Vol.2019-ARC-240 No.32
Vol.2019-SLDM-191 No.32
Vol.2019-EMB-53 No.32
2019/2/28

動的スクリプト言語の高効率実行を目的としたプロセッサアーキテクチャの拡張

眞下 達[1,a)]　塩谷 亮太[2,b)]　井上 弘士[3,c)]

概要：動的スクリプト言語は幅広い分野で利用されているものの，その実行時オーバーヘッドが大きな課題となっている．動的スクリプト言語を処理する仮想マシンでは一般に，実行時にさまざまな要素を動的に解決する必要がある．特に，仮想マシン上のオペランドの処理では多くのメモリ・アクセスを必要とし，それが実行性能を下げる大きな要因となっている．このオーバーヘッドを削減するために我々は OFAR (Operand Fetching And Remapping) と呼ぶ手法を提案する．OFAR は，(1) オペランド値の浮動小数点物理レジスタ (FPPR: Floating-Point Physical Register) へのマップと，(2) オペランド番号のフロントエンドによるフェッチの，2つから成る．一般に，仮想マシンの大分部は整数命令によって実装されており，FPPR の大部分は使用されていない．これを利用し，通常はメモリ上に置かれる仮想的なオペランドの値を FPPR にマップする．これにより，オペランド・アクセスに伴うメモリ・アクセスの多くを省略することができる．また一般に，仮想命令のオペランド番号は基本的には不変であり，命令コードと同様に演算の結果により書き換わることがない．これを利用し，通常はバックエンドで読み出されるオペランド番号を，命令フェッチと同じようにしてフロントエンドで読み出す．これにより，オペランド番号をロードするための命令が省略されることに加え，早期にオペランド番号が得られることによりレイテンシを削減する．

1. はじめに

動的スクリプト言語は，その高い生産性や移植性のために現在では広く用いられている．たとえば，JavaScript はウェブ・アプリケーションにおいて事実上の標準となっており，モバイル・アプリケーションからデータ・センターにおける大規模解析ソフトウェアに至るまで幅広く利用されている．このような動的スクリプト言語は一般に，High-Level Language Virtual Machine (HLL VM) と呼ばれる仮想マシン上で実行される．この HLL VM は，典型的には，仮想的な命令を動的にコンパイルして実行する Just-In-Time (JIT) コンパイラか，命令をそのまま解釈して実行するインタプリタ，あるいはその組み合わせによって構成されるが，近年では特にインタプリタの性能が重要な課題となりつつある．一般に JIT コンパイラは実行頻度の高い**ホット・ファンクション**が実行時間の大半を占める場合に大きな性能向上を得ることができる．しかし，近年の大型化したワークロードでは実行頻度の低い**コールド・ファンクション**が実行時間の多くを占めることが増えつつあり，そのようなプログラムでは JIT コンパイラはうまく働かない事が多い [15]．これに対しインタプリタは性能解析やコンパイルを行う必要がなく，小さいメモリ・フットプリントと短いスタートアップ時間で実行することができ，コールド・ファンクションを効率的に実行できる．このため，近年の代表的な HLL VM では，JIT コンパイラとインタプリタのハイブリッド構成をとることや [2,7,11,12]，インタプリタのみの構成をとることが増えている [8,14]．

HLL VM のインタプリタでは多くの場合，高級言語で書かれたソース・コードは 一旦 **VM 命令**と呼ぶ中間表現に変換された後に実行される．この VM 命令のオペランドを **VM オペランド**と呼ぶ．この VM オペランドの実現方法の違いにより，インタプリタは**スタック・ベース方式**と**レジスタ・ベース方式**に分類される．スタック・ベース方式では VM オペランドはメモリ上のスタック内に格納され，演算対象となる VM オペランドはスタックの先頭が暗黙的に指定される．一方，レジスタ・ベース方式では VM オペランドはメモリ上のテーブル内に格納され，VM 命令からは演算対象の VM オペランドをテーブルのインデックスによって明示的に指定する．このインデック

[1] 九州大学大学院システム情報科学府
〒 819–0395 福岡県福岡市西区元岡 744
[2] 東京大学大学院情報理工学系研究科
〒 113–8656 東京都文京区本郷 7–3–1
[3] 九州大学大学院システム情報科学研究院
〒 819–0395 福岡県福岡市西区元岡 744
[a)] susumu.mashimo@cpc.ait.kyushu-u.ac.jp
[b)] shioya@ci.i.u-tokyo.ac.jp
[c)] inoue@ait.kyushu-u.ac.jp

Vol.2019-ARC-240 No.32
Vol.2019-SLDM-191 No.32
Vol.2019-EMB-53 No.32
2019/2/28

スを **VM レジスタ ID** と呼び，そのテーブルを **VM レジスタ・セット**と呼ぶ．近年の動的スクリプト言語では，レジスタ・ベースのインタプリタを採用することが増えている [2,7,8,11,13]．これは，スタック・ベース方式ではスタックへのプッシュやポップ操作がボトルネックとなり，レジスタ・ベース方式と比較して性能が低くことが多いためである [17]．そのため，本研究ではレジスタ・ベースのインタプリタに着目する．以下ではインタプリタと言う場合はレジスタ・ベースのインタプリタを指す．

インタプリタの性能ボトルネックの代表的なものとして，VM オペランドのアクセスによるものがある．一般に VM オペランドのアクセスでは，(1) VM レジスタ ID を読み出すロード命令と，(2) VM レジスタ・セットから値を読み出す/書き込むためのロード命令/ストア命令の，2 つのメモリ命令を必要とする．以下では前者を **VM レジスタ ID ロード**，後者を **VM レジスタ値ロード/ストア**と呼ぶ．この 2 つの命令は頻繁に実行されるため，実行性能を下げる大きな要因となっている．

これに対し我々はオペランド・アクセスによるオーバーヘッドを削減する **OFAR (Operand Fetching And Remapping)** を提案する．OFAR は，VM オペランド値の浮動小数点 物理レジスタ (FPPR: Floating-Point Physical Register File) へのマップと，VM オペランド番号のフロントエンドによるフェッチの 2 つから成る．図 1 に OFAR の概要を示す．本研究による貢献は次の通りである．

- **VM オペランド・アクセスの解析**：我々は VM オペランド・アクセスの性能への影響を解析した．最新のレジスタ・ベースの JavaScript インタプリタ [2] を CPI (Cycle Per Instruction) スタックに基づいて解析し，VM レジスタ ID ロードと VM レジスタ値ロード/ストアが最大で実行時間全体の 36% を占めることを明らかにした．
- **VM オペランドの FPPR へのマップ**：図 1 (1) に示す，VM オペランド値を FPPR にマップする手法を提案する．一般にインタプリタによる実行中は FPPR はほとんど使用されないため，それを VM オペランド値のマップ領域として効果的に利用することができる．具体的には，レジスタ・リネーム機構の拡張により，VM レジスタ値ロード/ストアをレジスタ間ムーブ命令に変換する．
- **VM オペランド ID のフロントエンドでのフェッチ**：図 1 (2) に示す，VM レジスタ ID をフロントエンドでフェッチする手法を提案する．一般に，仮想マシンでは仮想的なオペランド番号はバックエンドでデータとしてロードされる．しかし，オペランド番号は命令コードなどと同様に基本的には不変であり，演算の結果書き換わることがない．これを利用し，提案手法では命令フェッチのようにフロントエンドでオペランド番号を読み出す．これにより，オペランド番号をロードするための命令が省略されることに加え，早期にオペランド番号が得られることによりレイテンシを削減する．
- **実装と評価**：提案手法をレジスタ・ベースの JavaScript インタプリタの上に実装し，性能と消費電力を評価した．評価の結果，従来のインタプリタと比較して平均で 10.9% の性能向上と，14.7% の EDP (Energy-Delay Product) の改善が得られることを示した．

