaScript).

Since TraceMonkey's appearance, a trace-based JIT compiler is widely studies as a

JIT compiler for the dynamically typed language. For example, TraceMonkey[19] and SPUR[6] for the JavaScript language, PyPy[9, 8, 3] for the Python language, and LuaJIT[40] for the Lua language uses a trace-based compilation approach. These compilers use some optimization techniques to optimize complex traces. PyPy uses escape analysis and redundant guard removal, and SPUR uses loop unrolling and invariant code motion.

PyPy[9] is an implementation of the Python programming language. Using PyPy, the programming language developers can derive an optimized interpreter from an interpreter written in RPython, a statically typed language that has similar syntax with the Python programming language. PyPy uses annotations to deriver a trace-based JIT compiler for the interpreter. Using this approach, JIT compilers for other programming languages are implemented, for example, Ruby[21], PHP[26].

SPUR is a trace-based JIT compiler for Common Intermediate Language (CLIR), and Maxpath[7] is a trace-based JIT compiler for Java. Both SPUR and Maxpath use non-optimizing compiler instead of an interpreter for baseline execution.

For Ruby, a method-based approach was studied. MacRuby[43] and Rubinius[41] have method-based JIT compiler which use LLVM compiler infrastructure[35] as backend of code generator. These system translate Ruby program to its own bytecode and convert bytecode to native code by LLVM. And Ishii et al. [29] proposed a method-based JIT compiler which extends CRuby runtime with its own code generator. Topaz is a trace-based JIT compiler for Ruby that is based on PyPy. Compared to Topaz, RuJIT is based on CRuby and is different from the point that RuJIT is able to use existing Ruby code or extension library without any modification.

Inoue et al. [27] used an existing method-based compiler for Java and added a trace-based JIT compiler. Their implementation uses similar optimization techniques which include native method invocation to traces. This approach is focused on Java and includes methods that do not invoke GC, other Java method and do not throw an exception.

5.2 Ruby Virtual Machines

A number of existing projects target the fast implementation of Ruby language. YARV(Yet Another Ruby Virtual machie)[44] is a stack machine based interpreter for CRuby. YARV applied polymorphic inline cache to reduce the cost of method searching and uses direct threading techniques in their interpreter to reduce the dispatch cost. Ishii et al.[29] built a method-based JIT compiler based on YARV. They uses devirtualization techniques to allows to replace generic method call sequences with static calls. In addition to this, to speed up the execution speed of exception in runtime library, they remove redundant invocation of setjmp function.[37]. Also, Ishii's VM seems to be an internal research project only. We have not been able to obtain it for comparative benchmarks.

Shiba et al.[45] proposed AOT compiler based on YARV. CastOff provide runtime profiling and they use profiling information to convert YARV bytecode to machine code. To reduce the overhead of invoking blocks, CastOff[45] uses inlining approach which inline block to the caller (block inlining). CastOff applies block inlining to the method which is written in C language and the programmer needs to implements the inline code by hand. In contrast, our approach is focused to Ruby program and we are able to inline block without any modification when both block and the caller is written Ruby language.

Chapter 6

Conclusions and Future Work

This thesis presented the implementations of JIT compiler for scripting languages. Concretely, the prototypes we implemented have been applied to two scripting language, first one is a method-based JIT compiler for statically-typed scripting language Konoha and second one is a trace-based JIT compiler for a dynamically-typed scripting language Ruby.

We evaluate application of the method-based and the trace-based JIT compilation to Konoha and Ruby and show that naive approach produces limited performance benefit. Then we propose a novel approach to our JIT compiler. We use domain specific knowledge including implementation of scripting language runtime and idioms which is heavily used in program. Our optimization techniques produce greater benefit in benchmark applications. As future work, we want to analyze more in depth the behavior of large scale applications to find another optimization opportunities to further improve the overall performance of the system.

