組込みシステム向け JavaScript 仮想機械生成系 eJSTK における合成命令の設計と実装

野中 智矢[1,†1,a)]　鵜川 始陽[1,b)]

概要：組込みシステムは一般に，搭載しているメモリが少なく CPU も遅い．そのため，組込みシステム向けの JavaScript 仮想機械（VM）では，VM の肥大化を避けながら高速化する必要がある．本研究では，組込みシステム向け JavaScript VM に，定数ロード命令と演算命令を合成した合成命令を追加することで実行を高速化する．一般には，合成命令を追加すると VM は大きくなってしまうので，本研究では，合成命令と素となる命令と合成命令とでコードの一部を共有することで，サイズの増加を抑える．さらに，合成命令は定数になったオペランドのデータ型に特化させ，型ディスパッチを単純化する．このような方針で生成した合成命令を持つ VM を生成するシステムに加え，合成命令を使って JavaScript のプログラムをコンパイルするコンパイラを開発した．

キーワード：合成命令，仮想機械，インタプリタ，JavaScript，組込みシステム，型ディスパッチ

Design and Implementation of Superinstructions for JavaScript Virtual Machine Generation System for Embedded Systems eJSTK

Tomoya Nonaka[1,†1,a)]　Tomoharu Ugawa[1,b)]

Abstract: Embedded systems generally have small amount of memory and slow CPUs. Therefore, JavaScript virtual machines (VMs) for such systems are desirable to be improved their execution speed while avoiding bloating VMs. In this research, we introduce superinstructions that are combinations of constant load instructions and computation instructions to improve execution speed. In general, introducing superinstructions bloats VMs. Thus, we designed superinstructions so that they share their implementation code with the computing instructions that they are made from. Furthermore, we simplify type-based dispatching code of superinstructions by specializing to the datatypes of their constant operands. We developed a VM generator that generates VMs that have superinstructions generated by these approach and a compiler that compiles JavaScript programs using the superinstructions.

Keywords: superinstruction, virtual machine, interpreter, JavaScript, embedded system

1. はじめに

近年，組込みシステムの開発が盛んになっている．Internet of Things (IoT) で利用するために様々な組込みシステムの開発が行われており，そのために，小型のプロセッサと小容量のメモリを搭載し，センサや通信機能を備えた様々なデバイスが簡単に手に入るようになっている．このようなデバイスのソフトウェア開発のために，eJS [1], [2]，JerryScript[*1]，mJS[*2] などの組込みシステム向けの JavaScript 処理系も開発されている．

組込みシステム向けの JavaScript 処理系では，小さなメ

[1] 高知工科大学
Kochi University of Technology, Kami, Kochi 782–0003, Japan
[†1] 現在，株式会社コロプラ
Presently with COLOPL, Inc., LTD.
[a)] nonaka@pl.info.kochi-tech.ac.jp
[b)] ugawa.tomoharu@kochi-tech.ac.jp

[*1] http://jerryscript.net
[*2] https://github.com/cesanta/mjs

モリで動作するように，仮想機械（VM）を小型化する工夫がなされている．そのような処理系は，メモリ等の資源の制約から just-in-time（JIT）コンパイルのような多くのメモリを要する技術は利用せず，純粋なインタプリタとして実装されている．そのため実行速度はブラウザに組込まれた JavaScript VM などと比べると遅い [2]．しかし，組込みシステムでも実行速度は重要である．特に，バッテリ駆動のシステムでは，短時間で必要な処理を実行し終えて CPU をスリープモードにできれば，消費電力の削減につながる．

そこで本研究では，組込みシステム向け JavaScript 処理系である eJS を対象に，合成命令 [3] の導入によって VM の肥大化を抑えつつ実行速度を改善することを目指す．eJS は，組込みシステムではシステム毎に特定のアプリケーションしか実行しないという特徴を利用し，そのアプリケーションに必要な機能だけを持つようにカスタマイズした VM をアプリケーション毎に生成することで，VM を小型化している．本研究では，VM をカスタマイズする一手段として，頻繁に実行される VM 命令を組合わせた合成命令を持つ VM を生成できるようにする．本研究では，定数のロード命令と演算命令の組合わせに限って合成する．

