LLVMを用いたスクリプト言語KonohaScriptのJust-in-timeコンパイラの設計と実装

井出真広[†] 志田駿介[††] 倉光君郎[†,†††]

近年，スクリプト言語の言語処理性能の向上させるための手法として，スクリプト言語の Just-in-time(JIT) コンパイル技法が注目を集めている．JIT コンパイラによって言語処理性能の向上を目指す際に，難しい点は，動的型付け（型推論）と独自の言語ランタイムにある．本研究の目的では，静的型付けスクリプト言語を使うことで，型推論のオーバーヘッド以外の部分の JIT コンパイルの性能を評価することである．本研究では，静的型付けスクリプト言語 KonohaScript を用いて 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，解決するために実装した最適化手法について述べる．

The implementation of Just in time compiler for KonohaScript using LLVM.

Masahiro Ide ,[†] Shunsuke Shida [††] and Kimio Kuramitsu [†,†††]

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

1. はじめに

近年，スクリプト言語の言語処理性能の向上させるための手法として，スクリプト言語の Just-in-time(JIT) コンパイル技法が注目を集めている．[1),2)] JIT コンパイラは，プログラム実行中にバイトコードを機械語に変換し，CPU によって直接実行することで処理速度を飛躍的に向上することが可能となっている．

しかし，スクリプト言語の JIT コンパイラが生成した機械語 (以下 JIT 生成コード) は，C++言語や Java 言語のコンパイラが生成した機械語と比較すると性能が劣る．これは，スクリプト言語の JIT コンパイラが，動的型付けを考慮したコードを生成する必要があり，また独自の言語ランタイムとの連携を考慮したコード生成を行う必要があるためである．

† 横浜国立大学大学院
Graduate School of Yokohama National University
†† 横浜国立大学
Yokohama National University
††† 日本科学技術振興機構 CREST
Japan Science and Technology Agency/CREST

スクリプト言語の JIT コンパイラと動的型付けの性能の関連性については，Chang ら[3)] はスクリプト言語への型情報を与える事による JIT コンパイラが生成する機械語の実行性能への影響を (1) 完全に型付けされたコード，(2) 一部が型付けされたコード，(3) 型付けされていないコードの 3 種類に対して比較を行い，動的型付けが JIT コンパイラに与える影響の評価を行なっている．スクリプトに対して型情報を追加することで平均して 60% 程度の速度向上があったと報告している．

本論文ではスクリプト言語の定義を実行中のソースコード解釈と言語ランタイムが不可分な言語処理システムを指すものとしている。本研究の目的は，静的型付けスクリプトを使うことで，JIT コンパイルによって生成されたコードのうち，ソースコード解釈を行う上で生じる動的型付け処理のオーバヘッドの部分以外，つまり言語ランタイムによるオーバヘッドを評価することである．

本論文では，静的型付けスクリプト言語である KonohaScript を用いてバックエンドとして Low Level Virtual Machine (LLVM)[4)] を用いた LLVM JIT コ

© 2012 Information Processing Society of Japan

ンパイラの設計と実装を行った．その上で，ランタイムライブラリにおいて JIT 生成コードのパフォーマンスに影響を与える要素について調査を行った．また，我々は構築した JIT コンパイラについてベンチマークアプリケーションを用いて実行性能について評価を行った．

本論文の構成は，次のとおりである．第 2 節では，一般にスクリプト言語に搭載されている言語ランタイムについて説明し，本論文で用いる KonohaScript 処理系の言語ランタイムについて説明する．第 3 節では本論文では提案する JIT コンパイラについて設計を述べ，また第 4 節では提案する JIT コンパイラの実装について述べる．第 5 節では性能について評価を行い，第 6 節では関連研究について述べる．第 7 節にてまとめと今後の課題を述べる．

2. スクリプト言語の言語ランタイム

本節では，まず現在のスクリプト言語処理系が提供する言語ランタイムについて特徴を述べる．次に本論文で対象とする KonohaScript 言語処理系のランタイムについて説明する．

2.1 スクリプト言語処理系

スクリプト言語の最も基本となる構成は，以下の 4 項目である．本論文では，下記項目のうちコンパイラとコード評価器を実行エンジンと呼び，拡張ライブラリ, メモリ管理機構をまとめてランタイムライブラリと呼ぶこととする．

