修 士 論 文 の 和 文 要 旨

研究科・専攻	大学院　情報理工学学研究科　情報・通信工学専攻　博士前期課程		
氏　名	漆原　明博	学籍番号	１２３１０１５
論文題目	サーバサイド向け JavaScript 処理系における Just In Time コンパイラの実装		

要　旨

JavaScript は、一般的に Web アプリケーションのクライアント側の開発で用いられるが、サーバ側でも JavaScript を用いる事かできれば、学習コストなどのアプリケーション開発コストの削減が期待できる。Web アプリケーションにおいて、処理にかかる時間の多くは I/O にかかる時間であるが、依然として統計処理などといった時間のかかる計算を行う場合もあり、高速に実行する需要がある。しかし、JavaScript は変数の型に多相性が存在し、実行時に型が定るまで処理が決定しないため、高速化のための静的な最適化は不向きである。そのため、頻繁に実行されるコードを実行時情報を用いてマシンコードにコンパイルし、繰り返し実行することで高速化を図る動的な最適化手法である Just In Time (JIT) コンパイル機構が有効である。本論文は、サーバサイド向けの JavaScript 仮想機械処理系において、C 言語等のバックエンドとして用いられている、コンパイル基盤である LLVM を用いた JIT コンパイル機構を実装し評価した。長期間のプログラムの運用を考慮した際、コンパイルにかかる時間は非常に短い時間の割合と考えられるため、本 JIT コンパイル機構では高速なマシンコード生成に注力した。実行時型情報、及び型推論機構から得た情報を元に、コンパイルにおいて様々な高速化を積極的に適応した。プログラムを特定の型環境下に特殊化し、更に、オブジェクトのプロパティの取得の最適化など、幾つかの JavaScript 依存の最適化や、更に LLVM の提供する様々な最適化を適応した。その結果、数値計算が中心のプログラムについて、長期間のプログラムの運用を前提とした場合、高速なマシンコードを生成が実行時間の短縮に繋がり、本機構の有効性が確認できた。

平成 25 年度修士論文

サーバサイド向け JavaScript 処理系における Just In Time コンパイラの実装

電気通信大学大学院情報理工学研究科
情報・通信工学専攻
コンピュータサイエンスコース

学籍番号　　: 1231015
氏名　　: 漆原 明博
主任指導教員 : 岩崎 英哉 教授
指導教員　　: 沼尾 雅之 教授
提出日　　: 2014 年 1 月 27 日

要旨

JavaScript は、一般的にアプリケーションのクライアント側の開発で用いられるが、サーバ側でも JavaScript を用いる事かできれば、アプリケーション開発コストの削減が期待できる。JavaScript はその特徴から、静的な最適化は不向きであり、頻繁に実行されるコードを実行時情報を用いてコンパイルする Just In Time (JIT) コンパイル機構が有効である。本論文は、サーバサイド向けの JavaScript 仮想機械処理系において、サーバサイドという特性を生かし、C 言語等のバックエンドとして用いられている、コンパイル基盤である LLVM を用いた JIT コンパイル機構を実装し評価した。JavaScript 依存の最適化を実装し、加えて LLVM の提供する様々な最適化を適応した。その結果、数値計算が中心のプログラムについて、長期間のプログラムの運用を前提とした場合、本機構の有効性が確認できた。

目次

1 はじめに

1.1 背景

近年、Web アプリケーション開発が盛んに行われている。Web アプリケーションにおけるクライアントサイドの開発言語としては、主に JavaScript が用いられる。JavaScript は ECMAScript [21] 準拠のプロトタイプベースのオブジェクト指向言語で、柔軟な記述が可能であり可読性も高いという利点がある。一方で、サーバサイドのプログラムを作成する際には、依然として他の言語を用いるのが一般的であるが、サーバサイド向けの用途に応じた実用的な JavaScript 処理系があれば、それを用いることでアプリケーションの開発コストの削減が期待できる。

また、Web アプリケーションのサーバサイド開発に、一般的に静的型付けの言語より実行速度の遅い、動的型付け言語が用いられるようになってきている。それは、レスポンスにかかる時間が、データの処理にかかる時間よりも、I/O 要求に支配されるためである。しかし、統計的処理など、依然として多くのデータを用いた時間を要する処理を行う場合もあり、動的型付け言語を用いる場合でも、高速な処理に対する要求がある。

JavaScript は、プログラムの処理速度を向上させる上で、言語の特性上静的な最適化に不向きであることが一般的に知られている。JavaScript は実行するまで変数の型が定まらず、また型によってプログラムの振る舞いが変化するためである。そこで、実行時情報を用いた最適化が効果的であると考えられる。プログラム実行中の高速化手法として、プログラム実行時の動的な情報を用いて、頻繁に実行される一部のプログラムコードをマシンコードのような高速化されたコードに変換し、それを複数回実行することにより実行時間の短縮を図る機構である Just In Time (以下、JIT) コンパイル機構が用いられる。

既存の JavaScript 処理系は、クライアントサイドの実行を視野に入れて構成されている。クライアントサイドでは、JIT コンパイル機構により生成されたマシンコードの実行回数はあまり多くないと想定されるため、マシンコードへの変換に時間をかけないように、JIT コンパイル機構に実装する最適化手法の取捨選択が行われている。一方でサーバサイドで用いる場合、Web アプリケーションのような、データセットを入力として受け取り、その情報を処理をして出力するプログラムの運用が想定される。長期間に渡り同じコードが実行され、また動作させるプログラムが運用前に決定しているので、運用前に JIT コンパイル機構により一部コードを予め変換しておくことなども可能である。これらの特性の違いから、サーバサイド向けに適した JIT コンパイル機構の構成は、クライアントサイドとは異なると考えられる。

コンパイラ基盤である LLVM [2] は高度な JIT コンパイル機構の作成支援機構を提供している。LLVM は中間言語である LLVM-IR を対象に、非常に様々な高速化機構を提供しており、高速なマシンコードの生成が可能である。統計処理等を想定した際、実行時間における特定の関数やループの割合は非常に大きく、そのコードを高速に実行することが、全体の実行速度の向上になる。LLVM の最適化及びマシンコード生成には若干時間を要するが、それらの時間は無視できる程の割合と考えられる。

1.2 目的と方針

本研究では、我々が開発しているサーバサイド向け JavaScript 仮想機械処理系 SSJSVM [11] にサーバサイド用途に応じた JIT コンパイル機構を設計及び実装し、その効果の検証を行う。先行研究 TESSA [1] を参考にして、幅広い言語に対応可能なコンパイラ基盤である LLVM [2] に含まれる JIT コンパイル機構の作成支援機構を用いて、SSJSVM における JIT コンパイル機構を実装する。また、実装に伴い、JIT コンパイル機構に幾つかの最適化手法を導入し、その効果の検証を行う。

1.3 構成

本論文の構成を述べる。2 章では本研究における予備知識について述べる。3 章では JIT コンパイル機構の設計ついて述べる。4 章では設計と実装について述べる。5 章では、その処理系の JIT コンパイラ機構についての実験結果について議論を行い、6 章では、関連研究について紹介する。最後に 7 章で本論文をまとめる。

2 予備知識

2.1 JavaScript

2.1.1 概要

JavaScript はプロトタイプベースのオブジェクト指向言語であり、オブジェクトに対してインスタンス変数やメソッドの追加をクラスに依存せずに行うことができる。このため、クラスベースのオブジェクト指向言語に比べて、プログラムの設計を柔軟に行うことができる。この特徴のため、手続き型言語のようなスタイルで記述される場合が多い。また、関数型言語の特徴である、第一級関数をサポートしている。これにより、プログラムをより簡潔に記述することができ、生産性の高さや、コードの可読性の高さに貢献している。基本的な仕様は ECMAScript [21] として標準化されており、それに加えて JavaScript 独自の仕様が加えられている。

JavaScript の主な用途として、Web アプリケーションにおけるコンテンツの生成及び動的な変更などが挙げられる。一方で、JavaScript は汎用プログラミング言語でもあるので、Web アプリケーション用途に限定されず、サーバーサイドで用いることができる。Google による既存の JavaScritpt 処理系 V8 [12] にネットワーク関連等のサーバサイド向け拡張を施した Node.js [20] 等がある。

近年、JavaScript 処理系は著しい進化を遂げている。以前はインタプリタ方式で実行されることが一般的であったが、実効速度を向上させるために、現在では JIT コンパイラ機構をはじめとした各種最適化がなされている。その結果、飛躍的にプログラムの実行速度が向上した。

2.1.2 特徴

JavaScript の型には、数値を表す `Number` 型、文字列を表す `String` 型、真偽値を表す `Boolean` 型が存在しそれらは基本データ型と呼ばれる。また、`Object` 型は複合データ型と呼ばれ、複数の基本データ型や複合データ型の集合である。関数を表す `Function` 型、配列を表す `Array` 型も `Object` 型の派生である。

JavaScript は型付けが緩く定義されている。変数やオブジェクトのプロパティに型を宣言する必要がなく、どのようなデータ型の値を代入してもよい。またその後、初めとは異なるデータ型の値を代入してもよい。さらに、演算時に期待されている型と異なる型のデータを渡した場合には、暗黙の型変換が行われる。このように、変数の型が静的に決まらず実行時に動的に決定するため、プログラムの処理内容が型により変化し一意に定まらない。

図 1 に、処理が一意に定まらない簡単な JavaScript プログラムの例を挙げる。図 1 のプログラムの関数 `func` は、引数の型により関数の処理が異なる。引数が `Number` 型の場合、`func(3)` は `4` を返すというように、数値同士の加算が行われる。また引数が `String` 型の場合、`func("a")` は `"a1"`を返すというように、文字列同士の結合が行われる。このように、JavaScript のプログラムでは、実行する直前まで処理が一意に決定せず、型変換が暗黙的に行われる。これは JavaScript プログラムの最適化を難しくしている要因の一つである。

また、JavaScript においてクローバル変数は、グローバルオブジェクトのプロパティという形

```
1 // JavaScript
2 var func = function(x){
3   return x + 1;
4 }
5
6 print(func(3));     // Output "4"
7 print(func("a"));   // Output "a1"
```

図 1　処理が一意に決まらない関数の例

で表現されている。this はメソッドのレシーバを表すが、レシーバを持たない関数においては、this はグローバルオブジェクトを参照する。これにより、グローバルオブジェクトを取得しプロパティを書き換えることによって、グローバル変数の書き換えが可能である。そのため、プログラムにおけるグローバル変数の有無や、型の特定が困難であるという特徴が挙げられる。