eJS は，JavaScript のプログラムを中間コードにコンパイルする eJS コンパイラと，その中間コードを実行する VM を生成する eJSTK (embedded JavaScript Tool Kit) で構成されている．eJSTK が生成する VM である eJSVM は RISC 型のレジスタマシンである．eJSVM の命令セットでは，演算命令のオペランドにはレジスタしか指定することができない．これにより，命令インタプリタを単純化し，VM を小型化している．しかし，定数との演算を行う際は，演算命令に先立って定数ロード命令で定数をレジスタにロードしておく必要がある．そのため，オペランドに定数も指定できる命令セットと比べると，実行速度の面では不利な点が二つある．まず，命令ディスパッチの回数が増える．また，レジスタは配列で実現しているので，定数ロード命令で定数を一旦メモリに格納し，演算命令でそれを読み出して使うことになる．この欠点を補うために，オペランドに定数を指定できれば，命令のデコード時にインタプリタの局所変数に格納された定数を演算に利用することが可能になる．本研究では頻繁に使われる定数ロード命令と演算命令に限って合成し，オペランドに定数をとれる新たな合成命令を追加した VM を生成できるようにする．

さらに本研究では，合成命令の実装を二つの方針で最適化する．まず，合成命令と合成元の演算命令で実装を共有し，VM の肥大化を抑える．次に，合成命令では定数オペランドの型が一意に決まることを利用して，合成命令の型ディスパッチのコードを単純化する．JavaScript の演算子はオペランドのデータ型についてオーバロードされている．例えば，加算演算子はオペランドに整数が与えられると数値の加算を行い，文字列が与えられると文字列を結合するなど，オペランドのデータ型によって様々な処理を行う．そのため，命令を実装するインタプリタのコードでは，オペランドのデータ型によって処理を選択する型ディスパッチを行う．しかし，合成命令では定数オペランドの型が静的に一意に決まることを利用すれば，型ディスパッチのコードを単純化することができる．これには，eJSTK が持つ，ユーザの指定に従ってオペランドのデータ型を制限したインタプリタを生成する仕組みを利用する．

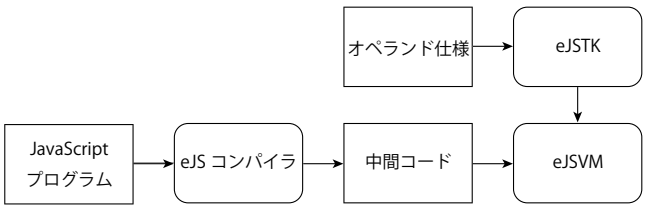


図 1: eJS の構成
Fig. 1 Structure of eJS.

以上のことを考慮に入れ，4 章で実装を最適化しやすい合成命令を設計する．4 章では，合成命令と合成元の命令でコードを共有するしたり，複製して定数のデータ型に特化させたりする 5 種類の実装方法を検討する．これらの実装方法は 6 章で比較する．

2. eJS 処理系

2.1 eJS の全体像

eJS [1], [2] はフットプリントの小さい VM を作るために，アプリケーションに特化した VM を生成する仕組みを持つ JavaScript 処理系である．eJS の VM 生成系である eJSTK は，特に演算命令のオペランドのデータ型について特化した VM を生成する．eJS は，ECMA Script 5.1 [4] のうち，`eval` 関数のような複雑な対応が必要な機能を除いたサブセットに対応している．

図 1 に eJS の構成を示す．ユーザは，対象のアプリケーションで各命令のオペランドに与えられる可能性のあるデータ型を，**オペランド仕様**として記述する．eJSTK はオペランド仕様に基づいて，最適化された命令インタプリタを持つ VM である eJSVM を生成する．eJS コンパイラは，アプリケーションを VM 命令列にコンパイルする．この VM 命令列を組込みシステム上に配備した eJSVM で実行する．

eJS はサーバサイド JavaScript VM を基にして開発されており，現時点ではまだ 64 ビット環境しかサポートしていないなど，組込みシステムに適合しない箇所も残っている．しかし，本研究の合成命令を追加する手法は，このような箇所が今後変更されても有効である．

2.2 eJSVM

eJSVM は C 言語で記述された JavaScript VM である．

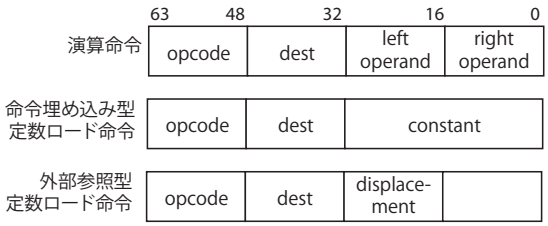


図 2: eJS の命令フォーマット

Fig. 2 Format of eJS instructions

VM のソースコードのうち，VM 命令のインタプリタは eJSTK の中核をなす VM 命令生成器によって生成される．オブジェクトに対する操作やガーベージコレクションなどの実行時システム部分と組込み関数は，現在のところ C 言語で記述された固定のソースコードを用いている．