- コード評価器
- コンパイラ
- 拡張ライブラリ
- メモリ管理機構

コード評価器・コンパイラ

スクリプト言語処理系はコード評価器にインタプリタを用いる実行処理系を持っており，処理速度を向上させるために独自のバイトコードインタプリタや JIT コンパイラを備えることが多い．スクリプト言語はこの実行処理系とランタイムライブラリを連携させながらプログラムの実行を行う形をとっている．

拡張ライブラリ

スクリプト言語では，C 言語などで開発されたライブラリ (以下，拡張ライブラリ) を用いて機能拡張することが容易に可能である．そのため 1 つの言語処理系にはバイトコードで表現されたメソッド，拡張ライブラリなど C 言語で記述されたメソッドなど異なる種類のメソッドが混在している．

各ランタイムライブラリでは，それぞれのメソッドが相互に呼び出しを行うために言語処理系ごとにインタフェースが定められている．また，ランタイムライブラリと実行エンジンとの間でデータの受け渡しを行うためにスクリプト言語ではスタック構造が利用されている．スタック構造の実装には C のスタックを用いる方法，もしくは独自のスタック構造を用いる方法の 2 通りがある．

メモリ管理機構

言語ランタイムはスクリプトを実行する際に利用するメモリを管理するため，ガーベジコレクション (以下 GC) を利用する．

言語ランタイムがデータ構造を表現する際には GC によって管理されているメモリ領域を独自のメモリレイアウトにもとづいて利用している．各言語ランタイムのメモリレイアウトはヘッダ領域と本体領域の 2 つから構成されている．

ヘッダ領域ではメモリ管理やオブジェクトの型情報，各オブジェクトの特性を示すフラグを保持するなど言語処理系が内部的に使用するデータを保持している．例えばオブジェクトの型チェックを行う際にはヘッダ領域に保持されている型情報にアクセスし，型検査を行うことが可能となっている．

なお，本体領域の利用方法はオブジェクトの型，種類によって異なっている．

2.2 KonohaScript 言語処理系

本節では本論文で用いるスクリプト言語処理系である KonohaScript 言語処理系について述べる．KonohaScript[5] は C 言語でスクラッチから開発された静的な型付けオブジェクト指向スクリプト言語である．KonohaScript の言語設計については文献[6] に詳しいので詳細はそちらを参照されたい．

KonohaScript 言語処理系を構成する要素について **図 1** に示す．

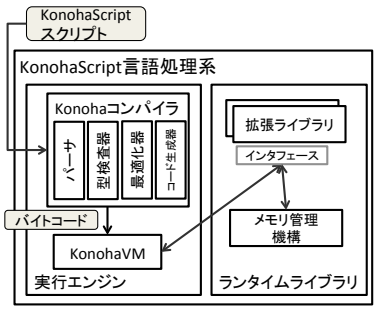


図 1 KonohaScript 言語処理系のアーキテクチャ

コード評価器・コンパイラ

ⓒ 2012 Information Processing Society of Japan

KonohaScript ではコンパイル時型検査や型情報に基づき，バイトコードとバイトコード評価器 (以下 KonohaVM) を実装している．KonohaVM は静的に型付けされたバイトコードを実行するために設計されたレジスタ型のバイトコードインタプリタである．KonohaVM は以下の順にソースコードを実行する。KonohaScript のソースコードはコンパイラによってメソッドを 1 つのコンパイル単位としてバイトコードにコンパイルされる．KonohaVM はスクリプトのトップレベルのコードから順番にバイトコードの評価を行い，最後に main メソッドを実行する．

拡張ライブラリ

KonohaScript では拡張ライブラリを提供するためのインタフェースとしてすべてのメソッドは以下の `knh_Fmethod` 型に合わせた形で定義される．

```
void (*knh_Fmethod)(CTX ctx, ksfp_t *sfp,
          long _rix);
```

ここで，`ctx` は言語ランタイムの管理情報 (Context) であり，`sfp` は KonohaScript の言語ランタイムが持つスタック構造 (Konoha スタック) の先頭を示すポインタ，`_rix` は返り値が置かれる Konoha スタック上のインデックスを示している．これに合致する C の関数であれば，API 定義スクリプトに書かれたメソッドから呼び出すことができる．メソッドの引数や返り値などメソッド間のデータの受け渡しには Konoha スタックが用いられる．メソッドを呼び出す際，呼び出し元はスタックポインタ上に引数を push し，対象のメソッドを呼び出す．