図 2 に、関数呼出し中にグローバル変数が変更される JavaScript プログラムの例を挙げる。関数 func は this によってグローバルオブジェクトを取得した後、第一引数で与えられたプロパティ名でグローバルオブジェクトのプロパティを決定し、第二引数に与えられた値でそのプロパティを書き換えている。これによって、グローバル変数が書き換わる。図 2 のプログラムは簡単な例ではあるが、大規模なプログラムになった場合、このようなプログラムに対して最適化を行うには静的に変数の生存期間を解析する必要があり非常に難しい。

上記で述べたような JavaScript の言語的な特徴のために、JavaScript のプログラムを最適化しようとした場合、プログラムを実行せずに構文上からその振る舞いを解析して行う静的な最適化は難しく限界がある。そのため、実行時における変数の型などの情報を用いた動的な最適化が JavaScript には有効である。

2.2　SSJSVM

SSJSVM (Server Side JavaScript Virtual Machine) [11] は、JavaScript バイトコードを解釈及び実行する、レジスタベースの仮想機械処理系である。SSJSVM で JavaScript コードを

```
1  // JavaScript
2  var x = 10;
3  var func = function(a, b){
4    var gobj = this;
5    gobj[a] = b;
6  }
7
8  print(x);  // 10;
9  func("x", "string");
10 print(x);  // string
```

図 2　グローバル変数を変更する関数の例

実行するためには、初めに SSJSVM コンパイラによって JavaScript コードを SSJSVM のバイトコードに変換する。SSJSVM はそのバイトコードを実行する。SSJSVM が対象とする言語は JavaScript のサブセットである。そのため、オブジェクトのプロパティを削除する delete 文、オブジェクトのプロパティを展開する構文である with 文、文字列を与えるとその文字列をプログラムとして実行する eval 関数には対応していない。

SSJSVM には、プログラム実行中の各種情報を格納する実行コンテキストが存在する。実行コンテキストには、グローバル変数等の各種変数情報や、実行中の命令の場所を示すプログラムカウンタ、またレジスタの情報等を格納するスタックを持っている。

SSJSVM には、既存処理系を参考にした幾つかの高速化手法が既に導入されている。まず、JavaScript Code [16] と呼ばれる Web フレームワーク内の JavaScript 処理系にあたる SquirrelFish Extreme [17]（以下、SFX）などに実装されている、インタプリタループの高速化手法である Direct Threading [4] の技法を実装している。これは、バイトコードを逐次実行する際に、各バイトコードの命令の次命令の処理が書かれたジャンプ先のアドレスを格納したテーブルを予め用意することよって、次の命令を実行する時のジャンプの手間を短縮したり、CPU の分岐予測を効きやすくする手法である。

ここで、switch 文を用いたインタプリタループについて、Direct Threading の最適化を適応した例を図 3 に示す。最適化前のプログラム（図 3(a)）では、switch 文で命令の種類に応じた処理を選択し実行するという、一連の動作をループする。最適化後のプログラム（図 3(b)）では、各命令に対応したラベルの飛び先を格納したジャンプテーブルを生成し、命令ディスパッチをして最初の命令に対応するラベルにジャンプする。各命令の最後には命令ディスパッチ、及びジャンプテーブルからジャンプするプログラムが挿入されている。ここで、&& はラベルのアドレスをとるという、GCC による C の拡張構文である。

更に、JIT コンパイラ機構に似た高速化手法として、バイトコードを動的に書き換える手法である Quickening [6] を用いた幾つかの最適化を実装している。まず、あるバイトコード命令において、処理の対象となるオペランドの型が固定していたり、あるいは型に偏りが確認できた場合、特定のオペランドの型に特化した命令に置換する。この最適化により、型による処理の分岐を時間をかけることなく行う事が可能になり、更に特定の型に特化した処理により実行速度の向上も望める。また、オブジェクトのプロパティ関連の命令について、プロパティの位置が固定、または殆ど変化がみられない場合、そのプロパティの位置を記録し、次にこの命令を実行する際には位置の取得にかかる時間を短縮する、インラインキャッシング [10] という最適化手法を実現している。これは、バイトコードに命令の書き換えの他に、プロパティの位置の情報についても命令に埋め込むことにより実現する。SSJSVM においては仕様上プロパティチェーンを辿る必要のない、グローバルオブジェクト関連、つまりグローバル変数の取得代入について実装している。

図 4 に Quickening によるインラインキャッシングを示す。(1) においてプロパティの要求をオブジェクトに対して行い、(2) においてそのプロパティの値と位置を取得する。このプロパティの位置を (3) で命令のオペランドに埋め込み、次にこの命令を実行する (4) の時、命令に埋め込んだ位置情報を用いて直接プロパティを取得する。

```
while(1){
  // 命令ディスパッチ
  instruction = instructions[pc];
  switch (getOpcode(instruction)) {
  case add:
    // add 命令の処理
    pc += 8; break;
  case sub;
    // sub 命令の処理
    pc += 8; break;
  ...
  }
}
```

(*a*) 最適化前

```
// 各命令の飛び先を格納する
jumpTable[] = { &&add, &&sub, ...};
start:
  instruction = instructions[pc];
  goto jumpTable[getOpcode(instruction)];
add:
  // add 命令の処理
  pc += 8;
  instruction = instructions[pc];
  goto jumpTable[getOpcode(instruction)];
sub:
  // sub 命令の処理
  pc += 8;
  instruction = instructions[pc];
  goto jumpTable[getOpcode(instruction)];
...
```

(*b*) 最適化後

図 3　Switch 文における Threaded Code の例

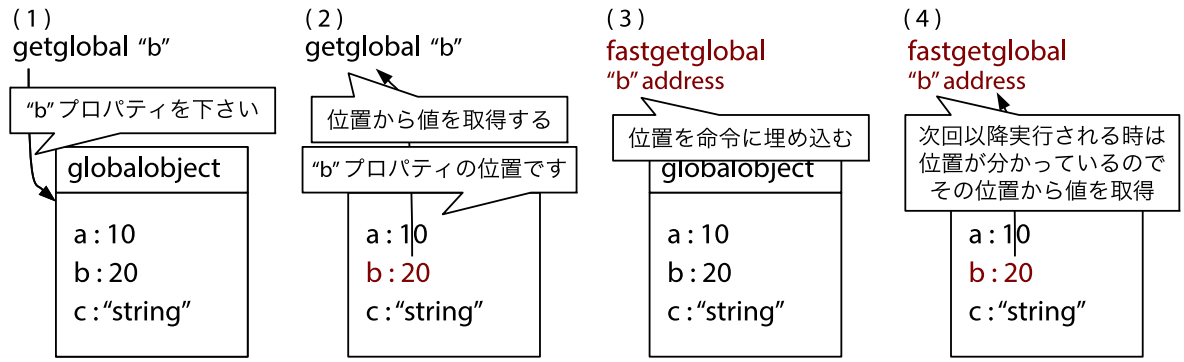

図 4　Quickening によるインラインキャッシング

2.3 LLVM

本研究では、幅広い言語に対応可能なコンパイラ基盤である LLVM [2] を JIT コンパイラ機構の実装に用いる。LLVM は、LLVM-IR という中間言語表現を対象にしており、LLVM-IR 向けの様々な最適化機構、及び様々な CPU アーキテクチャ向けのマシンコードへのバックエンドを提供している。また、LLVM は JIT コンパイラの作成支援機構を API として提供しており、最適化機構と合わせて用いる事で、高速な JIT コンパイラ機構の作成が可能である。LLVM は、C、C++ 等のコンパイラである Clang [15] のバックエンドなど、既に多くのプロジェクトにて用いられている。

2.3.1 **構成**

LLVM の構成を図 5 に示す。はじめに、各言語で記述されたプログラムを、それぞれの言語に対応するフロントエンドによって LLVM-IR に変換する。言語に依存する最適化は、各フロントエンドが行う。LLVM-IR のプログラムをリンカによって結合する。その結果生成されたコードに対して、LLVM が内部的に扱う中間言語に置き換えた後、手続き間最適化を含む様々な最適化を施す。そして、LLVM が対応する各アーキテクチャに対して、マシンコードにコンパイルする。また、マシンコードの実行時にプロファイルを取得し、分岐構文等の最適化を行うプロファイル最適化も行うことができる。

2.3.2 LLVM-IR

LLVM-IR は基本的な命令から構成されているアセンブリ言語に近い低級な言語設計である。しかし、基本的な命令による低水準な操作の組み合わせで、様々な高級言語に対応した簡潔な表現が可能である。数に制限が無く、値の代入が一箇所にでしか行えない特徴（静的単一代入、以下、SSA）を持つ仮想レジスタ（以下、静的無限レジスタ）を扱い、変数を表現する。LLVM-IR には様々な型の種類が存在各し、レジスタには型が対応している。例えば、32 ビット整数型を表す i32 型や、アーキテクチャに合わせた型として、128 ビット浮動小数点数型を表す fp128 型などがあり、細分化された型は高度な最適化を可能にする。また、LLVM は C 言語の標準ライブラリ等、多様なライブラリを提供している。

加算をし結果を表示するプログラムについて、C 言語、及びそのプログラムの変換後の LLVM-IR の例を図 6 に示す。LLVM-IR コードに C コードに、該当する部分をコメントとして記述し

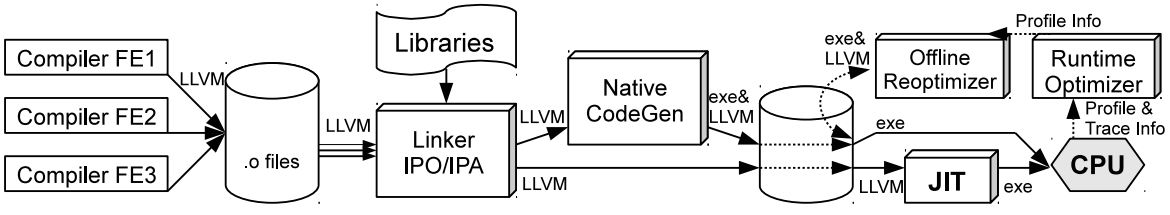

図 5　LLVM の構成図

```c
/* C */
int main(){
  int num = 3;
  printf("%d", num+1);
  return 0;
}
```

(*a*) C 言語

```
;; LLVM-IR
@.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
declare i32 @printf(i8*, ...)

;; int main()
define i32 @main() nounwind uwtable
  ;; int num
  %num = alloca i32, align 4
  ;; num = 3
  store i32 3, i32* %num, align 4
  ;; %1 = num
  %1 = load i32* %num, align 4
  ;; %2 = %1 + 1
  %2 = add i32 %1, 1
  ;; %3 = "%d\n"
  %3 = getelementptr [3 x i8]* @.str, i32 0, i32 0
  ;; %4 = printf(%3, %2)
  %4 = call i32 (i8*, ...)* @printf(i8* %3,  i32 %2)
  ;; return 0
  ret i32 0
}
```