2.2.1 VM データ型

JavaScript は動的型の言語である．eJSVM はそれを扱うために，JavaScript の値にデータ型を付加して扱っている．eJSVM 内部の **VM データ型**は，JavaScript の仕様で定められた型と一対一では対応していない．例えば JavaScript の Number 型は JavaScript の仕様では C 言語の double 型相当の型として定められているが，eJSVM では，固定長整数の場合，最適化のために fixnum 型という VM データ型を使い，それ以外は flonum 型という VM データ型を使っている．また，JavaScript の Boolean 型，`null`，`undefined` は，全て special 型という VM データ型になっている．

eJSVM では，VM データ型を，ポインタの一部のビットに記録した**ポインタタグ**と，ポインタで指されるヒープ中のデータ本体のヘッダに格納した型情報（以降では**ヘッダタグ**と言う）を併用して表している．ポインタタグの領域はビット数が限られているので，ワード中にデータを埋め込んだデータ型である fixnum 型と special 型の他，flonum 型や string 型といった頻繁に使われるデータ型に一意なタグを与えてポインタタグだけで判定できるようにしている．それ以外のデータ型は共通のポインタタグを持ちヘッダタグで型を判定する．なお，eJSTK はヒープ中にデータを持つデータ型のうち，どのデータ型をポインタタグで判定できるようにするかもカスタマイズできる．

2.2.2 命令セットと命令の内部フォーマット

eJSVM は RISC 型の命令セットを持つレジスタマシンである．関数呼出し毎に異なるレジスタの集合を持ち，レジスタは必要な数だけ使うことができる．レジスタの領域は C 言語の配列で実装された実行スタック上に割り当てられている．レジスタは JavaScript の局所変数のうちクロージャから参照されない変数や，JavaScript の式を計算する途中結果を格納するために用いる．演算命令が直接アクセスできるのはレジスタのみであり，クロージャから参照される局所変数やオブジェクトのプロパティは，専用の命令を使ってレジスタを介してアクセスする．

eJSVM 内部では，どの命令も 64 ビットで保持されており，そのうち 16 ビットがオペコードのフィールドである．eJSVM はスレッディッドコードで命令ディスパッチを行うので，実行中はオペコードは使われない．

本研究で合成の対象にする演算命令と定数ロード命令の命令フォーマットを**図 2** に示す．演算命令は 3 個のレジスタオペランドを取る．最初のオペランドは出力レジスタの番号で，続く 2 個のオペランドは演算対象のデータが格納されたレジスタの番号である．これら 2 個の入力オペランドを，以降は**左オペランド**と**右オペランド**と呼ぶ．各オペランドには 16 ビットのフィールドが割り当てられている．

定数ロード命令には，命令中に定数を保持している命令埋め込み型命令と，命令の外部に定数を置いている外部参照型命令がある．定数ロード命令でロードできるのは，fixnum 型，flonum 型，special 型，string 型，および正規表現を表す regexp 型の定数である．このうち，fixnum 型と special 型の定数をロードする命令は命令埋め込み型命令である．fixnum 型の定数をロードする `fixnum` 命令は，16 ビットの出力レジスタ番号のフィールドと 32 ビットの定数のフィールドを持つ．32 ビットを超える整数定数は外部参照型命令である `number` 命令でロードする．

32 ビットを超える fixnum 型の定数，flonum 型，string 型および regexp 型の定数をロードする命令は，外部参照型命令である．外部参照型命令でロードする定数は，中間コードを VM にロードするときに作られる．命令のワードには，定数が置かれている位置を示す 16 ビットのディスプレイスメントが格納される．

2.2.3 型ディスパッチ

JavaScript の演算子は，オペランドのデータ型についてオーバーロードされている．そのため，演算命令では各オペランドのデータ型を判定して適切な処理を選択する型ディスパッチの処理が必要になる．eJSTK の中核をなす VM 命令生成器は，ユーザが指定したオペランド仕様に従って，特化した型ディスパッチのコードを生成する．

型ディスパッチは，各オペランドについてポインタタグとヘッダタグを判定して処理を選択するような，ネストした `switch` 文で実現されている．VM 命令生成器は，各オペランドのポインタタグを順に判定し，その後，各オペランドのヘッダタグを判定する型ディスパッチのコードを生成する．ただし，ユーザが指定したオペランド仕様に従って `switch` 文の分岐先を制限したり，同じ処理をするコードが 1 箇所にしか現れないように分岐先をまとめたり，その結果，分岐先が一つになった `switch` 文を取り除いて，そのタグの判定を行わないようにする最適化を行っている．