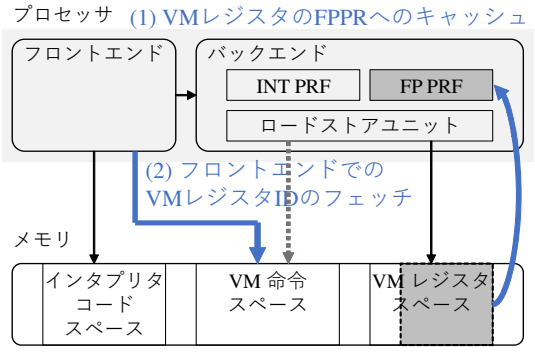

図 1: OFAR の概要．

2. 背景と動機

2.1 インタプリタの概要

2.1.1 構成

インタプリタは一般に，図 2 に示すインタプリタ・コード空間，VM 命令空間，VM レジスタ空間，VM ヒープ空間の 4 つのメモリ空間を持つ．このうち，インタプリタ・コード空間はインタプリタを構成するホスト命令を，VM 命令空間は現在実行中のアプリケーションから変換された VM 命令列を保持する．この VM 命令空間への VM 命令のロードはアプリケーションの実行開始時に行う．残りの 2 つのメモリ空間には，実行時に動的に生成されるデータが格納される．

インタプリタはハードウェアのプロセッサと同様に

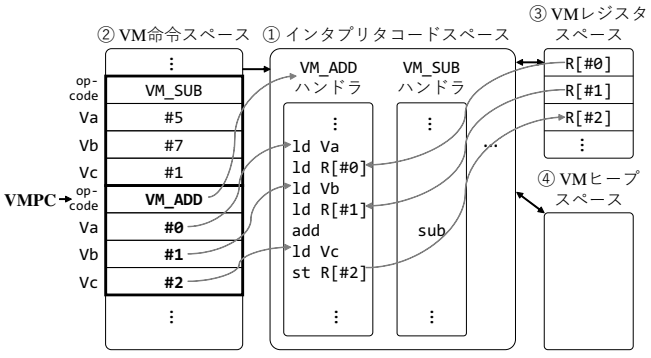

図 2: レジスタ・ベースのインタプリタを構成する 4 つのメモリ空間．

Vol.2019-ARC-240 No.32
Vol.2019-SLDM-191 No.32
Vol.2019-EMB-53 No.32
2019/2/28

VM 命令をフェッチしてディスパッチし，実行する．まず VMPC と呼ぶ現在実行している VM 命令を指すポインタにより，VM 命令の種類を表すオペコードのフェッチを行う．たとえば図 2 では，VMPC が指す VM_ADD 命令のオペコードがフェッチされる．次に，フェッチした VM 命令をディスパッチする．ディスパッチとは，フェッチした VM 命令のオペコードに対応したホスト・コードにジャンプすることである．以後，このホスト・コードのことを**VM 命令ハンドラ**と呼ぶ．最後に，VM 命令ハンドラで VM 命令を実行する．

インタプリタは VM 命令ハンドラ内で VM ヒープ空間と VM レジスタ空間にアクセスする．VM ヒープ空間にはオブジェクトなどのデータ構造，VM レジスタ空間には VM 命令のオペランドの値が格納される．典型的には，VM レジスタ空間はいくつかの VM レジスタ・セットを持ち，関数が呼び出されるごとに新しい VM レジスタ・セットを割り当てる．VM 命令は VM レジスタ ID を用いて VM レジスタ・セット内のオペランド値にアクセスする．

2.1.2 動作

本節では，加算を行う VM 命令 (vm_add) の実行を例として，インタプリタの動作について述べる．VM 命令は一般に，複数のエントリから成る．ここでは，各エントリのサイズは 8 バイトである．たとえば図 2 において，vm_add 命令は 4 エントリで構成される．本論文では，VM 命令のフォーマットを次のように仮定する：

- 1 つ目のエントリは VM 命令のオペコードを格納するフィールドである．オペコードの値は具体的には対応する VM 命令ハンドラのアドレスであり，ディスパッチ時にはこのオペコードの値のアドレスにジャンプする．図 2 では，vm_add のための VM 命令ハンドラのアドレスを VM_ADD と表している．
- 残りのエントリは VM レジスタ ID か即値を格納するフィールドである．以下ではこれを**VM オペランド・フィールド**と呼ぶ．VM オペランド・フィールドの数とその中に格納されるデータの種類は VM 命令の種類によって決まる．vm_add では，3 つの VM オペランド・フィールド Va，Vb，Vc は，それぞれ VM レジスタ ID である #0，#1，#2 を格納する．

図 2 では，VM 命令 vm_sub と vm_add を前述のフォーマットで格納している．vm_add のオペコードと 3 つの VM オペランド・フィールドのメモリ・アドレスは，VMPC がその先頭を指している場合，それぞれ VMPC+0，VMPC+8，VMPC+16，VMPC+24 となる．

図 3 (a) は，VM 命令 vm_add R[#0],R[#1],R[#2] をフェッチ，ディスパッチ，実行するホスト命令の疑似アセンブリ表現である．R[#i] は，VM レジスタ・セットにおける i 番目の VM レジスタを表す．つまり，この VM 命令は 2 つの VM レジスタ R[#0] と R[#1] の値を加算し，

```
1  NEXT_VMPC:
2    add    r0+32→ r0          # VMPC++
3  FETCH:
4    ld     [r0+0]→ r2         # load opcode
5  DISPATCH:
6    jmp    r2                 # go to VM_ADD
7  VM_ADD:
8    ld     [r0+8]→ r3         # load Va: #0
9    ld     [r1+r3]→ r4        # load R[#0]
10   ld     [r0+16]→ r5        # load Vb: #1
11   ld     [r1+r5]→ r6        # load R[#1]
12   add    r4+r6→ r7          # R[#0]+R[#1]
13   ld     [r0+24]→ r8        # load Vc: #2
14   st     r7→ [r1+r8]        # store R[#2]
```

(a) オリジナルのコード

```
1  NEXT_VMPC:
2    vmpc.inc r_vmpc+32→ r_vmpc # VMPC++
3  FETCH:
4    ld     [r_vmpc+0]→ r2     # load opcode
5  DISPATCH:
6    jmp r2                    # go to VM_ADD
7  VM_ADD:
8    ld.vm  [Va]→ r3           # load R[Va:#0]
9    ld.vm  [Vb]→ r4           # load R[Vb:#1]
10   add    r3+r4→ r5          # R[#0]+R[#1]
11   st.vm  r5→ [Vc]           # store R[Vc:#2]
```

(b) 提案手法を実装したコード

図 3: VM 命令 vm_add R[#0],R[#1],R[#2] をフェッチ，ディスパッチ，実行する疑似アセンブリ・コード．r0 が現在の VMPC を，r1 が VM レジスタ・セットのベース・アドレスをそれぞれ保持する．