(*b*) LLVM-IR

図 6　LLVM 言語表現の簡単な例

た。このように、記述量は多いものの、LLVM-IR の細かい命令の組み合わせで、C における振舞いを再現できる。

また、基本となる型と組み合わせて構成することができる派生型には、構造体、配列、ベクトルが存在する。また、例外処理を実現するための命令として、invoke 命令及び、unwind 命令が存在する。invoke 命令は関数呼出しの命令であり、同時に二つのラベルを渡す。一つは呼出した関数から戻ってきた時に飛ぶラベルであり、もう一つは例外発生時に飛ぶラベルである。unwind 命令は呼出し先の関数で利用する命令であり、例外を発生し、呼出し元の関数に巻き戻る命令である。また、例外処理の際に変数を受け渡すための組み込み関数も用意されている。

3 設計

3.1 プログラムの運用

SSJSVM は、サーバサイドでの用途を前提としている。一方、SFX や Google による V8[12] といった既存の JavaScript 処理系は、一般的にクライアントサイド用途を前提としている。このため、搭載されている JIT コンパイル機構の実行対象となるプログラムは、SSJSVM と SFX、V8 では異なると考えられる。

一般的には、クライアントサイドでは様々なプログラムを短期間、サーバサイドでは数種類のプログラムを長期間運用することが多い。JIT コンパイル機構において高度な最適化を行うことは非常にコストが高く、最適化にかかる時間も実行時間に含まれるために、どちらのサイドを前提とするかによって、採用する最適化手法を取捨選択する必要がある。一般にクライアントサイド向けの JIT コンパイル機構では、時間を要する最適化は敬遠され、単純ではあるが効果の認められる最適化手法を適度に採用する傾向にある。また、JIT コンパイル機構を多層にして、ホットスポット検出における実行回数の閾値を段階的に設定し、実行回数に合わせたより高度な最適化手法を採用しマシンコードに置換するといった方法をとることもある。

これに対し、本研究はプログラムの長期間の運用を前提としている。そのため、複数回実行される可能性のあるコードは、長期的な視野で多くの回数実行されると考えられる。これより、最適化にある程度の時間を要しても、実行の早い段階で十分な最適化を施したマシンコードを生成する意味があると考えられる。更に、Web アプリケーション向けの用途等を前提として考えると、実際に運用するコードが決まっているという特徴がある。プログラム起動直後は JIT コンパイル機構によるコード変換が発生するので、運用前からプログラムを起動しておき、ウォームアップをしておくことにより、実際の運用の際のオーバーヘッドを減らすという方法も考えられる。

また、長期的なプログラムの実行において、繰り返し実行されるコードの実行時間が、プログラム全体の実行時間であるとみなす。このため、本研究では、実行時に早めに、複数回実行される可能性のあるコードを高速なマシンコードに変換するように設計を行う。

3.2 ホットスポット

JIT コンパイラ機構のコード変換の対象となる頻繁に実行されるコードを、ホットスポットと呼ぶ。一般的にはインタプリタ実行中に、特定の関数やループが一定の閾値の回数を超えて実行された場合、そのコード部分をホットスポットとみなす。それは実行回数が少ないコードについて高速化することは効果がないという理由に基づくものである。

本研究では、サーバサイドでの運用を前提としており、プログラムが長期的に実行される。その際に、プログラム実行の立ち上がりの際にかかる初期化などの時間は、全体の実行時間に比べると非常に短い時間であるため無視する。実際にプログラムが立ち上がると、一部の関数やループが多くの回数実行される事が想定される。その際に低速なインタプリタ実行を避けるために本

研究では、ユーザ定義関数の呼出しが発生した際に、直ちに JIT コンパイル機構によりマシンコードの生成を行う方針を取る。

一般にユーザ定義関数をホットスポットとみなす場合、プログラムの実行前にプログラム全体をコンパイルする Ahead Of Time (以下、AOT) コンパイルも考えられる。AOT コンパイルでは、ユーザ定義関数内部でグローバル変数などの関数外で定義された変数を用いる場合、その変数の型などの実行中に決定する要素への対処が必要となる。JIT コンパイル機構では、実行時情報を用いた仮定を用いて対処できるが、AOT コンパイルではそのような対処法をとることはできない。そのため、その仮定を吸収するコードを生成するか、または全ての可能性毎に別のコードを生成するかのどちらかが必要になる。前者のケースでは、プログラムの実行速度を犠牲にしてしまう可能性があり、後者のケースでは、参照する外部変数の数による、組み合わせ爆発が発生してしまうという問題がある。そのため、本研究では AOT によらず、JIT コンパイラ機構を採用した。

3.3 コード生成

本研究の JIT コンパイラ機構は、直接マシンコードに変換をするのではなく、まず初めに最適化や簡単のために、LLVM-IR へ変換する。LLVM-IR は低級な言語表現であるため、高級な言語である JavaScript とは相性が良くないように思われるかもしれない。しかし、実行時情報や、その情報を元にした型推論を行うことによって十分な型情報が与えられば、高速化したマシンコードの生成が可能である。本研究では、LLVM の提供する API を用いて、プログラム実行中に JavaScript バイトコードを LLVM-IR へ変換して、その後 LLVM が LLVM-IR をアーキテクチャに合わせたマシンコードへと変換する。

本研究の JIT コンパイラ機構では、LLVM の最適化機構が適切に働くような LLVM-IR を生成しなければならない。Clang では、C、C++ のプログラムに対して特殊な変換を加えずに、記述された構文に近い形で LLVM-IR に変換する。本研究においても、SSJSVM バイトコードに対し複雑な変形をせずに LLVM-IR に変換する。本研究では、バイトコードの命令単位で LLVM-IR 変換を行う、SFX において採用されている Context Threading [5] JIT の生成方針をとり、生成されたプログラムは余分な制御構造を排除された素朴な制御構造にする。

LLVM の最適化機構が十分に働くためには、ある程度の最適化手法を JIT コンパイラ機構においても実装する必要がある。SSJSVM 内部では、JavaScript の変数は全てボクシングされた型で扱っている。これをそのまま変換すると型の特殊化がされていないため、変数の型の確認や、アンボクシングをするための処理等が挟まれることになり、LLVM の最適化が上手く働かない上に、余分な処理を行うことになる。そのため、本研究の JIT コンパイラ機構では、インタプリタ関数との橋渡し以外では、型の特殊化を行い変数をアンボクシングした型で扱う。

3.4 仮定の失敗

JIT コンパイル機構が生成するマシンコードには、マシンコード生成時の環境をもとにした仮定が多く存在する。下にその一部を示す。

- グローバル変数の型を特定の型と定める仮定
- オブジェクトのプロパティの型を特定の型と定める仮定
- 関数呼出しにおける呼出し先の関数を特定の関数と定める仮定
- 整数の演算におけるオーバーフローは生じないという仮定

マシンコードの実行中にそのマシンコードが置いている仮定が外れてしまった場合、マシンコードではなくインタプリタ実行に戻る必要がある。その際に、どのような仮定に外れたかを記録しておく。その後、JIT コンパイル機構が以前実行中に仮定に外れたことのあるマシンコードを実行する前に、以前外れた仮定も含めて実行環境が一致するかを確認し、一致すればそのマシンコードを実行し、一致しない場合は新たに現環境に基づいた新たなマシンコードの生成を試行する。

図 7 に仮定に失敗した時の処理を示す。オブジェクトのプロパティの型の仮定に失敗した場合、次の同じコードを実行する時に、その時の環境に応じたマシンコードを新たに生成する。

3.5 再現性

JavaScript の演算子と LLVM-IR における演算子の振舞いは、たとえば整数どうしの加算であっても厳密には異なる。LLVM-IR で JavaScript における振舞いと厳密に一致させるためには、単純に演算するよりもはるかに時間がかかる場合がある。しかし、加算等の単純な命令において、LLVM-IR と JavaScript の振舞いが異なる場合は、オペランドの値が非常に大きな値であるなど特殊な時でだけである。そのため、本処理系では通常の JavaScript の動作とは別に、完全に JavaScript の振舞いを再現はしないが、LLVM-IR でに高速に処理できる形にコンパイルを行うオプションを設けた。

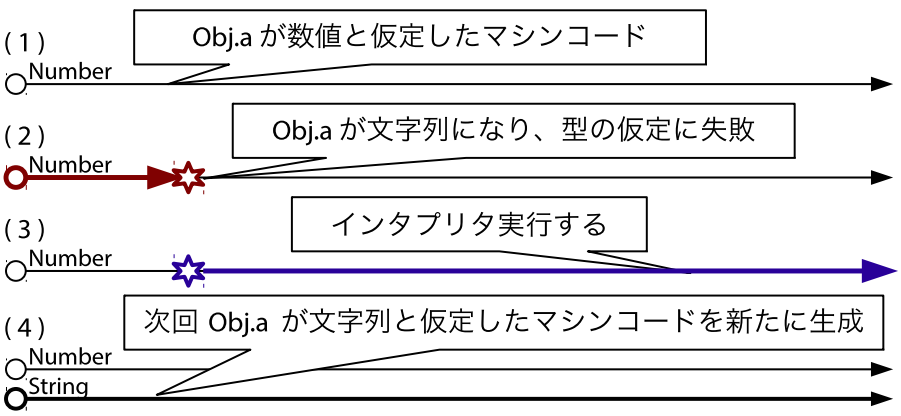

図 7　仮定の失敗処理

4　実装

本章では本研究で実装した処理系の全体構成、及び JIT コンパイル機構についての詳細を述べる。本研究で作成したバイトコード向けの型推論機構、及びバイトコードを LLVM-IR へ変換するコード生成機構、及び生成の際に行った最適化、また生成後の LLVM-IR に用いた LLVM による最適化について各々解説する。

4.1　処理系の構成

図 8 に本処理系の処理の流れを示す。SSJSVM は、既存の SSJS コンパイラにより JavaScript ソースプログラムからコンパイルされた独自のバイトコードを対象に動作する。本研究では、ユーザ定義関数に対応するバイトコード列を JIT コンパイル機構の対象とするため、関数呼出し命令の実行をトラップし、呼出された関数本体のバイトコードを LLVM-IR に変換する。その際に LLVM において十分な最適化が行えるように、幾つかの最適化を施す。その後、LLVM によって、LLVM-IR に幾つもの最適化が施された後マシンコードへコンパイルされ実行される。

4.2　型推論器

LLVM による十分な最適化を可能にするためには、SSJSVM のバイトコードから十分な型情報を得る必要がある。そのためにバイトコードに対する型推論機構を実装した。この型推論機