例えば，**図 3**(a) は最適化されていない `add` 命令の本体である．ここで `v1`，`v2` はオペランドが格納された変数であり，`dst` は結果を格納するレジスタに展開されるマクロである．また，`PTAG` はポインタタグを，`HTAG` はヘッダタ

```
ADD_HEAD:
switch(PTAG(v1)){
case FIXNUM: {
  switch(PTAG(v2)){
  case FIXNUM: {
    // 整数同士の加算
    cint s = fixnum_to_cint(v1) + fixnum_to_cint(v2);
    dst = cint_to_number(s);
  } break;
  caes GENERIC: {
    switch(HTAG(v2)){
    case OBJECT: {
      // 数値とオブジェクトの加算
      v2 = to_number(context, v2);
      goto ADD_HEAD;
    } break;
...
}
```

(a) 最適化なし
(a) No optimization.

```
ADD_HEAD:
switch(PTAG(v1)){
case FIXNUM: {
    // 整数同士の加算
    cint s = fixnum_to_cint(v1) + fixnum_to_cint(v2);
    dst = cint_to_number(s);
  } break;
case STRING: {
    // 文字列同士の連結
    dst = ejs_string_concat(context, v1, v2);
  } break;
}
```

(b) add 命令が整数同士と文字列同士の演算のみサポートするよう最適化
(b) Optimized add accepting a pair of integers and a pair of strings.

図 3: add 命令の本体
Fig. 3 Body of add instruction.

グを得るマクロである．GENERIC は複数の VM データ型で共有されたポインタタグであり，データ型を判定するには，さらにヘッダタグを調べる必要がある．図 3(b) に整数同士の加算と文字列同士の連結だけをサポートするようにし最適化した add 命令を示す．この例では，v1 だけ判定すれば処理を選択できる．さらに，fixnum 型と string 型は一意なヘッダタグを持つので，ヘッダタグの判定も取り除かれ，1 回の switch 文で型ディスパッチが行われている．

2.2.4 命令インタプリタ

eJSVM の命令インタプリタは，スレッディッドコード [5] で命令ディスパッチを行っている．各命令を処理するコードは，次のような処理で構成されている．

(1) 命令をデコードし，各オペランドをインタプリタの局所変数に格納する．
(2) 命令本体を実行する．
(3) 次の命令にジャンプする．

例えば，add 命令は**図 4** の I_ADD のようになっている．

```
I_ADD:
{
  // 出力レジスタ
  Register r0 = get_first_operand_reg(insn);
  // 左オペランド
  JSValue v1 = get_second_operand_value(insn);
  // 右オペランド
  JSValue v2 = get_third_operand_value(insn);
#include "insns/add.inc"
}
NEXT_INSN_INCPC();
...
I_FIXNUM:
{
  // 出力レジスタ
  Register r0 = get_first_operand(insn);
  // 定数（fixnum）
  int64_t  i1 = get_small_immediate(insn);
  regbase[r0] = cint_to_fixnum((cint) i1);
}
NEXT_INSN_INCPC();
```

図 4: VM 命令の実装
Fig. 4 Implementation of VM instructions

まず命令をデコードする．出力オペランドについては，命令からレジスタ番号を読み出し，入力オペランドについては，命令で指定されたレジスタからデータを読み出して，インタプリタの局所変数 r0，v1，v2 に格納する．ここで，Register はレジスタ番号を格納する変数の型，JSValue は JavaScript の値を格納する変数の型である．その後，これらの変数を使う命令本体を実行する．命令本体は VM 命令生成器によって，別のファイル（図 4 では insns/add.c）に図 3 のように生成されている．

fixnum 型定数のロード命令である fixnum 命令は，図 4 の I_FIXNUM のようになっている．まず，add 命令と同様に，命令をデコードする．ここで，get_small_immediate は命令に埋め込まれた 32 ビットの定数のフィールドを取り出すマクロである．その後，命令本体の動作として，cint_to_fixnum マクロによって取り出した定数に整数のタグをを付けて，r0 で示されたレジスタに格納する．ここで，regbase がレジスタの配列である．定数ロード命令の本体は VM 命令生成器では生成されず，固定のコードが使われる．

2.3 VM 命令生成器

eJSVM の命令を生成する VM 命令生成器は，オペランド仕様の他に，各命令の仕様を記述した**命令定義ファイル**を入力として受け取る．命令定義ファイルには専用の命令定義用のドメイン特化言語（DSL）により，各オペランドのデータ型の組合わせに対して実行すべき C 言語のプログラム断片が記述されている．VM 命令生成器は，オペランド仕様に従って生成した型ディスパッチする switch 文と，命令定義ファイルに記述された C 言語のプログラム断

片を組合わせて，図 3 のような VM 命令の本体のコードを生成する [2].