6.1 Method-based JIT for Konoha

In chapter 3, we described the design and implementation of our method-based JIT compiler for a statically-typed scripting language Konoha. Then we described our three optimization techniques built on top of our JIT compiler. In our experiments, we show that these optimization techniques remove the overhead of invoking runtime library's method, remove redundant stack operations, and accelerating field access. Then we show that the performance improvement in the benchmark application which heavily use object creations or method invocations because our optimization techniques not only optimize application but also increase the opportunities for optimization within runtime library. Also, we compared our JIT compiler to Java and C++ to reveal the performance gap between our JIT and Java, C++. We observed that our JIT compiler has an approximate equivalence to the performance in most of benchmarks. On the other hand, our implementation still has huge performance gap between Java and C++ in memory allocation benchmark. As a future work, we want to apply our optimization approach to other part of runtime library to find another optimization opportunities in runtimes.

6.2 Trace-based JIT for CRuby

In chapter 4, we investigates new techniques to treat the performance for the interpreted implementation of a dynamic language.

We describe application of the JIT compilation to CRuby. It works by recording the execution path including the control flow of the interpreter and using results of type checking and directions of conditional branch, it translates bytecode to highly optimized machine code. Then we propose optimization techniques to improve the quality of trace (making longer trace and fewer transition). These techniques produce greater benefit for the idiom which Ruby programmer heavily used. In our experiments, we show that these two techniques improve the coverage of traces: native method inclusion and invariant block guard hoisting. And we showed a 1.5x through 5x performance improvement by using these two methods.

Currently, the drawback of our implementation of the JIT backend is the relatively high cost of the JIT compilation. This is due to increasing the number of the trace that is not frequently execute and high compilation cost for each trace. As future work, we need to explore the method of extracting frequently executing trace and the way to compile trace with restricted resource usage.

Publications

Refereed papers

- Masahiro Ide, Kimio Kuramitsu. Just in time compiler for KonohaScript using LLVM. IPSJ Transactions on Programming 6(1), 9-16, 2013
- Masahiro Ide, Kimio Kuramitsu. A Trace-based Just-In-Time compiler for Ruby. IPSJ Transactions on Programming Vol. 8, No. 1, pp. 1-10, 2015

Refereed papers (co-author)

- 志田 駿介，井出 真広，倉光 君郎．Mindstorms NXT を対象とした Konoha 処理系のコンパクト化．IPSJ Transactions on Programming 6(4), 1-9, 2013

Domestic conference (reviewed)

- 井出 真広, 志田 駿介, 倉光 君郎．LLVM を用いた静的型付きスクリプト言語 KonohaScript の Just-in-time コンパイラの設計と実装．先進的計算基盤システムシンポジウム SACSIS, May 2012.

International conference (co-author)

- Takuma Wakamori. Masahiro Ide, Midori Sugaya, Kimio Kuramitsu. Reconfigurable Scripting Language with Programming Risk. Workshop on Open System Dependability 2012, in conjunction with ISSRE2012, 2012.