KonohaScript は静的な型情報を生かした自動的な boxing/unboxing を行なっており，プリミティブ型の値については必要ない限り unbox 化された状態 (オブジェクトではなく数値のまま) で扱われる．Konoha スタックでは数値とオブジェクトの参照を明確に区別可能なワイドスタックを採用している．

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

メモリ管理機構

KonohaScript は GC を用いた自動的なメモリ管理を提供している．

なお，64bit アーキテクチャでは KonohaScript の各オブジェクトはすべて 64byte でアライメントされており，**図 2** で示されるメモリレイアウトを持つ．

オブジェクトのヘッダ領域にはクラス情報 (cfino メンバ) とオブジェクトの状態を表すフラグと共に各 GC アルゴリズムが利用することができる gcinfo メンバが用意されている．

```
typedef struct {
  struct hObject {
    ClassInfo *cinfo; //クラス情報
    void *gcinfo; // 管理データ GC
    long flags;void *metadata;
  } h; // オブジェクトのヘッダ情報
  struct kObject **fields; //Object body
} kObject;
```

図 2 Object 型のメモリレイアウト

3. Konoha-LLVMJIT コンパイラの設計

本節では KonohaIR を用いて LLVM IR を生成する方法と，LLVM IR に変換する際に工夫した点について述べる．

3.1 Konoha-LLVMJIT コンパイラ

高速な実行を実現するために，我々は新たに中間表現 (以下 KonohaIR) を設計し，ソースコードから KonohaIR を生成し，さらに KonohaIR から LLVM Intermediate Representation (LLVM IR)[7] に変換する方針を取った．そして例えば共通部分式の除去といったコンパイラ最適化を LLVM のバックエンドを用いて実行し，最後に機械語を生成することとした．

KonohaIR は LLVM のコンパイラ最適化を用いて効率的なコード生成を行うことを目的としており，KonohaVM が用いるバイトコードと LLVM IR の両方の中間表現を元に命令の設計を行なっている．

KonohaIR には式とアノテーションの 2 つの要素から構成されている．まず，式は SSA 形式を用いて表現を行なっており，それぞれの式は複数の値をオペランドとして 1 つの値を出力する操作を表現している．式とそのオペランドにはすべて型情報が付与されており，また，アノテーションによって定数値などの情報を与えている．アノテーションによって与えられた情報は LLVM IR に変換する際，また LLVM バックエンドが最適化を行う際に利用している．

3.2 LLVM IR の生成

LLVMJIT では KonohaScript のメソッドは一旦 KonohaIR に変換された後，LLVM IR に変換される．KonohaIR は LLVM IR をベースに設計が行われているため，KonohaIR の命令セットと LLVM IR の命令セットについては 1 対 1 の対応がとれている．KonohaIR から LLVM IR への変換はそれぞれ対応する命令を生成するだけの単純なものとなっている．例えば，KonohaIR の整数値の加算命令である add 命

ⓒ 2012 Information Processing Society of Japan

令は LLVM IR の add 命令に対応している．

ここで，LLVM IR では，多くのプログラム言語，アーキテクチャに対応するために LLVM IR 上で表現できる変数の型，関数呼び出しの仕組みが定義されている．一方で KonohaIR は KonohaScript のデータ構造，ランタイム上に設計されている．そのため型システムと関数の扱いについては KonohaIR と LLVM IR の間では単純な 1 対 1 の対応では表現することができなかったため，別途対応を行なっている．以降の節では型システムとに関数呼び出しについて LLVM IR と KonohaIR の比較を行い，それぞれの対応と変換方法について述べる．

3.2.1 型の定義

LLVM IR では多くのデータ型のサポートを行なっており，bool 型 (i1 型), char(i8 型), long(i32 型) など N ビット長の整数型や複数の浮動小数点型を提供している．また，構造体や配列のサポートも行なっている．

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

KonohaScript はプリミティブ型として Boolean, Int, Float の 3 種類を用意している．LLVM IR 上ではそれぞれ boolean 型を i1 型，int 型を i64 型，Float 型を double 型に対応させている．