3. 合成命令の設計

本研究では，定数ロード命令と演算命令の組合わせに絞って合成する．これによって，演算命令のオペランドに定数を指定できる命令が作られる．このとき合成元となった演算命令を**合成元命令**と呼ぶことにする．例えば，fixnum 型定数をロードする `fixnum` 命令と，それを左オペランドに使う `add` 命令を合成して `addfixreg` 命令を作る．以降では，このような合成を「`fixnum` 命令を `add` 命令の左オペランドに合成する」と言う．また，定数が合成されたオペランドを**定数オペランド**と呼ぶ．合成命令の名前は，演算命令名の後に，各オペランドについて定数ロード命令を合成していれば定数の型名の省略形を，そうでなければ `reg` を付けた名前で表す．

3.1 合成命令の命令フォーマット

演算命令の各オペランドのフィールドは 16 ビットある（2.2.2 節参照）．合成命令では，その 16 ビットにレジスタ番号ではなく定数を表すビット列を格納する．fixnum 型の定数をロードする `fixnum` 命令以外は，定数を 16 ビットで表現できる．`fixnum` 命令は，最大 32 ビットの整数をロードできるが，`fixnum` 命令を合成するときは，16 ビット以下で表現可能な整数をロードするときのみ利用できる合成命令を作る．頻繁に利用される整数定数は小さいことが多いため，これでも問題はないと考えている．さらに，将来，eJSVM が命令のビット長を減らし，定数オペランドに指定できる命令長が 16 ビットよりさらに短くなったとしても，影響は小さいと考えている．

3.2 定数オペランドの制限

合成命令の本体で型ディスパッチを行うコードの最適化を余地を増やすため，合成命令の定数オペランドに指定できる定数に，以下のような制限を加える．

まず，合成命令毎に，定数オペランドの VM データ型が一意に決まるようにする．eJSVM の命令セットでは，定数ロード命令と，その命令でロードされる定数の VM データ型がほぼ一対一で対応している．`number` 命令だけは例外で，fixnum 型で表される 32 ビットを超える整数と，flonum 型で表される浮動小数点数の両方をロードできる．本研究では 3.1 節で 16 ビットを超える整数定数をオペランドに指定しないことにしたので，`number` 命令を合成した命令は，flonum 型の定数だけをロードするようにする．

次に，定数オペランドはポインタタグだけで判定できるデータ型に限ることにする．これによって，型ディスパッチのコードでポインタタグを判定するオペランドの順番を入れかえるだけで，合成命令と合成元命令の実装でコードを共有できるようになる（詳しくは 4.3 節で述べる）．eJSVM のデフォルトの設定では，正規表現を表す regexp 型以外は全てポインタタグだけで判定できる．将来，eJS がポインタタグに利用できるビット長を減らした場合，いくつかの定数はポインタタグだけでは判定できなくなる可能性はある．しかし，eJS では，どの VM データ型をポインタタグだけで判定できるようにするかはアプリケーション毎にカスタマイズできるため，合成命令の対象とする VM データ型をユーザが指定できる．

4. 合成命令の実装

2 章で述べたように，eJSVM の演算命令を処理するコードは，

- 命令デコード
- 型ディスパッチ
- 型ディスパッチにより選択される演算処理（以下，単に**演算処理**と言う）

からなる．このうち，型ディスパッチのコードはオペランドのデータ型が限定されると，それを利用して最適化ができる．

本研究では，次の方針で合成命令を実装し VM の肥大化を抑えつつ高速な実行を実現する．

方針 1 合成命令の型ディスパッチや演算処理は，合成元命令と実装のコードを共有することで VM の肥大化を抑える．

方針 2 定数オペランドのデータ型に特化した型ディスパッチを行うことで高速な実行を実現する．

これら二つの方針は相反するので，それぞれの方針の要素をどの程度取り込むかによって，様々な合成命令の実現方法が考えられる．

方針 1 だけに従った実装では，合成命令と合成元命令で，型ディスパッチと演算処理を完全に共有することが考えられる．この実装を「**型ディスパッチを共有する実装**」と呼ぶことにする．この実装では，合成命令は命令デコード後に合成元命令の型ディスパッチの入口にジャンプする．

型ディスパッチを共有する実装は，合成命令では定数オペランドのデータ型が分かっていることを利用して改良できる．合成元命令の型ディスパッチの最初の `switch` 文で調べているオペランドが，合成命令では定数になっていた場合，その `switch` 文を省略し，型ディスパッチの入口ではなく，`switch` 文の分岐先にジャンプする．この実装を「**型ディスパッチを共有する実装（改良版）**」と呼ぶことにする．

次に，方針 2 だけに従った実装では，合成命令は合成元命令とは独立に実装し，合成命令がとり得るオペランドのデータ型にだけ特化した型ディスパッチを行う命令にすることが考えられる．この実装を「**特化した型ディスパッチを持つ実装**」と呼ぶことにする．

表 1: 合成命令の実装方法
Table 1 Implementation strategies of superinstructions.