R[#2] にその結果を書き込む．ここで，r0-8 はホスト・プロセッサの論理レジスタを表し，r0 が現在の VMPC を，r1 が VM レジスタ・セットのベース・アドレスをそれぞれ保持する．

vm_add は図 3(a) で次のように処理される．なお，説明の簡単化のため，ここでは 1 つ前の VM 命令 (vm_sub) の最後の処理から説明を始める．(0) 2 行目のホストの add 命令が VMPC に 32 を加算し，前の VM 命令 (vm_sub) から次の VM 命令 (vm_add) に移る．32 が加算されるのは 1 つ前の VM 命令が 4 つの 8 バイトエントリを持つためである．(1) 次に，4 行目で VM 命令のオペコードをフェッチし，6 行目で対応する VM 命令ハンドラにディスパッチする．(2) VM 命令ハンドラ内 (8-14 行目) の複数のホスト命令によって，vm_add が実行される．詳細は次の節で説明する．

2.2 VM オペランド・アクセス

2.2.1 インタプリタの VM オペランド・アクセス手順

前述した加算を行う VM 命令 (vm_add

Vol.2019-ARC-240 No.32
Vol.2019-SLDM-191 No.32
Vol.2019-EMB-53 No.32
2019/2/28

R[#0],R[#1],R[#2]) を例に，インタプリタがどのようにオペランドへアクセスするかを説明する．VM オペランドは次の 2 つのホスト命令を用いてアクセスされる．

(1) **VM レジスタ ID ロード**は VM オペランドの VM レジスタ ID を VM 命令空間から読み出す命令である．

(2) **VM レジスタ値ロード/ストア**は，読み出された VM レジスタ ID を用いて VM レジスタ・セット内にある VM レジスタ値を読み出す/書き込む命令である．

図 3(a) の 8 行目から 14 行目に，vm_add の VM 命令ハンドラの疑似アセンブリ・コードを示す．まず，VM オペランドの R[#0] にアクセスするために，8 行目の VM レジスタ ID ロードが [VMPC+8] にある #0 を r3 にロードする．次に，9 行目の VM レジスタ値ロードが VM レジスタの値 R[#0] をロードする．ここで，r1 が VM レジスタ・セットの先頭アドレスを保持しているため，r1 + r3 は対象となる VM レジスタのメモリ・アドレスである．2 つ目のソース VM オペランドの R[#1] と デスティネーション VM オペランドの R[#2] は，10-11 行と 13-14 行のコードを用いてそれぞれ同様の手順でアクセスされる．ただし，R[#2] にアクセスする 2 つ目のホスト命令はここではロード命令ではなくストア命令 (VM レジスタ値ストア) である．結果として，1 つの加算 VM 命令を実行するために，6 つのホスト・メモリ命令 (5 つのロードと 1 つのストア) が必要となる．

2.2.2 VM オペランド・アクセスのコスト

前述したように VM オペランドのアクセスには多くのメモリ・アクセスが必要なため，これがインタプリタの性能を大きく低下させる．VM オペランド・アクセスの性能への影響を解析するため，我々は VM オペランド・アクセスが CPI (Cycle Per Instruction) スタックに占める割合を計測した．CPI スタック上で各要素が全体に占める割合は，特定の処理が実行時間に与える影響の大きさを示す．評価には Sniper シミュレータ [4] を使用した．図 4 に，Apple の JavaScript VM [2] に搭載されるレジスタ・ベース・インタプリタ LLInt 上で 2 つの JavaScript ベンチマーク・スイート (SunSpider [1] と v8 [6]) を実行した際の結果を示す．同図より，VM オペランド・アクセスは多くのベンチマークで 15%以上の割合を占めており，最大で v8-crypto において 36%を占めることがわかる．

2.2.3 先行研究: スタック・キャッシング

VM オペランド・アクセスのオーバーヘッドを削減する手法として**スタック・キャッシング** [5] がある．これはスタック・ベース方式のためのソフトウェア・ベースの手法であり，VM オペランドの値をホスト・プロセッサ上の論理レジスタにマップする．前述したように，スタック・ベース方式ではオペランドの値はオペランド・スタックの先頭から LIFO に従ってアクセスされる．したがって，基本的にはオペランド・スタックは先頭の数要素しかアクセスされない．このため，頻繁にアクセスされるスタックの先頭部分は数個の論理レジスタにより効果的にマップできる．

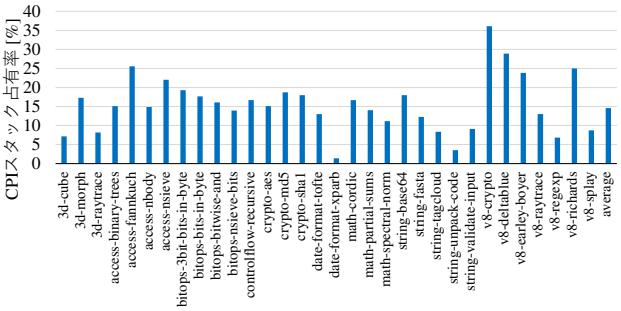

図 4: VM オペランド・アクセスが CPI スタックに占める割合．

しかし，この手法はレジスタ・ベース方式には適用することが難しい [5,16]．これは，スタック先頭の数要素のみがアクセスされるスタック・ベース方式とは異なり，レジスタ・ベース方式では数十個の VM レジスタ内の任意のエントリにアクセスするためである．そのため，レジスタ・ベース方式で論理レジスタにオペランドを効果的にマップする手法は今日まで提案されていない．

我々がレジスタ・ベース方式に着目するのは，オペランドが効果的にマップできなくてもなおレジスタ・ベース方式の方がスタック・ベース方式よりも性能が高いためである．Shi らは，レジスタ・ベース方式はスタック・キャッシングを実装したスタック・ベース方式よりも高い性能を達成することを報告している [17]．このため，我々はレジスタ・ベース方式において効果的なマップを行う新しいハードウェア・ベースの手法を提案する．

3. 提案手法の概要

我々は OFAR (Operand Fetching And Remapping) と呼ぶ VM オペランド・アクセスのオーバーヘッドを削減する新しいハードウェア・ベースの手法を提案する．本節ではこの OFAR の概要について述べる．OFAR 以下の 2 つの手法から成る．1 つ目は **VMRegMap** と呼ぶ手法であり，VM レジスタをアウト・オブ・オーダー (OoO: Out-of-Order) プロセッサ上の使われていない FPPR にマップする．2 つ目は **VMOpFetch** と呼ぶ手法であり，VM レジスタ ID をプロセッサのフロントエンドで早期にロードする．以下ではそれぞれの手法について述べる．

3.1 VMRegMap: FPPR を用いた VM レジスタ値マップ

1 つ目の手法は VM レジスタ値ロード/ストアによるメモリ・アクセスをレジスタ間ムーブに変換する．レジスタ間ムーブはロード/ストアと異なりロード・キューや L1 データ・キャッシュなどの高価な機構にアクセスする必要がなく，実行レイテンシも短いため，VM オペランド・ア

クセスのオーバーヘッドが削減される．

この手法は次の 2 つの知見に基づいている：

(1) **インタプリタ実行中の FPPR の利用率は非常に低い．** VM 命令ハンドラ内のほとんどのホスト命令は制御のための整数命令であり，浮動小数点演算を行う VM 命令を実行している場合でもそれは変わらない．たとえば，前述した LLInt インタプリタ上で浮動小数点のオペランドを用いた vm_add を実行する例では，33 個のホスト命令のうち，浮動小数点演算を行うホスト命令は 3 つのみである．また，LLInt 上で JavaScript のベンチマーク (Speedometer [3] と V8 benchmark suite [6]) を実行した場合，我々の測定では浮動小数点演算のホスト命令が全体に占める割合は 1%以下であった．そのため，我々はインタプリタで利用されていない多くの FPPR を，VM レジスタの値をマップするために用いる．