KonohaIR を LLVM IR に変換する際に，LLVMJIT では以下の変換コードを挿入し対応を行なっている．

- プリミティブ型の値の boxing/unboxing
- 動的型付けコードに対する型チェックを行う関数の挿入

3.2.2 関数呼び出し

KonohaScript ではすべてのメソッドは `knh_Fmethod` 型に合わせた形でメソッドが定義される．そのため，LLVMJIT で定義されるメソッドは LLVM IR 上では `knh_Fmethod` のインタフェースを用いて定義することとした．

そのため，LLVM IR に変換する際に各メソッドの先頭にメソッドの引数を Konoha スタックからロードする処理，メソッドの末尾には返り値を Konoha スタックに書き戻す処理を挿入している．

3.3 コンパイル例

本節では LLVMJIT の処理の流れについて，KonohaScript のコードを用いて説明を行う．**図 3** に以下の説明で用いる KonohaScript のソースコードを示す．

図 3 のソースコードは LLVMJIT によって一旦 KonohaIR に変換される．(**図 4**) まず，KonohaScript の各ブロックは KonohaIR では 1 つの BasicBlock に変換される．また，KonohaIR は SSA 形式であるため，変数の宣言と利用を一意になるように PHI 命令が挿入される．PHI 命令は BasicBlock と変数のペアのリストで表現される．

次に，LLVMJIT は KonohaIR のコードから LLVM IR に変換を行う．(**図 5**) 前節で述べた例外を除き，KonohaIR から LLVM IR への変換は 1 対 1 の変換される．

最後に LLVMJIT は LLVM IR を LLVM バックエンドを用いて機械語を生成し，処理を終了する．

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

図 3 KonohaScript サンプルコード

```
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
}
```

図 4 図 3 を KonohaIR に変換したコード

4. 実　　装

我々は JIT コンパイラが言語ランタイムと連携するために生成するコードのうち，関連が高いと考えられる以下の項目について JIT コンパイラがコード生成

ⓒ 2012 Information Processing Society of Japan

```
define void @f(%ctx* %ctx, %sfp* %sfp,
               i64 _rix) {
bb0:
  %0   = getelementptr %sfp* %1, i32 1, i32 1
  %arg = load i64* %0
  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
  ret void
}
```

図 5 図 4 を LLVM IR に変換したコード

を行う際に最適化を施すこととした．

- メソッドのインタフェース
- ランタイムライブラリ呼出
- オブジェクトのメモリレイアウトの変更

本節では LLVMJIT コンパイラ に適用した最適化について述べる．

4.1 メソッド呼出の直接呼出

ランタイムライブラリが管理するメソッドには，拡張ライブラリが提供する C 言語などで記述されたメソッドやバイトコードで表現されたメソッドなど多種類のメソッドが混在する．これら種類の異なるメソッドが互いのメソッドを呼び出し可能とするために共通の呼出しインタフェースに合わせたコードの生成を行う．本コンパイラが対象とする KonohaScript では第 2 節で述べたとおり KonohaScript のメソッドは共通のインタフェース `knh_Fmethod` で定義される．そのため LLVM IR で記述されたメソッド同士でメソッドを呼び出す際にも Konoha スタックに引数を積む必要がある．

JIT 生成コード同士の呼び出しの場合，呼出インタフェースを介する必要はなく，またスタックに引数を積み直す必要がない．

LLVMJIT コンパイラでは従来の呼出インタフェースに加え，JIT 生成コード同士の呼び出しでスタックに引数を積む操作や返り値をスタックに積むなどインタフェース処理部分を省き，メソッドを直接呼び出すインタフェース (以下，特化インタフェース) を用意する．

動的束縛を行う場合には LLVMJIT コンパイラは callee/caller がともに JIT 生成コードであることを保証できないため，LLVMJIT コンパイラでは callee が static または final メソッドである場合にこの工夫によって他のメソッドと同じインタフェースを保ちつつ高速にメソッドの呼出を行うことが可能となる．

特化インタフェースの例として factorial の KonohaScript コード (**図 6**) と factorial メソッド，そして特化インタフェースの例を**図 7** に示す．なお JIT コンパイラによって生成された factorial メソッドとその特化インタフェースは読みやすさのために JIT 生成コードを C 言語に書きなおしたコードを示す．