実装方法	型デイスパッチ	演算処理	
実装 1	ナイーブな実装（4.1 節）	独立	独立
実装 2	型ディスパッチを共有する実装（4.2 節）	共有	共有
実装 3	型ディスパッチを共有する実装（改良版）（4.3 節）	一部共有	共有
実装 4	特化した型ディスパッチを持つ実装（4.4 節）	独立（特化）	独立
実装 5	特化した型ディスパッチを持つ実装（改良版）（4.5 節）	独立（特化）	共有

```
I_ADDFIXREG:
{
  // 出力レジスタ
  Register r0 = get_first_operand_reg(insn);
  // 左オペランド（整数定数）
  int16_t i1 = get_second_operand_int(insn);
  JSValue v1 = cint_to_fixnum(i1);
  // 右オペランド
  JSValue v2 = get_third_operand_value(insn);
#define DEFLABEL(l) ADDFIXREG ## l
#define USELABEL(l) ADDFIXREG ## l
// addfixreg.inc は add.inc とラベル名以外同じ
#include "insns/addfixreg.inc"
#undef DEFLABEL
#undef USELABEL
}
NEXT_INSN_INCPC();
```

図 5: ナイーブな合成命令の実装
Fig. 5 Naive implementation of a superinstruction.

特化した型ディスパッチを持つ実装であっても，演算処理は合成元命令と共有できる．この実装を，「**特化した型ディスパッチを持つ実装（改良版）**」と呼ぶことにする．この実装では，合成命令の型ディスパッチの各分岐先から，合成元命令の対応する演算処理にジャンプする．

本研究では，これらの 4 種類の実装に加えて，コードの共有や，定数のデータ型に特化した型ディスパッチを行わない「**ナイーブな実装**」の 5 種類を実装する．これら 5 種類の実装を**表 1** にまとめる．本章の以降では，各実装の詳細を説明する．

4.1 ナイーブな実装

ナイーブな実装では，合成命令は合成元命令と独立に実装する．合成命令を実装は，合成元命令の命令デコード処理を定数ロード処理に単純に置き換えたものにする．

例えば，`fixnum` 命令を `add` 命令の左オペランドに合成した `addfixreg` 命令は，**図 5** のようにする．左オペランドを格納する局所変数 `v1` には，命令中の左オペランドに対応するフィールドから整数を読み出し，`cint_to_fixnum` マクロで fixnum 型を表すタグを付けた値を格納する．その後は `add` 命令と同じ処理をするので，命令本体には合成元の `add` 命令の本体である `insn/add.inc` とほぼ同じ内容の `insn/addfixreg.inc` をインクルードする．ただし，

```
{
  Register r0;
  JSValue  v1, v2;
I_ADDFIXREG:
  r0 = get_first_operand_reg(insn);
  {
    int16_t i1 = get_second_operand_int(insn);
    v1 = cint_to_fixnum(i1);
  }
  v2 = get_third_operand_value(insn);
  goto I_ADD_BODY;
I_ADD:
  r0 = get_first_operand_reg(insn);
  v1 = get_second_operand_value(insn);
  v2 = get_third_operand_value(insn);
I_ADD_BODY:
#define DEFLABEL(l) ADD ## l
#define USELABEL(l) ADD ## l
#include "insns/add.inc"
#undef DEFLABEL
#undef USELABEL
}
NEXT_INSN_INCPC();
```

図 6: 型ディスパッチを共有する実装
Fig. 6 Implementation in which instructions share type-based dispatching code.

`insn/addfixreg.inc` では命令本体のコードで局所的に使われるラベルの名前が衝突するのを防ぐようにしている．命令本体のコードで使われるラベルには，命令定義ファイル中に記述されたラベルと，VM 命令生成器が自動的に生成するラベルがある．図 5 の `DEFLABEL` マクロと `USELABEL` マクロは，前者の名前の衝突を避けるために命令定義ファイル中で使われるラベル名にプレフィクスを付けるためのものである．命令定義ファイルは，生成される命令本体のコードでラベルを定義する際に

```
DEFLABEL(HEAD):
```