(2) **インタプリタが実行時に用いる VM レジスタの数は十分に少ない．** VM レジスタの数はアプリケーションによって異なるため，ベンチマークを用いて VM レジスタの数を評価した．その結果，ほとんどの関数は 32 個以下の VM レジスタを用いることが分かった．具体的には，Speedometer では 98%，v8 では 97%の関数が 32 個以下の VM レジスタを使用していた．一方．今日の OoO プロセッサは一般に 100 個以上の FPPR を持つ．そのため，ほとんどの場合は VM レジスタは FPPR にマップ可能である．

3.2 VMOpFetch: フロントエンドにおける VM レジスタ ID のロード

2 つ目の手法では，フロントエンドがホスト命令に対して行っているように VM レジスタ ID を “フェッチ” する．この手法は，VM 命令空間内の VM レジスタ ID はプログラムの実行開始時に格納され，その後基本的には変更されない点に基づいている．一般に，フロントエンドではプログラム・カウンタ (PC: Program Counter) が指すアドレスにある命令のフェッチを行う．これと同様に，本手法はプロセッサに仮想マシンの PC である VMPC を保持するレジスタを追加し，その VMPC を用いてアドレスを計算することで，VM レジスタ ID をフロントエンドでフェッチする．これにより，実行されるホスト命令数が削減されることに加え，VM レジスタ ID のロード・レイテンシを隠蔽することができる．

3.3 アーキテクチャの概要

OFAR は命令セットアーキテクチャ (ISA: Instruction Set Architecture) とマイクロアーキテクチャの拡張で構成される．インタプリタを拡張された命令を用いて変更することで，プロセッサがインタプリタの実行情報を追跡できるようにする．この情報を用いてプロセッサは動的に VM レジスタ を FPPR にマップし，VM レジスタ ID をフロントエンドでフェッチする．

先述した 2 つの手法は協調して動作する．図 5 に OFAR の動作の概要を示す．Figure 5 (a) は，図 3 (a) の vm_add の VM 命令ハンドラ内の疑似アセンブリ・コードを実行する様子を示している．ここでは簡単のために関連しないいくつかの命令は省略している．i1 は，VMPC を更新するための add ホスト命令である．i2 と i3 は，それぞれ VM レジスタ ID ロードと VM レジスタ値ロードである．この 2 つの命令によってロードされた VM レジスタ値を用いて i4 が加算を実行し，その結果が i5 によって書き戻される．i1 から i5 までの命令は直列に依存しており，図中の①で実行が完了する．

これに対し，図 5 (b) は提案する OFAR の実行の様子である．同図は OFAR のために変更された vm_add のハンドラを提案するアーキテクチャが実行する様子である．vmpc.inc と ld.vm，st.vm は，OFAR で新しく追加されたホスト命令である．これらの命令の詳細は次の章で説明する．OFAR の重要なアーキテクチャの機能は，フロントエンド内にある (1) VM レジスタ ID の投機的なロードと，(2) VM レジスタの値がマップされている場所 (FPPR ないしはメモリ) に応じた micro-op の生成である．OFAR の動作の概要は以下の通りである：

(1) **i6**: vmpc.inc は元のコードの i1 に相当し，VMPC を更新する．i1 のような通常の加算命令とは異なるのは，この命令は加算自体に加えて VMPC の値をプロセッサに通知する点である．また，② に示すようにプロセッサは次の VMPC を予測し， i2 のように投機的に VM レジスタ ID をロードする．この投機的なロードは vmpc.inc のフェッチと並行して行われるため，そのレイテンシは隠蔽される．この命令の詳細は後で述べる．

(2) **i7**: ld.vm は元のコードの i3 に相当し，VM レジスタの値をロードする．デコード・ステージにおいて，i6 がフェッチした VM レジスタ ID を用いて ld.vm は対応する VM レジスタ値が FPPR とメモリのどちらにマップされているかを確認する．もし FPPR にマップされていた場合，本命令は，FPPR から対象の論理レジスタへ値をムーブする micro-op にデコードされる．一方，メモリにマップされていた場合，本命令は，メモリ上の VM レジスタ空間から値をロードする micro-op にデコードされる．図の例は，値が FPPR にマップされていた場合を示す．

(3) **i8**: 得られた VM レジスタ値を用いて加算を実行する．

(4) **i9**: ld.vm と同様の処理を行い，実行結果を FPPR に書き戻す．

Vol.2019-ARC-240 No.32
Vol.2019-SLDM-191 No.32
Vol.2019-EMB-53 No.32
2019/2/28

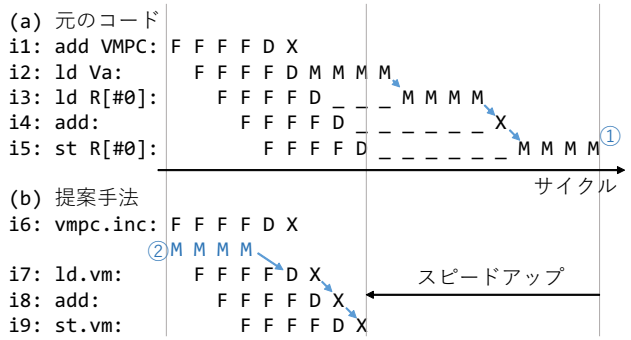

図 5: VM 命令を実行するプロセッサのパイプライン図．F: フェッチ，D: デコード，X: 実行，M: メモリー (L1 データ・キャッシュ) アクセス．L1 データ・キャッシュのアクセス・レイテンシは 4 サイクルとする．

表 1: OFAR の ISA 拡張．r_j は論理レジスタ ID を表している．Vx は VM オペランド・フィールドを表している (例: Va=1st)．

ホスト命令	説明
vmpc.inc r_j or imm	r_{vmpc} に r_j か imm を加算
vmpc.br Vx	r_{vmpc} に Vx を加算
vmpc.ijmp r_j	r_j の値で r_{vmpc} を更新
ld.vm Vx, r_j	R[Vx] を r_j にロード
st.vm r_j, Vx	r_j を R[Vx] にストア
mov.vmbase r_j	r_j の値で r_{base} を更新
st.vmregflush	全 VM レジスタ値をメモリにストア

4. ISA 拡張

提案手法では，次の 2 つの新しい論理レジスタを ISA に追加する．

- r_{vmpc}: 現在実行中のインタプリタの VMPC を保持する．
- r_{base}: 現在の VM レジスタ・セットのベース・アドレスを保持する．

表 1 に，新しく ISA に追加するホスト命令をまとめる．以下では，上記の新しく導入されたレジスタや命令を図 3 (b) の例を用いて説明する．この図は vm_add を実行する元のコード (図 3 (a)) を OFAR のために変更したもので，変更された命令を太字で強調している．