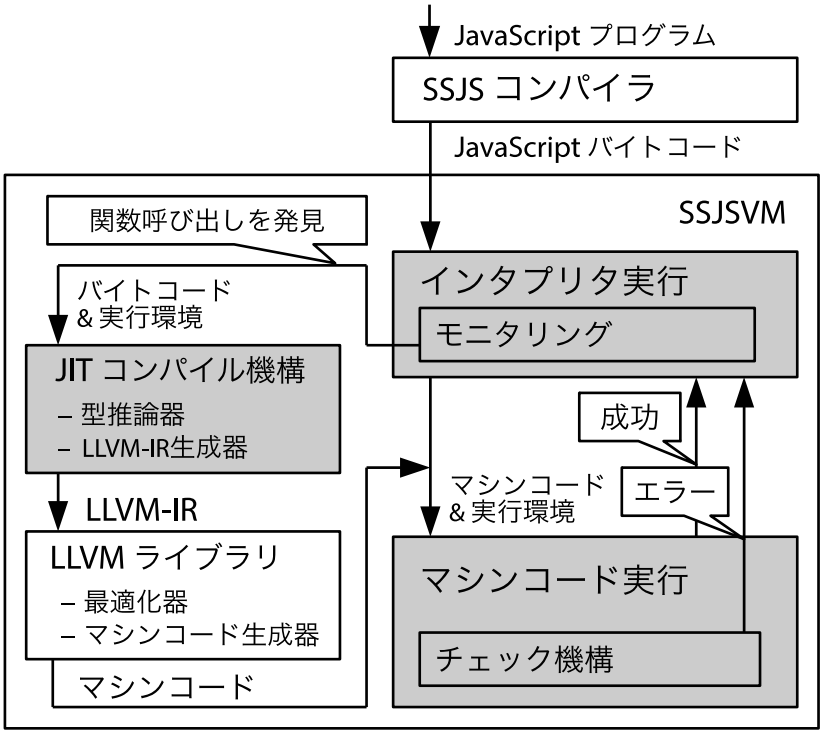

図 8　本処理系の処理

構は Click らによる方法 [9] に基づいている。まず、プログラムの制御フローグラフを作成し、プログラムの基本ブロックを生成する。その後、レジスタや変数等に対応するノードを生成し、各々のノード間の型の依存関係をグラフ化する。その後に型情報を命令とそのオペランドに合わせて伝播させる。全てのノードの波及を済ませた時点で、どれか一つのノードでも型の情報に変化が見られる場合、もう一度全てのノードについて型情報を波及させる。全てのノード変化が見られなくなった時に型推論を終了する。

以下に型推論の例を示す。図 9 は対象となるプログラムを、図 10 は図 9 で引数 `arg` に int 型の値が与えられた場合に対する型推論の動作例を示している。図 10 でグレーで示す部分は実行前に既に型が決定している部分である。変数等の型が不確定な部分を決定するため、まず (1) では、型推論に関連する命令に対して、型の関連性を表すノードを生成している。(2)〜(5) では、それらノードを順に処理して変数の型を判断していく。(5) では、整数どうしの除算についての処理を行い、JavaScript において、整数どうしの除算結果は浮動小数点数もありうるため、結果として `float`（浮動小数点）型に型推論される。その後、変化があるまで一連のノードを順に処理し、(6) の処理を終えた以降、もう一度順に処理を行っても変化がなかったため、ここで型推

```
1 var f = function(arg){
2   var a = arg;
3   var b = a + 2;
4   a = (a + b) / 2;
5   return a;
6 }
```

図 9　型推論のプログラム例

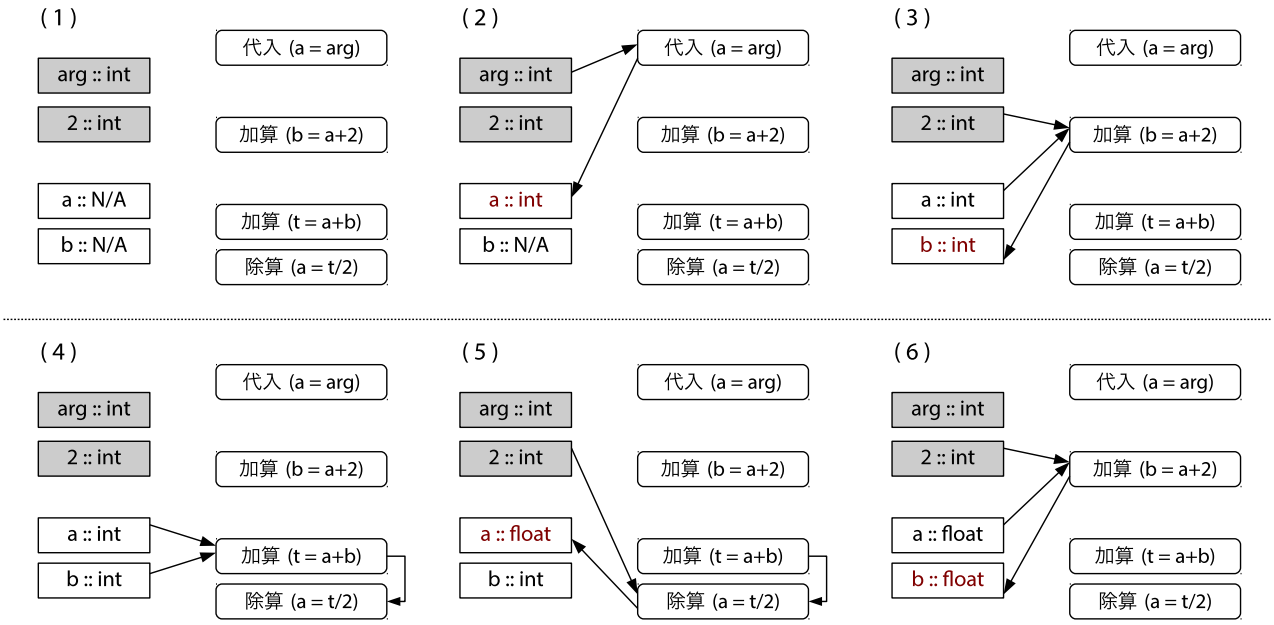

図 10　型推論機構の動作例

論が収束し、a、b の変数の型がどちらも float 型だということが判明した。

4.3 LLVM-IR 生成器

SSJSVM バイトコードからの LLVM-IR の生成に際しては、各バイトコードに対応する LLVM-IR コードを生成する。バイトコードはレジスタベースで、SSA 形式に沿っているため、同一の設計思想である LLVM-IR へは、複雑な過程を経ることなく変換することが可能である。変換の際はロードストア命令等、時間を要する命令も積極的に用い、高速なコードよりも LLVM が構造を解析しやすいコードを、Clang 等の既存のフロントエンドを参考に出力した。

4.4 実行コンテキスト

プログラム中に関数 f 内で定義した関数 g が、関数 f の局所変数を参照する場合がある。JIT コンパイル機構では局所変数はレジスタのみで扱う方が高速化が見込めるので、実行時の各種変数の情報を格納する実行コンテキストには、マシンコード実行時における局所変数の情報は記録されていない。このため、呼び出す関数 g で現在の関数 f 内の局所変数を用いることが判明したケースについては、その関数 g の実行に遷移する前に、局所変数等の情報を実行コンテキストに記録するコードを埋め込む。

4.5 最適化機構

本研究で LLVM-IR 生成器に実装した幾つかの最適化手法を紹介する。ここでは JavaScript に依存する最適化を実装した。これは、言語非依存の最適化機構は LLVM が提供しているからである。また JavaScript に依存する最適化を行うことによって、LLVM の最適化機構が効くようにするという意図もある。

型のアンボクシング化 SSJSVM では JavaScript の型をボクシングした型で扱っている。そのため、型に応じて処理を振り分けるのに、命令毎にまず型のチェックを行い、その命令の対象となる変数をアンボクシングし、処理を行ったあとボクシングするという手間があった。しかし、コード生成時には型が決定しており、型チェックをする必要がないため、アンボクシングされた LLVM-IR の型で処理を行う。図 11 にこの最適化の擬似プログラムを示す。

値の事前型変換 型に依存して処理が定まる場合、値の型変換が生じる場合がある。例えば、文字列と数値の加算は、数値を文字列に変換した上で文字列結合を行う。ここで数値が定数の場合、その定数に対する型変換を事前に施してコードを生成すると高速化が望める。図 12 にこの最適化の擬似プログラムを示す。

関数の詳細化 関数呼出しについて、引数の型の組合せ毎に関数を生成する最適化を実装した。プログラムの実行が進むにつれてある関数に渡す引数の型が変化する場合がある。その場合、以前とは別の、現在の引数型に特殊化したマシンコードを生成する。それにより特定の型に特殊化された関数を呼び出せるので、より高速な実行が可能になる。図 13 にこの最適化の擬似プログラムを示す。

関数の引数のアンボクシング化 関数呼出しの際の引数について、ボクシング及びアンボクシングの除去の最適化を実装した。関数呼出しにおいて、引数として値を渡す際に、それらの値をボクシングした値の配列として渡していた。しかし、それには時間を要すため、インタプリタからの入り口の関数である、ボクシングされた型を引数として受け取るエントリ関数と、基本となるデータ型を引数として受け取る効率化された関数の二つを用意して高速化を図る。図 14 に基本となるデータ型を引数として受け取る関数を示す。

オブジェクトのプロパティ取得の除去 オブジェクトプロパティ取得についての不要コードの除去の最適化を行った。実装した JIT コンパイル機構において、処理が複雑な命令については、インタプリタの関数を呼ぶ。その一例としてオブジェクトのプロパティ取得が挙げられる。LLVM による最適化は、LLVM-IR のみを解析するため、インタプリタの関数を呼出す際、その関数における副作用の有無などの関数の特性を LLVM の最適化機構は知ることができない。そのため、共通部分式であり除去が可能であるコードでも、その間に関数呼出しがある場合、LLVM の最適化では除去することができない。この最適化では構造を確認し、ジャンプ文等の出現しない基本ブロック内におけるオブジェクトプロパティの取得について重複するコードを纏める。この際、オブジェクトのプロパティの代入が生じた場合、その後はもう一度改めて取得し直すように

```
1 double val1 = JSValueToDouble(jsval1);
2 double val2 = JSValueToDouble(jsval2);
3 double res = val1 + val2;
4 JSValue jsres = DoubleToJSValue(res);
```

(*a*) 最適化前

```
1 double res = val1 + val2;
```

図 11　型のアンボクシング化

```
1 function(char *string){
2   char *tmp = intToString(1);
3   char *result = concatStr(string, tmp);
4 }
```

(*a*) 最適化前

```
1 function(char *string){
2   char *result = concatStr(string, “1”);
3 }
```

(*b*) 最適化後

図 12　値の事前型変換

する。

図 15 にこの最適化の擬似プログラムを例で示す。最適化前では、必要になる度に Propget によりプロパティを取得している。最適化後のプログラムでは、再利用できる可能性のある部分まで、副作用の可能性を排除できた場合、そのプロパティを使いまわすことにより、無駄を排除している。