Domestic conference

- 井出 真広, 平岡佑太郎, 養安元気, 中田晋平, 菅谷みどり, 倉光君郎, 大規模主記憶環境における GC 方式の性能評価, 情報処理学会第 83 回プログラミング研究発表会, 京都, March, 2011
- 井出 真広，倉光君郎，JavaScript 向け Ahead-Of-Time コンパイラの GCC による実装，日本ソフトウェア科学会第 28 回大会論文集, Sep, 2011
- 若森拓馬，井出 真広，中田晋平，倉光君郎，プログラム解析情報を利用可能なバイトコード操作器の設計と実装，第 84 回プログラミング研究発表会，June. 2011
- 井出 真広，倉光君郎，コード評価器の実行時最適化を行う Just-in-time コンパイラの設計と実装，並列／分散／協調処理に関する『北九州』サマー・ワークショップ SWoPP 鹿児島 2011, July. 2011
- 養安元気，菅谷みどり，井出 真広，倉光君郎, 移動のない Incremental GC におけるリアルタイム性評価, 第 176 回 SE・第 25 回 EMB 合同研究発表会, 5 月 2012 年, 東京.
- 内田 篤史, 養安 元気, 井出 真広, 菅谷みどり, 倉光君郎, スクリプト言語 Konoha によるカーネル診断スクリプトの開発, ポスター, 組込みシステムシンポジウム 2012, 東京, 10 月, 2012 年.
- 井出 真広, 志田 駿介, 倉光 君郎，JavaScript 生成言語への難読化処理の適用と性能評価．情報処理学会 第 94 回プログラミング研究発表会 PRO94
- 志田 駿介，井出 真広，菅谷 みどり，倉光 君郎．Mindstorms NXT を対象としたスクリプト処理系のコンパクト化．第 94 回プログラミング研究発表会 PRO94
- 石井 正樹，井出 真広，倉光 君郎，アシュアランス駆動プログラミングに向けて，情報処理学会プログラミング研究会 PRO94
- 石井 正樹，小野田 武朗，岡本 悠希，井出 真広，倉光 君郎，GSN からの実行可能なスクリプト生成の提案，情報処理学会プログラミング研究会 PRO96

References

[1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The jikes research virtual machine project: Building an open-source research community. *IBM Syst. J.*, 44(2):399–417, Jan. 2005.

[2] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The jikes research virtual machine project: Building an open-source research community. *IBM Syst. J.*, 44(2):399–417, Jan. 2005.

[3] H. Ardö, C. F. Bolz, and M. FijaBkowski. Loop-aware optimizations in pypy's tracing jit. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 63–72, New York, NY, USA, 2012. ACM.

[4] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. In *PROCEEDINGS OF THE IEEE, 93(2), 2005. SPECIAL ISSUE ON PROGRAM GENERATION, OPTIMIZATION, AND ADAPTATION*, volume 93, 2004.

[5] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. *SIGPLAN Not.*, 35(5):1–12, May 2000.

[6] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. Spur: a trace-based jit compiler for cil. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 708–725, New York, NY, USA, 2010. ACM.

[7] M. Bebenita, M. Chang, G. Wagner, A. Gal, C. Wimmer, and M. Franz. Trace-based compilation in execution environments without interpreters. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 59–68, New York, NY, USA, 2010. ACM.

[8] C. F. Bolz, A. Cuni, M. FijaBkowski, M. Leuschel, S. Pedroni, and A. Rigo. Allocation removal by partial evaluation in a tracing jit. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '11, pages 43–52, New York, NY, USA, 2011. ACM.

[9] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, New York, NY, USA, 2009. ACM.

[10] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[11] M. Chang, B. Mathiske, E. Smith, A. Chaudhuri, A. Gal, M. Bebenita, C. Wimmer, and M. Franz. The impact of optional type information on jit compilation of dynamically typed languages. In *Proceedings of the 7th symposium on Dynamic languages*, DLS '11, pages 13–24, New York, NY, USA, 2011. ACM.

[12] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 1–19, New York, NY, USA, 1999. ACM.

[13] M. corporation. Common language runtime. `https://msdn.microsoft.com/en-us/library/8bs2ecf4`.

[14] M. corporation. Dynamic Language Runtime. `http://dlr.codeplex.com/`.

[15] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[16] ECMA. *ECMAScript Language Specification (Standard ECMA-262).* Third edition, Dec. 2011.

[17] Free Software Foundation. GCC home page. `http://gcc.gnu.org/`.

[18] E. M. Gagnon and L. J. Hendren. Sablevm: A research framework for the efficient execution of java bytecode. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 3–3, Berkeley, CA, USA, 2001. USENIX Association.

[19] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM.

[20] A. Gal, C. W. Probst, and M. Franz. Hotpathvm: an effective jit compiler for resource-constrained devices. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 144–153, New York, NY, USA, 2006. ACM.

[21] A. Gaynor. Topaz. `https://github.com/topazproject/topaz`.