追加された 2 つの新しい論理レジスタは以下で述べる追加された命令によりインタプリタから明示的に更新する．一般に，多くのインタプリタでは VMPC を論理レジスタ上に保持している．たとえば，図 3 (a) では論理レジスタ r0 が VMPC を保持しており，2 行目の add ホスト命令によって更新される．同様に，r_{vmpc} は追加した vmpc.inc (increment VMPC) と vmpc.br (branch)，vmpc.ijmp (indirect jump) を用いて更新する．これらの追加されたホスト命令は既存のインタプリタが行う典型的な VMPC の更新操作にそれぞれ対応する．vmpc.inc は，通常の加算を行うホスト命令のようにふるまう．たとえば，図 3 (a) の 2 行目の add ホスト命令は，図 3 (b) の 2 行目のように vmpc.inc に置き換えられる．vmpc.br も vmpc.inc と同様に add ホスト命令のようにふるまうが，そのオペランドが VM オペランド・フィールドからロードされた値となる点が異なる．なお，これらの新しく追加された命令は VM オペランド・フィールドからロードされた値を VM レジスタ ID として扱うが，vmpc.br だけはディスプレースメントとして扱う．たとえば VM オペランド・フィールド Va 内の値が #32 の時，vmpc.br はこれを用いて VMPC+=R[#32] とするのではなく，VMPC+=32 とする．vmpc.br は常に VMPC を更新するため，条件分岐の VM 命令は vmpc.br と条件分岐を行うホスト命令の組み合わせによって実装される．vmpc.ijmp は，論理レジスタ r_j 内の値で VMPC を更新する．

r_{base} は，現在の VM レジスタ・セットのベース・アドレスを保持する．たとえば LLInt では，ベース・アドレスは各 JavaScript 内の関数呼び出しごとに固有なため，このレジスタは関数が呼び出されるごとに変更される．このレジスタは mov.vmbase によって更新される．

2.2.1 節で述べたように，VM オペランド・アクセスは VM レジスタ ID ロード (図 3 (a) の 8，10，13 行目) と VM レジスタ値ロード/ストア (図 3 (a) の 9，11，14 行目) の 2 つのホスト命令によって実現される．ld.vm/st.vm は，この 2 つのホスト命令を組み合わせた間接ロード/ストア命令のように機能する．つまり，これらの命令はまず VM レジスタ ID をロードし，それを用いて VM レジスタ値をロード/ストアする．Vx は対象とする VM レジスタ ID が入っている VM オペランド・フィールドを指定する即値である．たとえば，図 3 (b) の 8 行目において，Va は 1 番目の VM オペランド・フィールドを指しており，この ld.vm は [r_{base}+[r_{vmpc}+8]] をロードする．

論理的な動作としては ld.vm/st.vm は VM レジスタ ID ロードと VM レジスタ値ロード/ストアを組み合わせたものと等価であり，単純にこれらの命令を追加しただけではオペランド・アクセスは高速化されない．**これに対し，OFAR はインタプリタから通知された情報をあわせて用いることにより，その実効レイテンシをレジスタ間ムーブと同じにまで削減する．**この詳細については次節で述べる．

st.vmregflush は，FPPR にマップされたすべての VM レジスタをメモリに書き戻す命令であり，r_{base} が更新される前に呼び出される．これは，FPPR 内の VM レジスタは r_{base} のベース・アドレスが指す現在の VM レジスタ・セットに紐づけられているためである．

また，プロセッサは VM 命令のフォーマットが OFAR が規定したものに従っていることを想定して動作する．本論文では図 2 の例で用いた次のフォーマットを用いる． 1

番目のエントリは VM 命令のオペコードであり，その後のエントリーは VM オペランド・フィールドであり，VM レジスタ ID ないしは即値オペランドを保持する．

5. マイクロアーキテクチャの拡張

5.1 構成

我々は，OFAR を実現するために従来の OoO プロセッサを拡張する．図 6 に，OFAR のパイプラインの概要を示す．ここでは，説明のために図 3 (b) の 2 行目 (vmpc.inc) と 8 行目 (ld.vm) の命令を用いる．灰色の四角は，OFAR のために追加されたハードウェア・ユニットである．vmst と vmld，vmmov は，ld.vm からデコードされた micro-op を表す．実線の矢印は追加されたハードウェア・ユニットに関連するデータの流れを，点線の矢印は命令か micro-op の制御の流れを，そして角丸の四角は命令ないしは micro-op の処理をそれぞれ表す．

4 節で提案した論理レジスタに加え，OFAR では 2 つのハードウェアユニットを追加する．これらをそれぞれ *VM register ID fetch unit* (IDFU) と *VM Remap Table* (VMRT) と呼び，以下で詳しく説明する．

5.1.1 IDFU

IDFU は VMOpFetch の中心となるユニットであり，vmpc.inc ないしは vmpc.br がフェッチされた時に次の VM 命令の VM レジスタ ID をフェッチする．このユニットは最大で n 個の VM レジスタ ID を保持するために n 個の **IDREGx** と呼ぶレジスタを持つ (x はアルファベットで表したインデックスである) ．これらのレジスタは valid bit フィールドと値フィールドの 2 つのフィールドを持つ．1 つの VM 命令が持つ VM レジスタ ID をすべて IDREG にフェッチするため，n は 1 つの VM 命令が持てる VM オペランドの最大数を決める．そのため，n はインタプリタ実装の VM オペランドの最大数を十分サポートできるだけ大きい必要がある．一般に，VM 命令はハードウェア・プロセッサのホスト命令と同様に 3 つ以下の VM オペランドを持ため，IDREG は 3 つあれば充分である．

5.1.2 VMRT

VMRT は VMRegMap の中心となるユニットであり，VM レジスタ ID と それに対応する VM レジスタ値を持つ FPPR の間の対応を管理するテーブルである．これは，従来の OoO プロセッサが持つレジスタ・リネーム・テーブルと同様の機構であり，さらに valid bit フィールドとタグ・フィールドを追加している．VMRT と従来のレジスタ・リネーム・テーブルの違いは主に 2 つある．1 つ目は，レジスタ・リネーム・テーブルが論理レジスタ番号によってアクセスされ，それを FPPR の物理レジスタ番号にリネームするのに対し，VMRT は VM レジスタ ID によってアクセスされる点である．2 つ目は，VMRT のエントリ数は VM レジスタの数より大きい，ないしは小さい場合がある点である．これは，前述したように VM レジスタ・セットのに含まれるレジスタ数は関数ごとに異なるためである．このため，VMRT ではアクセスされた VM レジスタ ID がミスする場合がある．一方，レジスタ・リネーム・テーブルではエントリ数が常に ISA が定めた論理レジスタ数に等しいため，常にヒットする．そのため，VMRT はアクセス時に valid bit フィールドとタグ・フィールドを用いてヒット・ミス判定を行う．

本節の残りでは，VMRegMap と VMOpFetch を実装するための様々な仕組みについて，詳細に説明する．

5.2 VMOpFetch と VMPC の更新 (vmpc.inc)

本節では VMOpFetch に関連する VMPC の更新について説明する．ここでは図 3 (b) に示した add VM 命令 (vm_add R[#0],R[#1],R[#2]) のソース・コードを用いる．図 3 (b) の 2 行目の通り，OFAR を実装したインタプリタは vmpc.inc を実行することで r_{vmpc} 内の VMPC を更新する．プロセッサは，vmpc.inc をフェッチすると IDFU を用いて投機的に次の VM 命令の VM レジスタ ID のフェッチを開始する．以下，vmpc.inc (図 3(b) の 2 行目) の処理を，パイプライン・ステージの順で説明する．