オブジェクトのインラインキャッシング オブジェクトについても、SSJSVM には既に実装されているインラインキャッシングを実装した。SSJSVM では、バイトコード命令を置換し取得するプロパティの位置情報を埋め込むことにより実現していたが、本 JIT コンパイラ機構においてはマシンコードを書き換えることはできない。そこで、プログラムコード自体に、オブジェクトのプロパティの位置情報を持つ変数を用意する。オブジェクトの構造が変化しない限り、その変数の値を更新することなしに、プロパティの取得が可能になる。そのため、さらにプロパティの構成に対して一意の番号を持つ変数を用意する。それら二つのを用いる事で、プロトタイプチェーンを辿らないプロパティ取得についてインラインキャッシングを実現した。SSJSVM ではグローバルオブジェクトのみの高速化を行っているが、本処理系ではローカルのオブジェクトに対してもこの最適化を導入した。

```
1 JSValue r1 = f(context, doubleToJSValue(arg1_1));
2 JSValue r2 = f(context, stringToJSValue(arg1_2));
```

(*a*) 最適化前

```
1 // 浮動小数点数用の関数
2 JSValue r1 = f_d(context, doubleToJSValue(arg1_1));
3 // 文字列用の関数
4 JSValue r2 = f_s(context, stringToJSValue(arg1_2));
```

(*b*) 最適化後

図 13　関数の詳細化

```
1  // ボクシング化した関数
2  JSValue argv[argc];
3  argv[0] = IntToJSValue(argv1);
4  argv[1] = IntToJSValue(argv2);
5  JSValue (fptr *)(Context*, JSValue*);
6  JSValue retv = fptr(context, argv);
7  int ret = JSValueToInt(retv);
8
9  // 追加する関数
10 int (fptr *)(Context *, int);
11 int ret = fptr(context, argv1, argv2);
```

図 14　関数の引数のアンボクシング化

図 16 にオブジェクトのインラインキャッシングにおける、インタプリタ関数の内部処理の変更を示す。最適化前の関数はオブジェクトとプロパティ名を引数で受け取り、そのオブジェクトのプロパティテーブルの何番目に目的のプロパティがあるか取得し、そのプロパティを返す関数である。最適化後の関数は加えてオブジェクトの構成に対して与えられるバージョン番号と前回の実行におけるプロパティの位置をそれぞれ引数で受け取る。現在のバージョン番号と引数のバージョン番号が一致した場合、前回の結果をそのまま返す。それ以外の場合、最適化前と同様に動作した後、プロパティを返す前に、現在の構成の番号とそのプロパティの位置とを各々記録

```
1 JSValue p1 = PropGet(obj, “str”);
2 print(p);
3 JSValue p2 = PropGet(obj, “str”);
4 char *string = JSValueToCStr(p2) + “1”;
```

(a) 最適化前

```
1 JSValue p = PropGet(obj, “str”);
2 print(p);
3 char *string = JSValueToCStr(p) + “1”;
```

(b) 最適化後

図 15　オブジェクトのプロパティ取得の除去

```
1 JSValue getprop(JSValue obj, char *str){
2   ...
3   ind = getIndex(obj,str);
4   return prop[ind];
5 }
```

(a) インタプリタ内部の関数の最適化前

```
1 JSValue getprop(JSValue obj, char * str, long *ver, JSValue *ptr
    ){
2   // 構成番号が一致した場合、前回の結果を使い回す
3   if(*ver == getVersion(obj)){ return *ptr; }
4   ...
5   ind = getIndex(obj,str);
6   *ver = getVersion(obj);
7   *ptr = &prop[ind];
8   return prop[ind];
9 }
```

(b) インタプリタ内部の関数の最適化後

図 16　オブジェクトのインラインキャッシング

する。

4.6　仮定の失敗によるアボート処理

仮定の失敗によるアボート処理では、インタプリタ実行に戻る必要がある。SSJSVM において、実行時の情報は実行コンテキストに格納されるが、JIT コンパイル機構の生成したマシンコード実行時では高速化のために、実行時情報を全て実行コンテキストに格納することはない。そのため、マシンコード実行中にアボート処理をする場合、インタプリタ実行に戻るようにするため、実行コンテキストに実行時情報を追加しなければならない。

実行コンテキストに書き戻す内容として、ローカル変数の情報及び、各レジスタの値が挙げられる。しかし、バイトコードは細かい単純な命令で構成され、また SSA 形式を採用しているためにレジスタの数が多く、仮定に失敗した後に不要なレジスタも存在する場合が多く、全てのレジスタの情報を実行コンテキストに書き込むのは現実的ではない。このため、コード生成時にレジスタの依存関係からグラフを生成し、仮定の失敗した箇所以降で必要十分なレジスタの集合を割り出し、それらだけを書き戻すコードを挟み込む。

図 17 にレジスタの書き戻し処理の例を示す。この例ではオブジェクトのプロパティの型について、特定の型と決め付けた仮定が失敗した際の書き戻すレジスタを判別している。レジスタの依存関係のグラフから、仮定に失敗した状態をまたいだ依存関係を探索し、その依存関係の解決に必要なレジスタを書き戻すようにする。

また、仮定の失敗によるアボート処理で、仮定に失敗した環境でのインタプリタ実行に戻るのではなく、仮定が失敗した関数が実行される前の環境でインタプリタ実行に戻りたい場合がある。例えば、対象の関数について、既に何度も実行されている場合、直ぐに現在の実行環境に合わせてマシンコードを生成したい場合である。そのようなアボート処理に備えて、実行コンテキストに対する上書きが発生する可能性のある部分について、上書き前に元のデータをどこかに退避させる必要がある。

たとえば、オブジェクトプロパティに対しての書き込み等がそれに該当する。正常にプログラムが実行された場合は、退避したデータを破棄する。アボートが発生した場合は、退避したデー

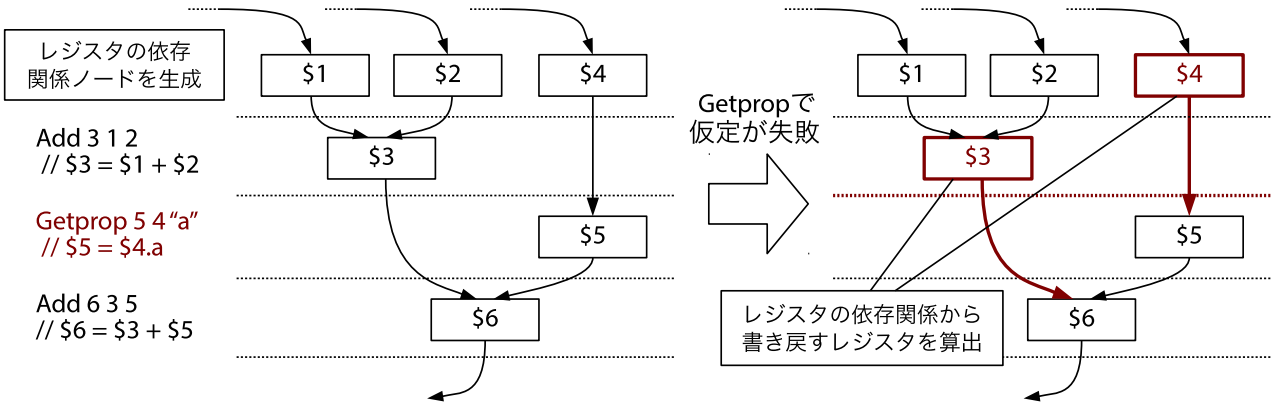

図 17　レジスタの書き戻し処理

タを実行コンテキストに上書きする。これにより、関数の実行前の環境に戻ることができる。しかし、このような処理ができる関数は限られている。オブジェクトプロパティへの書き戻み以外にも副作用を及ぼす可能性のあるコード、例えば標準出力に情報を出力した場合等はこのアボートによる実行巻き戻しはできない。

図 18 にオブジェクトプロパティの書き戻し処理の例を示す。この例では、プロパティに値を書き込んでいるが、仮定の失敗を考慮して、上書きされる前の値をグレーで示している退避領域に書き込んでいる。インタプリタ実行に戻った際には、退避領域のデータは破棄される。

4.7 LLVM による最適化

LLVM には LLVM-IR に対し、幾つもの最適化をオプションを用意している。本処理系ではそれを利用して、生成した LLVM-IR に最適化を施す。LLVM による最適化には大きく二つのフェースがある。

第一に、LLVM-IR を、高速な LLVM-IR にコード変換を行うもので、LLVM により提供されている各種最適化手法を選択し適用することができる。本研究では JIT コンパイラ機構において有効性が既に認められている最適化手法 [7] を積極的に選択し適用した。以下にいつくか紹介する。

まず、不要コードの除去の最適化を適用した。SSJSVM バイトコードから LLVM-IR へのコード生成において、一部を除き不要コードを除去していないためである。この最適化により、例えば、用いられない変数のロードは不要コードとして除去することができる。

次に、関数インライニングの最適化を適用した。SSJSVM バイトコードから LLVM-IR へのコード生成において関数単位でのコード生成を行い、関数間における最適化は施していない。しかしながら、呼出し先の関数が短い場合など、関数呼出しをその関数の定義と置き換えることにより、関数内部を含めた最適化が可能になり、さらに関数呼出しのオーバーヘッドも削減できる。この最適化手法を関数インライニングと呼ぶ。図 19 にその最適化の JavaScript における擬似プログラムを示す。

次に、ロードストア除去の最適化を適用した。LLVM-IR のコード内では、バイトコードにお

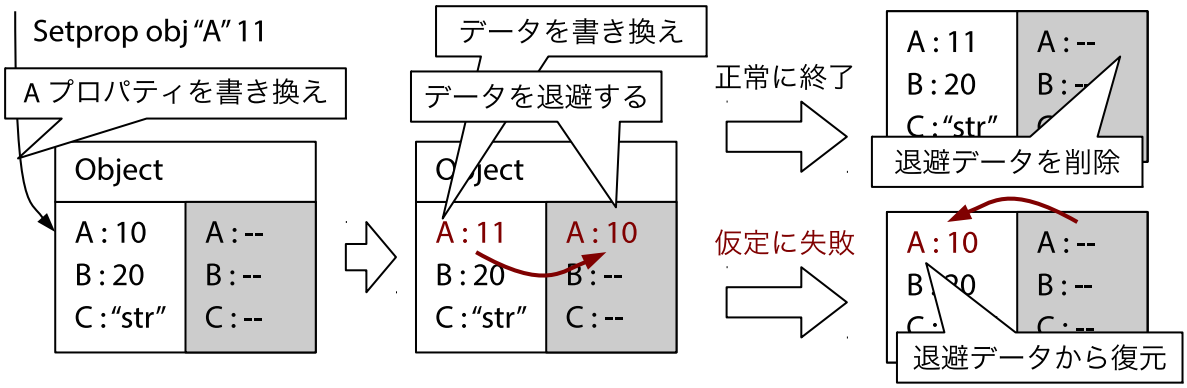

図 18　オブジェクトプロパティの書き戻し処理

ける変数を扱うために、変数用のメモリを確保してそこにデータをロード/ストアする。しかしこの方法はメモリを使用するために、オーバーヘッドを伴う。このため、LLVM の最適化機構では、ロードストア命令を除去して、レジスタで扱うようにする。そのために、有効なレジスタを選択する phi 関数を用いる。phi 関数は、遷移元のブロックを判別して、対応する有効なレジスタを取得する関数である。これにより、その変数に対応する最新のレジスタを取得することが可能になり、ロードストア命令の除去を行う事ができる。