[22] Google. V8 JavaScript Engine. `http://code.google.com/p/v8/`.

[23] N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Javatm just-in-time compiler and virtual machine improvements for server and middleware applications. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3*, VM'04, pages 12–12, Berkeley, CA, USA, 2004. USENIX Association.

[24] D. Hiniker, K. Hazelwood, and M. D. Smith. Improving region selection in dynamic optimization systems. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 141–154, Washington, DC, USA, 2005. IEEE Computer Society.

[25] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. ACM.

[26] A. Homescu and A. Şuhan. Happyjit: A tracing jit compiler for php. In *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11, pages 25–36, New York, NY, USA, 2011. ACM.

[27] H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani. A trace-based java jit compiler retrofitted from a method-based compiler. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 246–256, Washington, DC, USA, 2011. IEEE Computer Society.

[28] IronRuby community. IronRuby. `http://ironruby.net/`.

[29] N. Ishii, S. Murata, Y. Chiba, and N. Doi. An implementation of a dynamic compiler for ruby. *IPSJ Transactions on Programming*, 4(1):109–122, mar 2011.

[30] ISO/IEC. *ISO/IEC 30170:2012 Information technology - Programming languages Ruby*. 2012.

[31] JRuby community. JRuby. `http://jruby.org/`.

[32] H. JVM. Java version history (j2se 1.3). http://en.wikipedia.org/wiki/Java_version_history.

[33] K. Kuramitsu. Konoha - implementing a static scripting language with dynamic behaviors. In *Workshop on Self-sustaining Systems 2010*, S3, The University of Tokyo, Japan, September 2010.

[34] S. M. Laboratories. Benchmarking java with richards and deltablue. http://labs.oracle.com/people/mario/java_benchmarking/index.html.

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

[36] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, Apr. 1960.

[37] S. Murata, N. Ishii, Y. Chiba, and N. Doi. Optimizing exception handling in a dynamic compiler for ruby. *IPSJ Transactions on Programming*, 4(1):123–133, mar 2011.

[38] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura. Openjit: An open-ended, reflective jit compiler framework for java. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, ECOOP '00, pages 362–387, London, UK, UK, 2000. Springer-Verlag.

[39] M. Paleczny, C. Vick, and C. Click. The java hotspot^{TM} server compiler. In *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[40] M. Pall. The luajit project. http://luajit.org/.

[41] E. Phoenix. Rubinius. http://rubini.us/.

[42] E. H. Ryan Davis. ruby2c. https://rubygems.org/gems/ruby2c/.

[43] L. Sansonetti. Macruby. http://macruby.org/.

[44] K. Sasada, Y. Matsumoto, A. Maeda, and M. Namiki. Yarv : Yet another rubyvm : The implementation and evaluation. *IPSJ Transactions on Programming*, 47(2):57–73, 2006-02-15.

[45] S. Shiba, K. Sasada, and K. Hiraki. Cast off : A compiler for ruby implemented as a library. *IPSJ Transactions on Programming*, 2012(1):1–22, oct 2012.

[46] K. Shudo. shuJIT - Java Just-in-Time Compiler for x86 Processors. http://www.shudo.net/jit/index.html.

[47] Syoyo Fujita. Ambient Occlusion Benchmark. http://code.google.com/p/aobench/.

[48] M. Tatsubori, A. Tozawa, T. Suzumura, S. Trent, and T. Onodera. Evaluation of a just-in-time compiler retrofitted for php. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '10, pages 121–132, New York, NY, USA, 2010. ACM.

[49] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM.

[50] C. Wimmer and T. Würthinger. Truffle: A self-optimizing runtime system. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 13–14, New York, NY, USA, 2012. ACM.

[51] xue.yong.zhi. xruby. `https://code.google.com/p/xruby/`.

[52] Yukihiro Matsumoto. Ruby Programming Language. `https://www.ruby-lang.org/`.