5.2.1 次の VMPC の予測

図 6 ①に示すように，プロセッサは vmpc.inc のフェッチ時に BTB を用いてそれが vmpc.inc であるかどうかを判定する．このために，BTB の各エントリには通常の分岐命令と vmpc.inc を区別する 1 ビットの識別フィールドを追加する．また vmpc.inc である場合，BTB のエントリには，現在の VMPC とその次の VMPC の予測オフセットが格納される．この予測オフセットは現在の VMPC が指す VM 命令の大きさを表す．BTB エントリが vmpc.inc によって更新される手順は，後で説明する．最後に，このオフセット予測値を現在の VMPC と足し合わせることで次の VMPC を予測する．

5.2.2 VM レジスタ ID のフェッチ

図 6 ② に示すように，IDFU は予測された次の VMPC を受け取ると，それを用いて VM レジスタ ID のフェッチを開始する．まず，すべての IDREG の valid bit をリセットする．次に，予測された VMPC を基に VM レジスタ ID が格納されているフィールドのアドレスを VMPC+8，VMPC+16，というように計算する．これは，4 節で述べたように IDFU では VM 命令が OFAR が規定したフォーマット（各フィールドは 8 バイト）で VM 命令空間に格納されていることを前提に動作するためである．

IDFU はフェッチすべき VM レジスタ ID の個数を決定することができない．なぜなら，予測された VMPC に対応する VM 命令の VM オペランド数がフェッチ開始時点ではわからないためである．そこで，本論文では固定数の VM レジスタ ID を常にフェッチする単純な方法を採用

Vol.2019-ARC-240 No.32
Vol.2019-SLDM-191 No.32
Vol.2019-EMB-53 No.32
2019/2/28

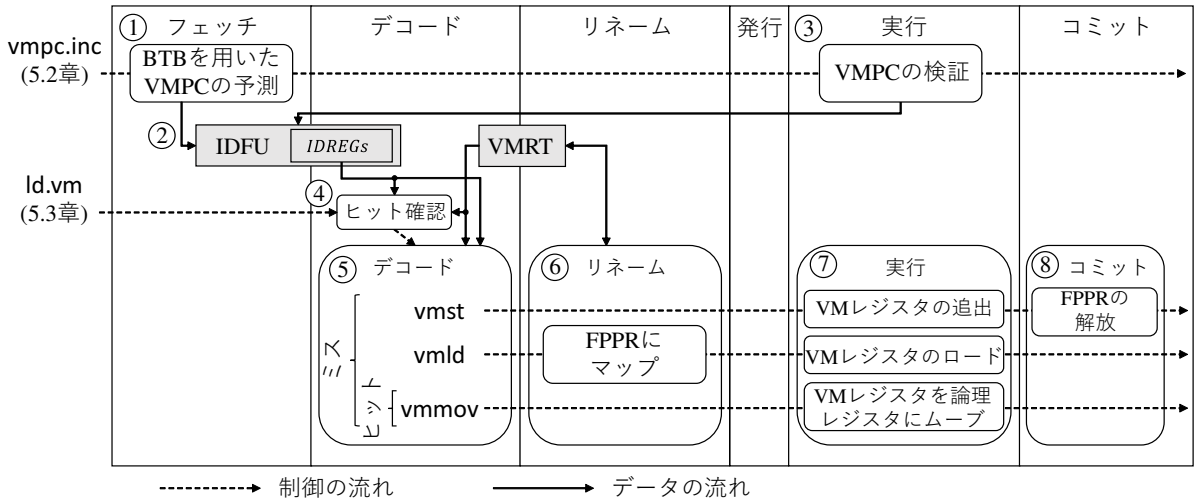

図 6: OFAR のパイプライン概要.

した．たとえば LLInt では，ほとんどの VM 命令は 3 つ以下の VM オペランドを持つため，3 つの VM レジスタ ID をフェッチする．この手法では，VM 命令が 2 つ以下の VM オペランドしか持たない場合，無駄なメモリ・アクセスを行うことで性能を低下させる可能性がある．しかし，評価よりこれによる性能低下は無視できる程度である．

IDFU はバックエンドにあるロード・ストア・ユニット (LSU: Load Store Unit) を介して L1 データ・キャッシュ (L1DC) から VM レジスタ ID をフェッチする．既存の LSU を用いるため，専用のロード・ユニットやトランスレーション・ルックアサイド・バッファ (TLB: Translation Lookahead Buffer) を追加する必要はない．LSU が IDFU とバックエンドの両方から同時にメモリ・リクエストを受け取った場合，IDFU からのリクエストを優先する．最後に，[VMPC+8i] からフェッチした VM レジスタ ID を IDREG に格納し，その valid bit をセットする．

5.2.3 予測した VMPC の検証

図 6 ③ に示す通り，フェッチ・ステージで予測した VMPC は vmpc.inc がバックエンドの実行ステージで実行された際に検証される．vmpc.inc は，実行ステージで実際の次の VMPC を計算し，r_{vmpc} に格納する．この時，計算結果の VMPC を予測 VMPC と比較し，予測ミスを検出する．これは，従来のプロセッサが通常の分岐命令の分岐予測結果を検証する方法と同様である．

VMPC の予測ミスは，IDFU が誤った VM レジスタ ID をフェッチし，後続のそれを用いる命令 (ld.vm と st.vm) が誤った実行結果を返す原因となる．そのため，プロセッサは VMPC の予測ミスを検出すると，次の 3 つの操作によって正しい動作に復帰する．1 つ目は，予測ミスした vmpc.inc の後続の命令をすべてフラッシュする．この操作は どの命令も誤った VM レジスタ ID を用いて実行されないことを保証するために必要である．2 つ目は，正しい VMPC を用いて IDFU に正しい VM レジスタ ID をフェッチさせる．3 つ目は，予測ミスした vmpc.inc に対応する BTB エントリを正しい VMPC のオフセットで更新する．この操作によって，BTB は同じ vmpc.inc に対する次回以降の予測のために正しいオフセットを学習する．

5.3 VMRegMap と VM レジスタ値ロード (ld.vm)

次に，ld.vm (図 3 (b) の 8 行目) の処理を，パイプライン・ステージの順番に説明する．この命令は，R[#0] の VM レジスタ値を論理レジスタ r3 にロードする．処理の概要を図 6 の下部に示す．

5.3.1 VM レジスタのマップ状態の確認

図 6 ④ に示すように，ld.vm はまず R[#0] が FPPR にマップされているかをデコード・ステージでチェックする．この命令は Va をオペランドに持つため，IDREGa から VM レジスタ ID を受け取る．この命令は，デコードされるとき IDREGa 内にある #0 をインデックスとして VMRT にアクセスする．もし #0 が VMRT 内でヒットした場合，R[#0] が FPPR にマップされていることを意味する．

ここで，もし IDREGa の valid bit がセットされていない場合，IDFU が IDREGa に VM レジスタ ID をフェッチするまで，この命令はデコード・ステージでストールする．このストールは，それ以降のすべての命令の処理を止めるため性能低下を起こす可能性がある．しかし，IDFU は通常十分早期に VM レジスタ ID をフェッチできるため，このストールはほとんど起きない．これは IDFU はフェッチ・ステージで vmpc.inc の命令キャッシュからのフェッチと並行して VM レジスタ ID のフェッチを行うためである．