この最適化の一例を図 20 に示す。この最適化では変数 a を、ロードストアを用いずに、レジスタで表現できている。phi 関数によって、最新の a の値を持つレジスタを遷移元から判断し、適切な値をレジスタに置くことができる。

また、ピープホール最適化を適用した。コード生成機ではバイトコードの命令に沿った一般的なコードを生成するが、特殊な条件下の場合、別に最適なコードを用いる事ができる場合がある。ピープホール最適化とは、命令列を走査していき、一定のパターンを見付け、それをより高

```
1 var func = function(a){
2   return -a;
3 }
4 var res = 1 + func(1);
```

(a) 最適化前

```
1 var res = 1 + (-1);
```

(b) 最適化後

図 19　関数インライニング

```
1 ;; LLVM-IR
2 ;; 最適化前
3 %3 = add i64 %1 %2              ;; $3 = $1 + $2
4 store i64 %3, i64* %a           ;; a = $3
5 jump %4                         ;; jmp label:%4
6 %4:                             ;; label:%4
7 %5 = load i64* %a               ;; $5 = a
```

(a) 最適化前

```
1 %3 = add i64 %1 %2              ;; $3 = $1 + $2
2 jump %4                         ;; jmp label:%4
3 %4:                             ;; label:%4
4 %5 = phi [%3, %0], [%11, %4]    ;; $5 = $3 or $11
```

(b) 最適化後

図 20　ロードストア命令の除去

速に実行できるであろうパターンに置換していく最適化手法でである。単純に 1 命令についてより良い命令に置換する場合もあれば、複数の命令の列を置換する場合もある。

図 21 にその最適化の JavaScript における擬似プログラムを示す。このプログラムにおいて、ピープホール最適化により、1 との乗算が単純な代入に置換され、更に 2 との乗算が左シフトに置換され、各々より高速な文に置換されていることが分かる。

第二に、LLVM-IR からマシンコードを生成する際の最適化がある。LLVM では、LLVM-IR からマシンコードを生成する時に、その最適化度を指定することが可能である。これでは最適化手法ごとの取捨選択など細かな指定はできず、大まかにどの程度の最適化を行うかしか指定できない。最適化度は、NONE、LESS、DEFAULT、AGGRESSIVE の四段階がある。TESSA [1] においては、LESS 以降は最適化にかかる時間が実行時間の高速化に見合わないという理由により、LESS を採用している。一方で、本研究ではサーバサイド用途を考えるため、時間を要しても十分に高速化したコードを生成することを目的にしているため、AGGRESSIVE に設定した。

```
1 // 最適化前
2 var b = a * 1;
3 var c = b * 2;
```

(*a*) 最適化前

```
1 var b = a;
2 var c = b << 1;
```

(*b*) 最適化後

図 21　ピープホール最適化

5 実行例

本章は、本研究の JIT コンパイル機構が行うコード変換の例を示す。図 22 は JIT コンパイル対象の関数である。この関数は、0 から与えられた引数未満の数値までの和を返す。

5.1 SSJSVM バイトコード

まず、SSJSVM コンパイラによって対象のコードをバイトコードに変換する。そのバイトコードを図 23 に示す。

SSA 形式の単純なバイトコードである。定数命令、四則演算命令、制御命令やなどの各種取得代入命令など、基本的な細かい操作毎に命令が用意され、それらを組み合わせる事で JavaScript のプログラムを構成する。局所変数やグローバル変数には取得代入する命令も用意されており再代入が許されている。これらの特徴から、このバイトコードは解析が比較的容易である。ジャンプ、及び条件ジャンプ命令での飛び先は、現在のプログラムの実行位置との相対位置で表現されており、バイトコードにはラベルは存在しない。また、関数の返り値は seta 命令により特殊なレジスタに格納される。

5.2 LLVM-IR 生成器による変換

次に、LLVM-IR 生成器によって生成された LLVM-IR のコードを示す。図 23 のバイトコードを LLVM-IR に変換すると、図 24 に示すコードになる。プログラムの初めの方では、必要となるローカル変数の領域を確保し、引数の値をロードしている。その後、ローカル変数の値を初期化した後、for 文の継続条件式にあたるブロックにジャンプする。ラベル 12 以降では、計算に必要となる変数をロードし、計算を行い、必要に応じてストアするという、バイトコードにも見られる命令列をそのまま LLVM-IR に変換している。ラベル 12 の最後にあるジャンプによって、for 文の継続判定が行われる。このように、LLVM-IR では制御構文をブロックと呼ばれるコードの集合で表現し、その最後には飛び先を指定したジャンプ文、又はリターン文が必要になる。ラベルにはジャンプ元のブロックを示す注釈が記載されており、これらの情報を元に LLVM は最適化を行う。

```
1 var f = function(arg){
2   var sum = 0;
3   for(var i=0; i<arg; i++){
4     sum += i;
5   }
6   return sum;
7 }
```

図 22　対象のコード

```
1  callentry 0                    # 関数に関連する情報
2  sendentry 1                    # 関数に関連する情報
3  ...（関数実行の準備）...
4  fixnum 2 0                     # $2 = 0
5  setlocal 0 2 2                 # $local[2] = $2
6  fixnum 5 0                     # $5 = 0
7  setlocal 0 3 5                 # $local[3] = $5
8  jump 9                         # jump after 9
9  getlocal 8 0 2                 # $8 = $local[2]
10 getlocal 9 0 3                 # $9 = $local[3]
11 add 10 8 9                     # $10 = $8 + $9
12 setlocal 0 2 10                # $local[2] = $10
13 getlocal 11 0 3                # $11 = $local[3]
14 fixnum 13 1                    # $13 = 1
15 add 12 11 13                   # $12 = $11 + $13
16 setlocal 0 3 12                # $local[3] = $12
17 getlocal 16 0 3                # $16 = $local[3]
18 getarg 17 0 0                  # $17 = $arg[0]
19 lessthan 18 16 17              # $18 = $16 < $17
20 jumptrue 18 -11                # if $18 jump after -11
21 getlocal 20 0 2                # $20 = $local[2]
22 seta 20                        # set return val $20
23 ret                            # return
```

図 23　SSJSVM バイトコード

最後に、LLVM が LLVM-IR に各種最適化を施した後のコードを図 25 に示す。ロードストア除去の最適化により、ローカル変数がレジスタで扱われるようになっているのが分かる。ラベル 6 のブロックのはじめにおける phi 命令により、変数 sum と i について遷移元のブロックを参照し、有効な値を選択している。ローカル変数をレジスタのみで扱うことにより、ロードストアの時間が短縮され、大幅な高速化に繋がる。更に、不要コード除去により、用いられなかった値の取得などのコードが消され、ピープホール最適化により、ラベル 8 のブロックの返り値の変数に型のタグ情報を付加している命令が、加算から一般的な CPU においてより高速に実行されるであろう OR 命令に置換されていることが分かる。このように、LLVM による最適化を施すと、非常に洗練されたコードが生成される事が分かる。

```
1  define i64 @f4879_(%struct.contextCell* %context, i64* %argv0) {
2    %l = alloca i64                              ;; 変数宣言 long sum
3    %l1 = alloca i64                             ;; 変数宣言 long i
4    ... (不必要なコード) ...
5    %a = alloca i64                              ;; 変数宣言 long arg
6    %4 = load i64* %argv0                        ;; 引数の値をロード
7    %5 = ashr i64 %4, 3                          ;; 引数をアンボクシング
8    store i64 %5, i64* %a                        ;; 引数の値をストア
9    store i64 0, i64* %l                         ;; sum = 0
10   store i64 0, i64* %l1                        ;; i = 0
11   br label %12                                 ;; jump %12
12
13 ; <label>:6                                    ; preds = %12
14   %7 = load i64* %l                            ;; %7 = sum
15   %8 = load i64* %l1                           ;; %8 = i
16   %9 = add i64 %7, %8                          ;; %9 = %7 + %8
17   store i64 %9, i64* %l                        ;; sum = %9
18   %10 = load i64* %l1                          ;; %10 = i
19   %11 = add i64 %10, 1                         ;; %11 = %10 + 1
20   store i64 %11, i64* %l1                      ;; i = %11
21   br label %12                                 ;; jump %12
22
23 ; <label>:12                                   ; preds = %6, %0
24   %13 = load i64* %l1                          ;; %13 = i
25   %14 = load i64* %a                           ;; %14 = arg
26   %15 = icmp slt i64 %13, %14                  ;; %15 = %13 < %14
27   br i1 %15, label %6, label %16               ;; jump (%15)? %6 : %16
28
29 ; <label>:16                                   ; preds = %12
30   %17 = load i64* %l                           ;; %17 = sum
31   %18 = shl i64 %17, 3                         ;; 返り値をボクシング
32   %19 = add i64 %18, 7                         ;; 返り値の変数の型付け
33   ret i64 %19
34 }
```

図 24　LLVM-IR への変換例

```
1  define i64 @f4879_(%struct.contextCell* %context, i64* %argv0) {
2    %1 = load i64* %argv0                          ;; 引数の値をロード
3    %2 = ashr i64 %1, 3                            ;; 引数をアンボクシング
4    br label %6                                    ;; jump %6
5
6  ; <label>:3                                      ; preds = %6
7    %4 = add i64 %l.0, %l1.0                       ;; %4 = %l.0 + %l1.0
8    %5 = add i64 %l1.0, 1                          ;; %5 = %l1.0 + 1
9    br label %6                                    ;; jump %6
10
11 ; <label>:6                                      ; preds = %3, %0
12   %l1.0 = phi i64 [ 0, %0 ], [ %5, %3 ]         ;; %l1.0 = 0 or %5
13   %l.0 = phi i64 [ 0, %0 ], [ %4, %3 ]          ;; %l.0 = 0 or %4
14   %7 = icmp slt i64 %l1.0, %2                    ;; %7 = %l1.0 < %2
15   br i1 %7, label %3, label %8                   ;; jump (%7)? %3 : %8
16
17 ; <label>:8                                      ; preds = %6
18   %9 = shl i64 %l.0, 3                           ;; 返り値をボクシング
19   %10 = or i64 %9, 7                             ;; 返り値の変数の型付け
20   ret i64 %10
21 }
```