と，ラベルにジャンプする際に

```
goto USELABEL(HEAD);
```

となるように記述する．VM 命令生成器が生成するラベルの名前の衝突を避けるためには，合成元命令の本体と，ラベル名のプレフィクスだけが異なる命令本体のコードを合成命令毎に生成する．

4.2 型ディスパッチを共有する実装

型ディスパッチを共有する実装では，合成命令の実装において命令をデコードした後，goto 文によって合成元命令の型ディスパッチの入口にジャンプする．これにより，合成元命令と合成命令でコードを共有する．命令のオペランドを格納するインタプリタの局所変数も，合成元命令と合成命令で共有する必要があるため，合成命令は合成元と同じスコープに作る．

図 6 に，型ディスパッチを共有する実装による addfixreg 命令を示す．I_ADDFIXREG が addfixreg 命令の開始位置である．命令をデコードして，オペランドの値をインタプリタの局所変数 r0，v1，v2 に格納した後，add 命令の本体である insns/add.inc の直前 I_ADD_BODY にジャンプしている．

4.3 型ディスパッチを共有する実装（改良版）

型ディスパッチを共有する実装（改良版）では，合成命令の命令デコード後に，可能であれば合成元命令の型ディスパッチの途中にジャンプする．これによって，switch 文の実行を省略する．

例えば addfixreg 命令の実装は**図 7** のようになる．合成命令の実装は，型ディスパッチを共有する実装と似ているが，addfixreg 命令の命令デコード後に，goto 文でジャンプする先のラベルが異なる．この実装では，ジャンプ先が add 命令本体である add.inc 中の，型ディスパッチの途中に付けられたラベルになる．addfixreg 命令では左オペランド（v1 の値）が fixnum 型に決まるので，左オペランドの型によるディスパッチを省略して，右オペランド（v2 の値）によるディスパッチの入口にジャンプする．

これを可能にするために，VM 命令生成器は，型ディスパッチを実現する switch 文のネストを木構造とみなして，オペランドのデータ型の一部が分かっている場合に，どこまで静的に木を降りることがでるかを，全ての場合についてあらかじめ計算し，ラベルを付けておく．例えば，add 命令では，左オペランドだけが fixnum 型と分かっている場合，TLadd_fixnum_any というラベルを付ける．ここで any はデータ型がまだ分かっていないオペランドに対応する．

さらに，型ディスパッチでできるだけ多くの型の判定を省略できるように，型を判定する順序を入れ替える．図 7 の add.inc のように左オペランド，右オペランドの順に調べる型ディスパッチのコードでは，右オペランドだけが定数でも型ディスパッチの途中にジャンプすることができない．そこで，右オペランドだけが定数の合成命令を追加するときは，右オペランドから判定するような型ディスパッチのコードを生成する．同じ命令に対して，左オペランドを定数にした合成命令と右オペランドを定数にした合成命令の両方を追加するときは，どちらを優先するかをユーザが選択するようにする．eJSVM はまず全てオペランドのポインタタグを判定してから，必要であればヘッダタグを判定する．これは，ヘッダタグの判定にはメモリアクセスを伴いコストが大きからである．定数オペランドのデータ型は，3.2 節の設計によってポインタタグだけで判定できることになっているので，順序を入れ変えるのはポインタタグの判定だけでよい．そのため，ヘッダタグの判定が増えることはない．

```
{
  Register r0;
  JSValue  v1, v2;
I_ADDFIXREG:
  r0 = get_first_operand_reg(insn);
  {
    int16_t i1 = get_second_operand_int(insn);
    v1 = cint_to_fixnum(i1);
  }
  v2 = get_third_operand_value(insn);
  goto TL_add_fixnum_any;
I_ADD:
  r0 = get_first_operand_reg(insn);
  v1 = get_second_operand_value(insn);
  v2 = get_third_operand_value(insn);
#define DEFLABEL(l) ADD ## l
#define USELABEL(l) ADD ## l
#include "insns/add.inc"
#undef DEFLABEL
#undef USELABEL
}
NEXT_INSN_INCPC();
```

(a) 合成命令の実装

(a) Implementation of a superinstruction.

```
switch (PTAG(v1)) {
case FIXNUM:
TLadd_fixnum_any:
  switch (PTAG(v2)) {
  case FIXNUM:
  TLadd_fixnum_fixnum:
    cint s = fixnum_to_cint(v1) + fixnum_to_cint(v2);
    regbase[r0] = cint_to_number(s);
    break;
 ...
 }
...
}
```

(b) 合成元命令の本体（add.inc）

(b) Body of an original instruction （add.inc）.

図 7: 型ディスパッチを共有する実装（改良版）

Fig. 7 Implementation in which instructions share type-based dispatching code (improved).