5.3.2 ld.vm のデコード

図 6 ⑤ に示すように，ld.vm は VMRT のヒット/ミスに応じて異なる micro-op にデコードされる．

ヒットした場合，この命令の対象の VM レジスタ値は FPPR にマップされている．このため，この命令は FPPR から デスティネーションの論理レジスタに値をムーブする vmmov にデコードされる．vmmov はデコード元の命令 (ld.vm) のデスティネーション論理レジスタ (ここでは r3) をデスティネーションにとり，この命令のソース・オペランドの VM レジスタ ID (IDREGa 内の #0) をソースにとる．

ミスした場合，対象の VM レジスタ値はメモリにマップされているため，以下の 3 つの処理を行う．

(1) FPPR にマップされた VM レジスタの中から追い出し対象を選び，メモリに書き戻す．
(2) 読み出し対象の R[#0] を FPPR にマップし，その値をメモリからロードする．
(3) ヒット時と同様に，vmmov によって FPPR から論理レジスタに値をムーブする．

これらの処理のため，ld.vm は図 6 に示すように vmst と vmld，vmmov の 3 つの新しい micro-op にデコードされる

vmst は通常のストア命令を基にしており，上記の ld.vm の 1 つ目の処理である VM レジスタの FPPR からメモリへの追い出しに対応する．VMRT はダイレクト・マップ・キャッシュと同じ方法でアクセスするため，追い出し対象は一意に決定される．すなわち，現在 FPPR にマップされた VM レジスタの内，ミスした VM レジスタと VMRT のインデックスが同じ VM レジスタが追い出し対象に選ばれる．ここでは，追い出し対象の VM レジスタの VM レジスタ ID を *victimid* とする．vmst は 3 つのソース・オペランドを取る．最初の 2 つは r_{base} と *victimid* であり，R[victimid] のメモリ・アドレスを意味する．3 つ目は，現在 R[victimid] にマップされている FPPR の番号である．この FPPR 番号は VMRT に格納されている．

vmld は，通常のロード命令を基にしており，ld.vm の 2 つ目の処理である，R[#0] の値をメモリから FPPR にロードする処理に対応する．この micro-op は，1 つのデスティネーション・オペランドと 2 つのソース・オペランドを取る．デスティネーション・オペランドは，この命令 (ld.vm) によって指定された VM レジスタ ID (ここでは IDREGa 内の #0) となる．2 つのソース・オペランドは，r_{base} とデスティネーション・オペランドと同じ VM レジスタ ID であり，対象の VM レジスタ値のメモリ・アドレス (R[#0]) を意味する．

5.3.3 ld.vm のリネーム

図 6 ⑥ に示すように，デコードされた前述の 3 つの micro-op は特別な手順に従ってリネームされる．

vmst では，ソース・オペランドの r_{base} は通常の論理レジスタと同様にリネームされる．残りのオペランドについては，即値もしくは FPPR 番号であるため，リネームは不要である．

vmld では，ソース・オペランドの r_{base} は vmst と同様にリネームされる．デスティネーション・オペランドの VM レジスタ ID は，VMRT を用いて FPPR 番号にリネームされる．このリネーム操作は，VM レジスタ ID によってインデックスされた VMRT のエントリを，現在使用されていない FPPR の 番号で更新することによって行われる．このリネーム操作は，対象の VM レジスタを FPPR にマップする操作に対応する．

vmmov では，デスティネーション・オペランドの論理レジスタは通常の論理レジスタと同様にリネームされる．ソース・オペランドの VM レジスタ ID は，VMRT から読み出された FPPR 番号 にリネームされる．

5.3.4 ld.vm の実行とコミット

図 6 ⑦/⑧ に示すように，前述したように生成された 3 つの micro-op は通常のロード，ストア，レジスタ間ムーブと同様にして実行される．ただし，vmst のコミットだけは追加の手順が必要である．vmst は R[victimid] をメモリに書き戻す処理に対応しているため，この micro-op のコミットは追い出し対象の VM レジスタがもはや FPPR に割り当てられていないことを意味する．そのため，この VM レジスタに割り当てられていた FPPR の番号はフリーリストに返却される．

5.4 VM レジスタ値ストア (st.vm)

最後に，st.vm (図 3 (b) の 11 行目) の処理について説明する．この命令は，R[#0]+R[#1] の処理結果を R[#2] に書き込む．st.vm は，前述の ld.vm と同じ micro-op で構成される．ただし，vmmov に関してのみ，ld.vm とはソース・オペランドとデスティネーション・オペランドが逆となる．すなわち，ソース・オペランドの論理レジスタの値を，対象の VM レジスタにマップされた FPPR にムーブする処理を行う．

st.vm はコミット時に特別な処理が必要である．st.vm は対象の VM レジスタの値を上書きするため，その VM レジスタに前にマップされていた FPPR は解放可能になる．そのため，その FPPR 番号はフリーリストに返却される．

5.5 投機実行のリカバリとコンテキストスイッチ

投機ミスから回復する際，OFAR は VMRT と r_{vmpc}，r_{base} をロールバックする必要がある．この操作を実装するために，これらのユニットはレジスタ・リネーム・テーブルなどと同様にチェックポイントがとられる．

IDREG のリカバリは，r_{vmpc} 内の VMPC に応じて行われる．もし VMPC がリカバリの前後で変わらない場合，プロセッサは IDREG に対して何の操作も行わない．それ以外の場合，プロセッサは IDREG をリセットし，IDFU が VM レジスタ ID の再フェッチを行う．

コンテキスト・スイッチが起こった場合，FPPR にマッ

Vol.2019-ARC-240 No.32
Vol.2019-SLDM-191 No.32
Vol.2019-EMB-53 No.32
2019/2/28

プされたすべての VM レジスタは st.vmopflush を用いて書き戻す必要がある．この操作により，OFAR のためのすべての FPPR は解放され，スイッチ先のスレッドが利用可能になる．

5.6 分岐 VM 命令

本節では，4 節で説明した vmpc.br と vmpc.ijmp の動作について説明する．

vmpc.br は，VM オペランド・フィールド内の即値を VMPC に加算する命令である．この命令は，直接分岐 VM 命令のハンドラ中で VMPC を更新するために用いられる．すなわち，分岐結果が成立した際に VM オペランド・フィールド内の即値をディスプレースメントとして VMPC を更新する．この命令は，vmpc.inc と同様にフェッチステージで BTB を用いて vmpc.br であると判定される．その後，プロセッサは次の VMPC を予測し，IDFU に対応する VM レジスタ ID のフェッチを開始する．ただし，vmpc.inc と違い，VMPC の予測では BTB から読み出した即値を用いない．その代わり，IDFU がフェッチした VM オペランド・フィールド Va 内の即値を VMPC に加算する．この VMPC の予測に Va を用いるため，インタプリタ開発者は 分岐 VM 命令が VMPC に加算するディスプレースメントを必ず Va に持つように設計する．

vmpc.ijmp は，VMPC を論理レジスタ r_j 内の値で更新する命令で，間接分岐 VM 命令のハンドラの中で，分岐結果が成立した際に VMPC をターゲット・アドレスで更新するために用いられる．この命令は，vmpc.inc や vmpc.br と違い，フェッチ・ステージでは次の VMPC の予測や VMOpFetch による投機的な VM レジスタ ID のフェッチを行わない．その代わり，この命令は実行されるとそれ以降のすべての命令をフラッシュし，実行によって得られた正しい VMPC を用いて VMOpFetch に VM レジスタ ID をフェッチさせる．これは，次の VMPC が vmpc.ijmp のソース・オペランドの論理レジスタ内の値で決まるため，その予測が困難なためである．

vmpc.ijmp は，必ずパイプライン・フラッシュを行うため性能低下を引き起こすが，実際の性能への影響は小さい．これは，一般的にインタプリタでは間接分岐の VM 命令はほとんど実行されないためである．また，そもそもそうした VM 命令が実装されないことも多い．この理由としては，間接分岐 VM 命令は実行時に動的に生成されたターゲット・アドレスのセキュリティ面での検証が必要で，オーバーヘッドが大きいことがあげられる．