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

図 6 KonohaScript で記述された factorial 関数

```
/* factorial method (original) */
void fact(CTX ctx, ksfp_t *sfp, long _rix) {
    int n = sfp[1].ivalue;
    int ret = fact_opt(ctx, n);
    sfp[_rix].ivalue = ret;}
/* factorial method(opt version) */
int fact_opt(CTX ctx, int x) {
    if (n < 2) return 1;
    return n * fact_opt(ctx, n-1);}
```

図 7 図 6 から LLVMJIT が生成したコード

LLVMJIT コンパイラで生成されたメソッドを呼び出す際には Konoha スタックの push/pop 操作を省くことができるため，このような工夫をすることで他のメソッドと同じインタフェースを保ちつつ高速に実行することが可能となる．

4.2 ランタイムライブラリのインライン展開

インライン展開はメソッド呼出コストの削減とメソッド間にわたってコンパイラの最適化範囲拡大につながる広く知られている最適化手法である．インライン化によるコンパイル時の最適化が適応される範囲をランタイムライブラリにまで拡大し，生成されるコードの質の向上につながることが期待できる．

しかし，既存の JIT コンパイラでは C で実装されたコンパイル済みのランタイムライブラリをインライ

ⓒ 2012 Information Processing Society of Japan

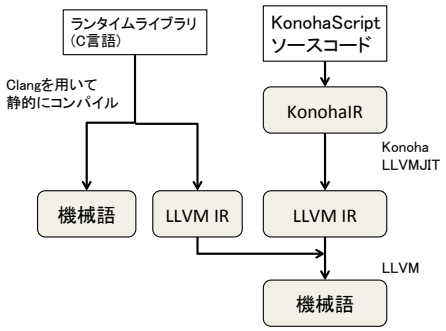


図 8 言語ランタイムのインライン展開

ン化することは容易ではない．これはインライン化するにあたり，ランタイムライブラリがどのような処理を行うのかを知ることは困難なためである．

我々はランタイムライブラリをインライン化するにあたり，ランタイムライブラリを事前に LLVM IR に変換しておき，LLVM IR 上でランタイムライブラリのインライン展開を行う方式を採用した．

図 8 に C 言語で実装されたランタイムライブラリを LLVM IR に変換し，インライン化するまでの変換例を示す．我々は，プログラム実行前にランタイムライブラリを LLVM の C フロントエンドである `clang` を用いてコンパイルし，ランタイムライブラリを生成すると同時に LLVM IR の生成を行う．LLVMJIT コンパイラは KonohaIR から LLVM IR の生成を行う際に，ランタイムライブラリから生成した LLVM IR を用いることでメソッドに対してインライン化を行うことが可能となる．

4.3 メモリレイアウトを考慮したフィールドアクセス

オブジェクトのフィールドアクセスは，オブジェクト指向プログラム言語においては多く利用される機能の 1 つである．そのためフィールドアクセスの高速化は実行性能に対して影響が少なくないことが予想できる．

オブジェクトのフィールドアクセス方法は言語処理系がそれぞれ定めているオブジェクトのメモリレイアウトによって決められる．我々はメモリレイアウトが実行性能に対してどの程度影響するか検討を行った．

素朴に JIT コンパイラが独自のメモリレイアウトを用いてオブジェクトの表現を行った場合，拡張ライブラリや GC などに対して多くの修正が必要となる．

ここで，KonohaScript では kObject 型 (図 2) は fields メンバに Object 型を格納するための配列を記録している．fields メンバはヒープ領域に malloc() によって獲得されており，GC によってそのオブジェクトが回収されるときに free() によって解放される．

```
typedef struct {
  hObject h;
  kObject **fields;
  kObject *smallobject[3];
} kSmallObject;
// ただし *fields = smallobject
```

図 9 フィールドの埋込み

なお，KonohaScript ではフィールドの数が少ないオブジェクトは Object の領域中 (**図 9** の smallobject メンバ) にフィールドの実体を記録するようにしてメモリ使用量の最適化が施されている．

またフィールドアクセスについては，インタプリタや拡張ライブラリ内で共通した操作を行うために fields メンバに Object の領域のアドレスを記録することで操作の共通化を行っている．

LLVMJIT ではメモリレイアウトの変更が性能にどの程度影響するかを検討するため，フィールドの数が少ないオブジェクトのうち，final なクラスについてフィールドアクセスの方法を Object に埋め込まれたデータにアクセスする方式に変更する最適化器を構築した．