図 25　LLVM による最適化後のコード

6 評価

サーバサイド環境を想定にしたベンチマークを用いて本研究の JIT コンパイル機構を評価した。Web サーバでは、要求に対して返答を返す際に、処理の時間よりも I/O の時間がネックになる場合が多い。一方で、収集したデータを用いて統計処理をしたり、長時間にかかる処理を行う事もあり、その際には処理時間が問題になってくる。そのため、ここでは表 1 のような、数値計算を中心とした幾つかのベンチマークを用いた。表実験環境は、Mac OSX 10.7.5 (intel i5 1.7GHz, 4GB RAM)、LLVM version 3.2 を用いた。

6.1 JIT コンパイラ導入の効果

SSJSVM に JIT コンパイル機構を実装した効果について検証する。本処理系について JIT コンパイラ機構を有効にしている場合と、無効にしている場合について実行時間を比較した。

結果を図 26 に示す。この図は、本研究の JIT コンパイル機構を用いない場合に対する実行速度比を示している。全体的に、JIT コンパイル機構により、大幅な実行速度の向上が見られた。SSJSVM のバイトコードは非常に細かい単位での命令で構成されているため、JIT コンパイラ機構において特別な最適化をしない場合においても、命令ディスパッチの手間が省けることによる高速化が望める。加えて本研究では、様々な最適化を加えることにより、大幅な実行時間の短縮に繋がっている。例外として、3dmorphsnop のベンチマークでは JIT コンパイラ機構を有効にした場合の方が遅かった。このベンチマークは、計算結果の格納に配列を使っている。本 JIT コンパイル機構は、配列の操作に対して十分な高速化を行なっていないのに加えて、組み込み関数の呼出しの際の値のボクシング及び、返り値のアンボクシングに非常に時間を要している。インタプリタではボクシングした値のみを用いているので、特にこのような手間が発生しない。この問題については、各組み込み関数について LLVM-IR での関数を作成することにより、高速化が可能であると考えている。

表 1　各ベンチマークの実行時間 (秒)

ベンチマーク名	行数	概要
3dmorphsnop	53	特定の行列を計算し作成するプログラム
bisection	27	二分法による方程式の近似解を求めるプログラム
bitopsinbyte	21	ビット演算による簡単な計算プログラム
cordic	58	CORDIC アルゴリズムによる三角関数の値を求めるプログラム
rungekutta	64	ルンゲクッタ法による常微分方程式の近似解を求めるプログラム
simpson	45	シンプソン法による積分の近似解を求めるプログラム
trai	8	呼出し回数の多い再帰関数を実行するプログラム

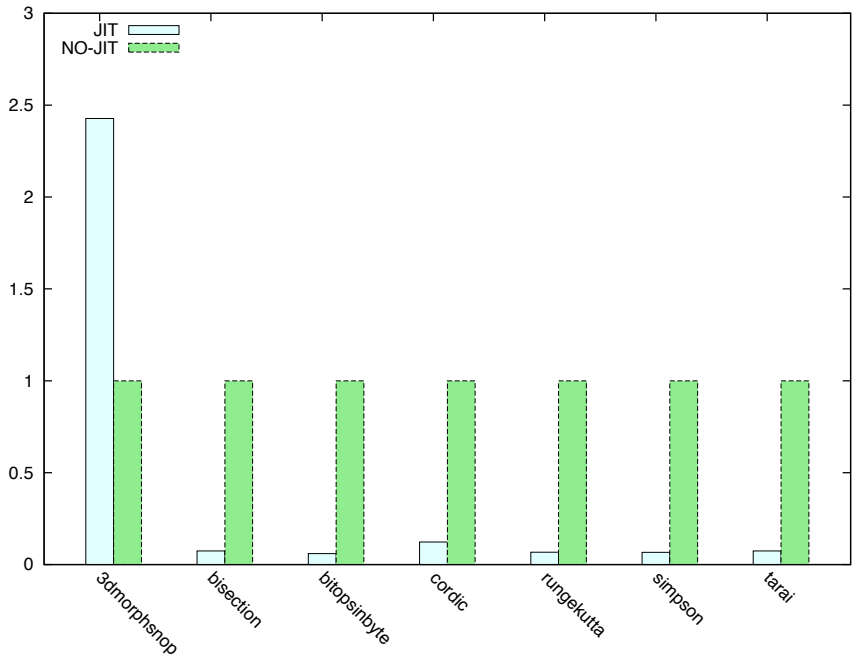

図 26　長期的実行における実行速度比

6.2　他処理系との比較

本研究で作成した JIT コンパイル機構によるコードの実行速度を他の JavaScript 処理系と比較した。本処理系の使用用途は、サーバサイド向けであり、長期的な実行を視野に入れているので、プログラムの実行時間が十分に現れるように、実行時間が 500ms 程度になるようにベンチマークを改変した上で、V8 と SFX における実行時間を比較した。

結果を図 27 に示す。この図は、処理系に対する V8 と SFX の実行速度比を示している。rungekkuta は、既存処理系に比べて高速に動作した。このベンチマークは JIT コンパイル機構の対象となった関数が十分に長く数値計算のみで構成されているため、関数全体に渡る最適化が有効に働いたためである。また、bitopsinbyte も高速に動作した。このベンチマークはマシンコードに変換したコードは短いが、特定の命令における LLVM による最適化が有効に働いたためである。また、bisection、simpson、tarai では、既存処理系と比べてほぼ同程度の速度で動作した。bisection、simpson ベンチマークは、繰り返し実行される部分が単純な制御構造と演算のみで構成されているため、既存処理系の JIT コンパイラ機構で十分な最適化が可能であり、既存処理系と比較してコンパイル等の時間が差となって現れた形になった。また、tarai は再帰関数であるが、呼び出す関数が実際に本当に自身の関数であるかをチェックする必要がある。図 28 に定義した再帰関数が、グローバル変数の変化によって再帰関数ではなくなる例を示す。このプログラムでは再帰関数 `rec` を定義し、それを変数 `func` に代入している。その時の `func` は再帰関数なのだが、`rec` を書き換えると `func` 内では `rec` を呼出しているので、再帰関数ではなくなってしまう。このため、コンパイル時点で再帰関数である場合でも、グローバル変数をロードするコードを挟む必要がある。tarai では、そのロードに時間を要している。これについては

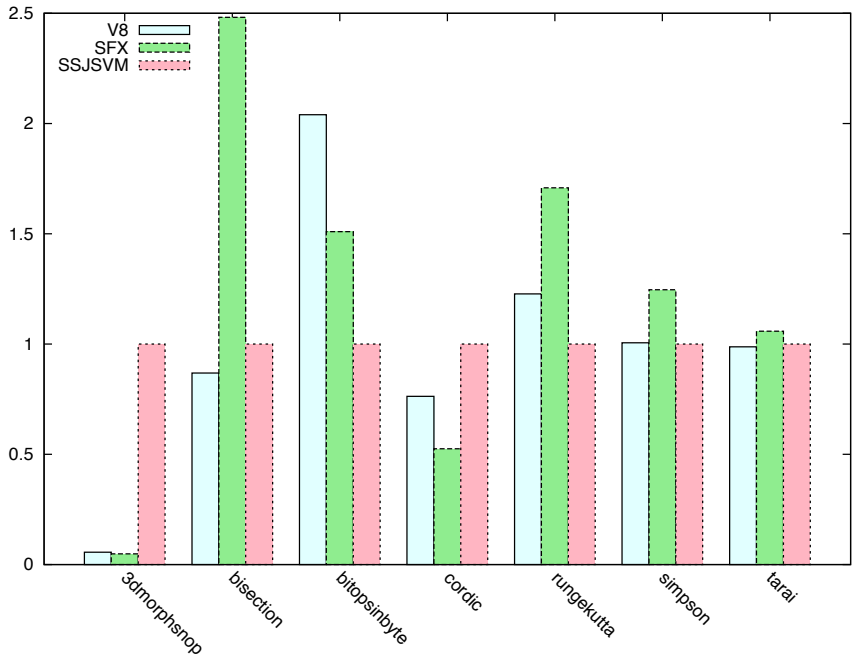

図 27　長期的実行における実行速度比

```
1  // 再帰関数を定義
2  var rec = function(a){
3    if(a == 0) return 0;
4    return rec(a-1) + a;
5  }
6  var func = rec;
7  print(func(10)); // 55
8
9  // グローバル変数が変化
10 rec = Math.abs;
11 print(func(10)); // 19
```

図 28　再帰関数でなくなる例

他の処理系についても同様で、単純なプログラムながら実行に時間を要する。cordic では、プログラム中にグローバル変数のロードの回数が多く、グローバル変数のロードについて十分な最適化をコード変換の際に行っていないために、既存処理系と比べ差がついた。3dmorphsnop では、JIT コンパイル機構を導入していない場合の理由と同じ理由により動作が遅い。

6.3 関数の実行回数と実行速度

本研究で作成した JIT コンパイル機構はマシンコードへのコンパイルに非常に時間を要する。一方で、長期的な実行におけるプログラムの実行時間は、幾つかのベンチマークにおいて、本処理系の方が既存の処理系より短いことが分かる。ここでは、本処理系が最も高速に動作したプログラムである rungekutta について、関数の実行回数と実行時間の関連を見ていく。

結果を図 29 を示す。処理系のプログラムのロードや、グローバル環境のセットアップ等の立ち上げの時間や、JIT コンパイル時間を含めたプログラム実行外の時間が、本処理系は 25ms 程度なのに対し、既存の V8、SFX では対応する時間はおおよそ 15ms 程度である。この 10ms の差は、生成したマシンコードを何度も再利用しするうちに、徐々にその差が埋まっていく。SFX について 50 万回程度、V8 については 150 万回程度そのコードが再利用された時に、累計の実行時間が本研究で作成した処理系とおおよそ等しくなる。それ以上の回数コードが再利用された場合は、本研究で作成した処理系の方が実行時間は短い。