6. 評価結果

6.1 評価環境

提案手法を Sniper シミュレータ [4] と LLInt [2] を用いて評価した．基本的なハードウェアのパラメータは Intel の Skylake プロセッサに基づいており，次の 2 つのシステムをシミュレータ上に実装した．

(1) BASE: ベースラインの OoO プロセッサ
(2) OFAR: BASE に提案手法を実装したもの

表 2 にパラメータの詳細を示す．OFAR の VMRT のエントリ数は 32 とする．これは，3.1 節で述べたように，大部分の関数は 32 個以下の VM レジスタを使用するためである．LLInt は gcc -O3 でコンパイルし，JIT コンパイラとガーベージコレクタは無効化している．評価には，SunSpider [1] と v8 [6] の 2 つの JavaScript ベンチマーク・スイートを使用した．すべてのベンチマークでは，実行開始から終了までの全体をシミュレーションした．

表 2: アーキテクチャパラメータ．

	パラメータ
プロセッサ	6-way アウト・オブ・オーダー 224 エントリ ROB 72 エントリ ロード・キュー 56 エントリ ストア・キュー 97 エントリ 発行キュー グローバル分岐予測機 512 エントリ/way 4-way BTB
メモリ・システム	128 エントリ ITLB 64 エントリ L1 DTLB 512 エントリ L2 TLB 32 KB 8-way L1 命令キャッシュ 32KB 8-way L1 データ・キャッシュ (4-cycle) 256KB 8-way L2 キャッシュ (9-cycle) 2MB 16-way L3 キャッシュ (35-cycle) メイン・メモリ (66 ns) L1・L2 ストライド・データ・プリフェッチャ

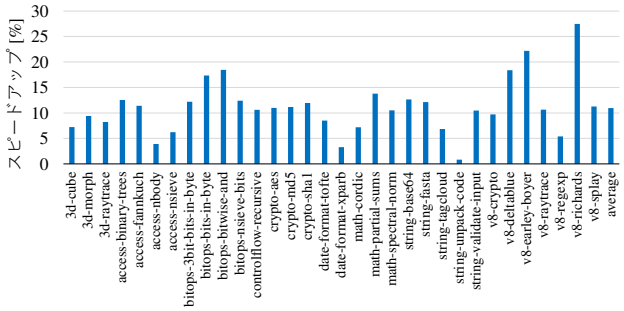

図 7: BASE に対する性能向上 % (大きい方が良い)．

6.2 性能と消費エネルギー

図 7 に，BASE に対する性能向上を示す．OFAR は平均で **10.9%**，v8-richards において最大で **27.4%**だけ性能が向上している．

図 8 に，BASE に対する OFAR の消費エネルギー削減率を示す．2 つのシステムの消費エネルギーは，McPAT [9] を用いて 22 nm プロセスを想定して見積もった．同図よ

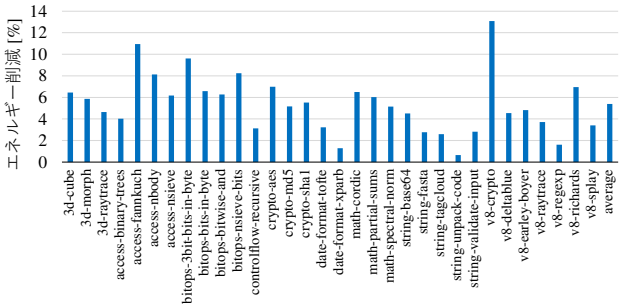

図 8: BASE に対するエネルギー消費削減 % (高い方が良い).

り，OFAR は平均で 5.38%の消費エネルギーが削減している．これは，VMRegMap によって多くのメモリ・アクセスがより消費エネルギーの小さいレジスタ間ムーブに変換されたことや，VMOpFetch によって VM レジスタ ID ロード命令が取り除かれ，命令数が削減されたためである．

また，これらの性能向上と消費エネルギー削減の結果，OFAR は BASE に対して平均で 14.7% エネルギー遅延積を改善した．

6.3 面積

McPAT [9] と CACTI-P [10] を用いて，22 nm プロセスを想定して OFAR の面積オーバーヘッドを見積もった．ここで，OFAR の面積は OoO プロセッサの面積と VMRT の面積の合計とした．これは，OFAR の面積オーバーヘッドは RAM ベースのユニットである VMRT によるものが支配的であるためである．IDFU は，主に数個のレジスタ (IDREG) とメモリ・アドレス計算のための加算器で構成されるため，その面積オーバーヘッドは非常に小さい．評価の結果，OFAR により面積は 0.4%増加した．

7. 結論

動的スクリプト言語は，高い生産性と移植性のため現在では広く用いられている．しかし，これらの言語の多くは，インタプリタ実行時にオペランド・アクセスに起因する大きなオーバーヘッドを持つ．このオペランド・アクセスのオーバーヘッドを削減するために，我々は OFAR を提案した．OFAR は，オペランドの値をメモリから浮動小数点の物理レジスタにマップし，オペランドの情報をプロセッサのフロントエンドでフェッチする．評価の結果， OFAR は平均で性能が 10.9%向上することを示した．

謝辞 なお，本研究は一部，JSPS 科研費 JP19H01105，JP16H05855，JSPS 特別研究員奨励費 JP17J10388，未来社会創造事業 JPMJMI18E1 による．

参考文献

[1] Apple. Sunspider javascript benchmark. `https://webkit.org/perf/sunspider/sunspider.html`.
[2] Apple. Webkit, javascriptcore. `https://github.com/WebKit/webkit/tree/master/Source/JavaScriptCore`.
[3] BrowserBench. Speedometer 1.0. `http://browserbench.org/Speedometer/`.
[4] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(3), 2014.
[5] M. A. Ertl. Stack caching for interpreters. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 315–327, 1995.
[6] Google. V8 benchmark suite. `http://www.netchain.com/Tools/v8/`.
[7] Google. V8 javascript engine. `https://v8.dev/`.
[8] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes Filho. The implementation of lua 5.0. *Journal of Universal Computer Science (J.UCS)*, 11(7):1159–1176, 2005.
[9] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceeding of the 42nd International Symposium on Microarchitecture (MICRO)*, pages 469–480, 2009.
[10] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi. Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 694–701, 2011.
[11] Microsoft. Chakracore. `https://github.com/microsoft/ChakraCore`.
[12] Mozilla. Spidermonkey. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`.
[13] Parrot Foundation. Parrot vm. `http://parrot.org`.
[14] Python Software Foundation. Cpython. `https://github.com/python/cpython`.
[15] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. Jsmeter: Comparing the behavior of javascript benchmarks with real web applications. In *Proceedings of the USENIX Conference on Web Application Development (WebApps)*, pages 3–3, 2010.
[16] Y. Shi. *Virtual Machine Showdown: Stack versus Registers*. PhD thesis, University of Dublin, Trinity College, 2007.
[17] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(4):2:1–2:36, 2008.