5. 評　　価

本節では第 3 節で述べた LLVMJIT コンパイラおよび第 4 節で述べた最適化の効果を評価するために評価対象プログラムとして**表 1** に挙げるベンチマークプログラムについて KonohaScript へ移植したプログラムを用いて実験を行った．

以下に本実験で用いた環境と, 利用した C コンパイラ,LLVM ライブラリは以下のとおりである．

- CPU Intel(R) Core(TM) i7 2.2GHz
- メモリ 8GB 1333MHz
- OS MacOSX 10.7.2
- C コンパイラ GCC 4.4.5
- LLVM version 3.0

また，LLVM は部分冗長式除去や定数伝搬など様々な最適化パスを用いて LLVM IR の最適化を行うことが可能である．最適化パスの数, 適応の順番によって得られる性能は，大きく異なる．本実験では LLVM が最適化パスとして用意している，関数の最適化パスのセット (StandardFunctionPass)，プログラム全体の最適化パスのセット (StandardModulePass)，をそれぞれ適応し，実験を行った．

ⓒ 2012 Information Processing Society of Japan

ベンチマーク名	説明
AObench[8]	レイトレーシング (オブジェクト生成, フィールドアクセス)
DeltaBlue[9]	Constraint solver benchmark(フィールドアクセス)
NBody	惑星の N 体問題のシミュレーション (フィールドアクセス)
Binarytrees	深さ n の 2 分木を大量に生成 (オブジェクト生成)
Mandelbrot	マンデルブロ集合を計算する (浮動小数点演算)
Richards[9]	OS カーネル シュミレーション.(フィールドアクセス, メソッド呼出)
Spectralnorm	正方行列の 2 ノルムを計算する.(浮動小数点演算, 配列アクセス)

表 1 ベンチマークプログラム (括弧内は主な処理)

5.1 実行時間の比較

図 10 は各ベンチマークプログラムを第 4 節で述べた最適化を施した LLVMJIT を用いてコード生成を行い，実行した場合の実行時間の比較である．なお，縦軸はそれぞれ，LLVMJIT コンパイラを利用しコード生成を行った場合 (JIT)，特化インタフェースの最適化を施した場合 (JIT+API)，特化インタフェースとフィールドアクセスの最適化を施した場合 (JIT+API+FLD)，特化インタフェース，フィールドアクセス，ランタイムライブラリのインライン化の最適化を施した場合 (JIT+API+FLD+ILN) を表している．

それぞれの値は LLVMJIT コンパイラのみを利用した場合の時間を 1 として正規化されており，短いほうが高速であることを示している．また，LLVMJIT コンパイラで実行した場合の時間についてはコンパイル時間を含めたプログラム全体の実行時間とした．

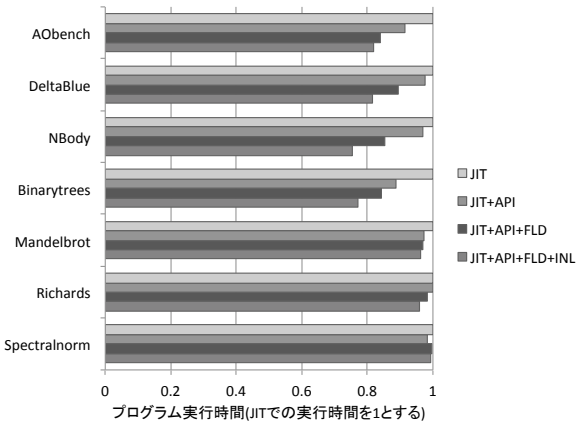


図 10 ベンチマーク結果

図 10 において Mandelbrot, Richards, Spectralnorm のプログラムは，最適化の効果を一部，もしくはほとんど得ることができなかった．生成された KonohaScript のプログラムと生成された LLVM IR を観察すると，原因は以下のとおりであると考えられる．

Mandelbrot と spectralnorm は配列操作と浮動小数点演算で構成されており，ランタイムライブラリを用いたコードはほとんど利用されていない．そのため LLVMJIT が生成したコードは最適化を施した場合とほぼ同じ結果となったと考えられる．

また，Mandelbrot, Richards はフィールドアクセスの最適化の対象外である．大きいオブジェクトに対してフィールドアクセスを行うプログラムとなっており，ループ内でアクセスされることによるオーバヘッドが残っている．