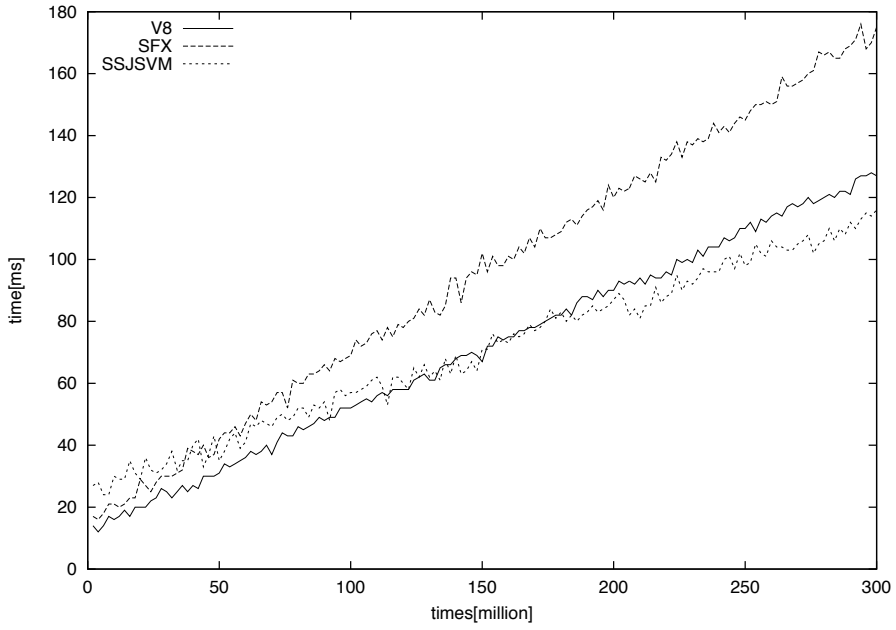

図 29　再利用回数と実効速度の関連

7　関連研究

TESSA [1] は、JIT コンパイラ機構に LLVM を採用した ActionScript 処理系である。ActionScript は JavaScript と同様に ECMAScript 準拠の言語であるが、JavaScript とは異なり、プログラムに型のアノテーションが付加されている。そのため TESSA における ActionScript の JIT コンパイラ機構は、実行時の情報の一部だけを、形注釈と型推論機構によって最適化したマシンコードの生成に必要な型情報の補完に用いている。TEESA では、LLVM-IR の生成、その最適化及びマシンコードへの変換は多大なコストであると認識しており、LLVM におけるマシンコード生成の最適化レベルは LESS に設定されている。

SFX [17] は Apple の製作した JavaScript 処理系である。その特徴として高速なインタプリタ実行部に加え、二段構成の JIT コンパイラ機構がある。ホットスポット検出の閾値に合わせて、Baseline JIT と呼ばれる簡易的な JIT コンパイラ機構及び、DFG JIT と呼ばれるより高度な最適化を施す JIT コンパイラ機構が動作して、高速なマシンコードを生成して実行する。

V8 [12] は Google の製作した JavaScript 処理系である。その特徴として V8 にはインタプリタ実行の機構が存在しない。プログラムは全てマシンコードで実行される。V8 では JavaScript プログラムは一旦簡単なマシンコードに置き換えられ、そこに含まれるプロファイラによって関数等の実行回数の閾値を超えると、再度その部分について JIT コンパイラ機構により高速なコードに置き換えられる。

V8 では、オブジェクトに関する最適化手法に、オブジェクト指向言語 SELF における研究成果 [8] を用いている。その手法では、実行時に同じプロパティを持つオブジェクトをグループ化する。グループ化はオブジェクトの持つプロパティの種類に基づいて行う。すなわち、同じ初期化関数を用いて作成したオブジェクトのように。同じ名前のプロパティを持つオブジェクトは、同じグループに属すようにする。この時、同じグループのオブジェクトのプロパティは、オブジェクト内の同じ位置に格納する。さらに、グルーピングされたそのクラスのオブジェクトに対し、クラスベースのオブジェクト指向言語におけるインラインキャッシュの手法を適応している。この手法では、対象となるオブジェクトに変更が生じた場合でも、プロパティの構成が等しい場合はインラインキャッシングの対象となり、広範囲に及ぶ高速化を可能にしている。

また、別のアプローチから JavaScript を高速実行する試みもある。asm.js [18] は JavaScript に対してより高度な最適化を施せるように記法を策定した JavaScript のサブセットである。asm.js 準拠の JavaScript コードに対しては、実行前に高速なマシンコードにコンパイルする。高速化されたコードを実行する前に、引数が適切な型で与えられているかを確認する。asm.js では変数の型を一意に定めるために、JavaScript プログラムの変数に JavaScript 互換を保つような型についてのアノテーションを付加する必要がある。この型のアノテーションは JavaScript の型よりも細分化されている。例えば、JavaScript における Number 型は実数を扱う型であるが、asm.js では整数と浮動小数点数に区分される。実引数に対して整数値型であることを示す場合、実引数 a に対して、$a = a|0$ のように、0 と or をする一見無意味なコードを挿入する。同様に浮動小数点数であることを示す場合、変数 a に対して、$a = +a$ のように、単項演算子の + を付加したコードを挿入する。このように型のアノテーションをコードの随所に挟んだ場合、高

速化したコードを実行前に生成することができる。

図 30 に asm.js 準拠の JavaScript コードを変換した例を示す。また、asm.js の用途として、C、C++ によって書かれたコードを JavaScript で高速に実行する事が挙げられる。Clang によって、C コードを LLVM-IR コードに変換して、emscripten [19] により、asm.js 準拠の JavaScript に変換する。それにより、C コードの型情報が JavaScript コードに反映され、高速な実行が可能になる。

```
1  // JavaScript
2  function f(a, b){
3    return a + b;
4  }
5
6  // asm.js
7  function f(a, b){
8    "use asm";
9    a = a | 0;
10   b = b | 0;
11   return (a + b) | 0;
12 }
```

図 30　asm.js への変換

8 おわりに

本論文は、サーバサイド向けの JavaScript 処理系における、JIT コンパイル機構の提案と実装を行いその評価を行った。実装にはコンパイラ基盤の LLVM を用いて、バイトコードを LLVM-IR に変換した後、様々な最適化を施し高速化したコードの生成を試みた。その結果、数値計算をはじめとした幾つかのベンチマークにおいて、長期間の運用を前提とした場合、本研究における JIT コンパイル機構が有効であることが確認できた。

LLVM におけて十分に最適化するために、JavaScript に依存する最適化を行うことが必要であることがわかった。変数の型など言語の動的な情報を、実行時情報を元にした型推論機構を用いて、決定することにより、簡潔な LLVM-IR のコードを生成でき、高速化に繋がった。特に対象の型のアンボクシング化によって、変数の型を特殊化することによって、生成される LLVM-IR の処理の見通しがよくなり、LLVM による各種最適化が有効に作用した。

特に、局所変数のみを使用し、複雑な演算を行うようなユーザ定義関数については、本研究の JIT コンパイル機構により、高速なマシンコードの生成が可能であることがわかった。上記の型特殊化の最適化手法に加え、関数全体に LLVM による不要コードの除去や、高速な命令への置き換えの最適化が作用し、無駄のない高速なマシンコードに変換できた。このように、プログラム全体にわたり、LLVM における最適化が有効になった場合は、特に有効な結果が得られた。また、関数呼出しにおいても、関数インライニングにより、呼出し先の関数も含めた処理を最適化することができ、更なる高速化を実現できた。

一方で、オブジェクトについて本研究ではインラインキャッシングの最適化は施しているが、一部のオブジェクトを多用するベンチマークにおいては、一部処理に時間を要しており改善の余地がみられる。本研究において LLVM におけるロード/ストア命令除去の最適化が局所変数にしか適応されないため、オブジェクトのプロパティ、及びグローバル変数を多様するプログラムに対して、多くのロード/ストア命令が挿入されており、処理にオーバーヘッドを伴っているという問題がある。

今後、オブジェクトに関連する幾つかの最適化手法を実装すると共に、現状の JIT コンパイラ機構が未対応な構文であるものについて実装を進める。

謝辞

本研究を行うにあたり、終始変わらぬ御指導を賜わりました岩崎英哉教授、鵜川始陽助教、中野圭介助教に深く感謝致します。また、本研究に対して多大な御助言をいただきました、岩崎研究室の皆様に心から御礼申し上げます。

参考文献

[1] M. Chang, E. Smith, A. Chaudhuri, A. Gal, et al. The impact of optional type information on jit compilation of dynamically typed languages. *Proceedings of the 7th symposium on Dynamic languages* (DLS '11), pp. 13–24, 2011.

[2] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *Proceedings of the international symposium on Code generation and optimization* (CGO '04), pp. 7–, 2004.

[3] M. Anton and D. Gregg. Retargeting JIT compilers by using C-compiler generated executable code. *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques* (PACT '04), pp. 41–50, 2004.

[4] J. R. Bell. Threaded code. *Communications of the ACM*, 16(6), pp. 370—372, 1973.

[5] M. Berndl, B. Vitale, M. Zaleskik and A. D. Brown. Context Threading: A Flexible and Efficient Dispatch Technique for Virtual Machine Interpreters
Proceedings of the international symposium on Code generation and optimization (CGO '05), pp. 15–26, 2005.

[6] S. Brunthaler. Efficient Interpretation using Quickening. *Proceedings of the 6th Symposium on Dynamic languages* (DLS '10), pp. 1–14, 2010.

[7] M. R. Jantz and P. A. Kulkarni. Performance Potential of Optimization Phase Selection During Dynamic JIT Compilation. *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (VEE '13), pp. 131–142, 2013.

[8] C. Chambers, D. Ungar and E. Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes *Conference proceedings on Object-oriented programming systems, languages and applications* (OOPSLA '89), pp. 49–70, 1989.

[9] C. Click and K. D. Cooper. Combining analyses, combining optimizations *ACM Transactions on Programming Languages and Systems* (TOPLAS), pp. 181–196 ,1995.

[10] L. P. Deutsch, A. M. Schiman. Ecient implementation of the Smalltalk-80 system. *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages* (POPL '84), pp. 297—-302, 1984.

[11] 高田 祥, JavaScript 仮想機械における Quickening の効果, PPL2013, 2013.

[12] V8: Google's open source JavaScript engine
at `https://code.google.com/p/v8/` Retrieved January 2014.

[13] Computer Language Benchmarks Game.
at `http://shootout.alioth.debian.org/` Retrieved January 2014.

[14] TraceMonkey : an overview of TraceMonkey.
at `http://hacks.mozilla.org/2009/07/tracemonkey-overview/` Retrieved January

2014.
[15] Clang: C language family frontend for LLVM
at http://clang.llvm.org/ Retrieved January 2014.
[16] JavaScriptCore: Built-in JavaScript engine for WebKit
at http://trac.webkit.org/wiki/JavaScriptCore Retrieved January 2014.
[17] Introducing SquirrelFish Extreme
at https://www.webkit.org/blog/214/introducing-squirrelfish-extreme/ Retrieved January 2014.
[18] asm.js: An extraordinarily optimizable, low-level subset of JavaScript
at http://asmjs.org/ Retrieved January 2014.
[19] Alon Zakai, Emscripten: An LLVM-to-JavaScript Compiler
at http://www.emscripten.org Retrieved January 2014.
[20] Node.js : a platform for scalable network applications
at http://nodejs.org/ Retrieved January 2014.
[21] ecmascriptthe language of the web
at http://www.ecmascript.org/ Retrieved January 2014.