さらに Richards は多くのメソッドが動的束縛されるため，JIT コードのメソッドの直接呼出を行うことができない．

一方で，AObench, DeltaBlue, NBody, Binarytrees の 4 つのベンチマークプログラムについてはすべての最適化を施した場合，平均して 20% の実行性能の向上を確認することができた．

これらの性能向上は，LLVMJIT がランタイムライブラリを利用する際に生じるオーバヘッドを削減できていることを示している．

特に，フィールドアクセスの最適化について，小さなオブジェクトのみを対象しているため今回の実装では，大きいオブジェクトを扱うプログラムは高速化はされない．しかし，AOBench や NBody の計測結果からオブジェクトのメモリレイアウトを改善することで高速化が可能である．

6. 関 連 研 究

本節では，スクリプト言語の JIT コンパイラについて概観し，本研究との関連性を述べる．

近年，スクリプト言語の JIT コンパイラのコード生成器として LLVM を利用した研究/開発が行われている．LLVM を Ruby の JIT コンパイラとして利用しているプロジェクトとして Ruby で実装された Ruby 実行環境である Rubinius[10] がある．Rubinius ではコアライブラリ以外の機能を Ruby で記述し，プログラム実行中に LLVM を用いて機械語に変換している．また，LLVM を Python から利用するプロジェクトとして UnladenSwallow[11] がある．本研究では既存のプログラムや C 言語で実装されたランタイムライブラリを変更せずに利用できる点で両者共に異なっている．

ⓒ 2012 Information Processing Society of Japan

スクリプト言語における JIT コンパイラは JavaScript における TraceMonkey や, Python における Pypy などで研究，開発されている．これらの JIT コンパイラは実行時トレース情報を用い，ループなどプログラム中で頻繁に実行される箇所をコンパイルし，機械語の生成を行っている．これらのコンパイラではインタプリタを用いて実行時トレースを収集するため，ランタイムライブラリなど機械語で記述されたコードとの連携を行うことが難しい．本研究では KonohaScript のスクリプトを LLVM IR に変換し，また，ランタイムライブラリも LLVM IR に変換することで，ランタイムライブラリを含めた最適化を行う機会を提供している点が異なる．

7. ま と め

本稿では，静的スクリプト言語 KonohaScript に対して LLVMJIT コンパイラの設計を行い，言語ランタイムによるオーバヘッドについて，ランタイムライブラリを対象とした最適化について述べた．

我々は構築した JIT コンパイラについてベンチマークアプリケーションを用いて実行性能について評価を行った．我々はランタイムライブラリのうち，オブジェクトレイアウトとメソッドの呼び出し機構について JIT コンパイラが生成する機械語の実行性能に影響を与えると結論づける．なお，メモリ管理機構はアルゴリズムの性質上，実行されるスクリプトに関連しており，JIT コンパイラの関連は少ないと我々は考えている．

謝　　辞

本研究は，JST/CREST「実用化を目指した組込みシステム用ディペンタブル・オペレーティングシステム」領域の研究課題「実行時の安全性を確保する SecurityWeaver と P-SCRIPT」の一部として行われた．

参 考 文 献

1) Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin 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*, pp. 18–25, New York, NY, USA, 2009. ACM.
2) Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael 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, pp. 465–478, New York, NY, USA, 2009. ACM.
3) Mason Chang, Bernd Mathiske, Edwin Smith, Avik Chaudhuri, Andreas Gal, Michael Bebenita, Christian Wimmer, and Michael 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, pp. 13–24, New York, NY, USA, 2011. ACM.
4) Chris Lattner and Vikram 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.
5) Kimio 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.
6) Kimio Kuramitsu. Konohascript 静的スクリプト言語の設計と進化. 日本ソフトウェア科学会第 28 回大会, September 2011.
7) Chris Lattner. Llvm language reference manual. `http://llvm.org/docs/LangRef.html`.
8) Syoyo Fujita. Ambient Occlusion Benchmark. `http://code.google.com/p/aobench/`.
9) Sun Microsystems Laboratories. Benchmarking java with richards and deltablue. `http://labs.oracle.com/people/mario/java_benchmarking/index.html`.
10) Evan Phoenix. `http://rubini.us/`.
11) A faster implementation of python. `http://code.google.com/p/unladen-swallow/`, 2010.

© 2012 Information Processing Society of Japan