4.4 特化した型ディスパッチを持つ実装

定数オペランドのデータ型は，3.2 節の設計によって，合成命令毎に一意に決定できる．これを利用して，オペランドのデータ型に特化した高速な型ディスパッチのコードを生成する．

```
{
  Register r0;
  JSValue  v1, v2;
I_ADDFIXREG:
  r0 = get_first_operand_reg(insn);
  {
    int16_t i1 = get_second_operand_int(insn);
    v1 = cint_to_fixnum(i1);
  }
  v2 = get_third_operand_value(insn);
#define DEFLABEL(l) ADDFIXREG ## l
#define USELABEL(l) ADD ## l
#include "insns/addfixreg.inc"
#undef DEFLABEL
#undef USELABEL
  NEXT_INSN_INCPC();
I_ADD:
  r0 = get_first_operand_reg(insn);
  v1 = get_second_operand_value(insn);
  v2 = get_third_operand_value(insn);
#define DEFLABEL(l) ADD ## l
#define USELABEL(l) ADD ## l
#include "insns/add.inc"
#undef DEFLABEL
#undef USELABEL
}
NEXT_INSN_INCPC();
```

(a) 合成命令の実装
(a) Implementation of VM instructions.

```
DEFLABEL(HEAD):
switch(PTAG(v2)){
case FIXNUM: {
  // 整数同士の加算
  cint s = fixnum_to_cint(v1) + fixnum_to_cint(v2);
  dst = cint_to_number(s);
} break;
caes GENERIC: {
  switch(HTAG(v2)){
  case OBJECT: {
    // 数値とオブジェクトの加算
    v2 = to_number(context, v2);
    goto USELABEL(HEAD);
  } break;
...
}
```

(b) 合成命令の本体（addfixreg.inc）
(b) Body of a superinstruction （addfixreg.inc）.

図 8: 特化した型ディスパッチを持つ実装
Fig. 8 Implementation that specializes type-based dispatching code.

図 **8**(a) に特化した型ディスパッチを持つ実装による addfixreg 命令を示す．addfixreg 命令の本体は，ナイーブな実装と異なり，左オペランドを fixnum 型に制限して VM 命令生成器で図 8(b) のように生成する．左オペランドを fixnum 型に制限すると，左オペランドに関する型ディパッチが行われなくなり，その分だけ switch 文のネストが浅くなる．これにより VM の肥大化が抑えられるとともに，型ディスパッチが高速になる．

```
switch (PTAG(v1)) {
case FIXNUM:
  switch (PTAG(v2)) {
  case FIXNUM:
  TLadd_fixnum_fixnum:
    cint s = fixnum_to_cint(v1) + fixnum_to_cint(v2);
    regbase[r0] = cint_to_number(s);
    break;
 ...
 }
...
}
```

(a) 合成元命令（add.inc）
(a) Original instruction （add.inc）.

```
switch (PTAG(v2)) {
case FIXNUM:
  goto TLadd_fixnum_fixnum;
...
}
```

(b) 合成命令（addfixreg.inc）
(b) Superinstruction （addfixreg.inc）.

図 9: 特化した型ディスパッチを持つ実装（改良版）
Fig. 9 Implementation that specializes type-based dispatching code (improved).

ただし，この実装ではナイーブな実装と異なり，合成命令でも合成元命令のコードを利用することがある．これは，定数オペランドを型変換した後，再度ディスパッチする場合があるからである．例えば JavaScript の加算演算子は，片方のオペランドが文字列型で他方のオペランドが整数型だったとき，整数型のオペランドを文字列型に変換した後文字列の連結を行う．この他にもオペランドの型変換が行われるデータ型の組み合わせがあり，それらを統一的にコンパクトなコードで処理するために，eJS では，型変換後に型ディスパッチの switch 文の入口にジャンプすることがある（図 3(a) 参照）．このとき，オペランドのデータ型を制限していると，定数オペランドを型変換したときにディスパッチ先がなくなっている可能性がある．例えば，addfixreg 命令の左オペランドを文字列型に変換すると，左オペランドを fixnum 型に制限した命令本体では処理することができない．

そのため，命令定義ファイル中の記述に由来する合成命令中のジャンプ先は，合成元命令本体のラベルにする．insns/addfixreg.inc をインクルードする直前の USELABEL マクロが ADD をプレフィクスに付けるのはそのためである．なお，insns/addfixreg.inc 中で定義したラベルは使われない．また，インタプリタの局所変数を共有できるように，合成命令は合成元命令と同じスコープに作る．

4.5 特化した型ディスパッチを持つ実装（改良版）

特化した型ディスパッチを持つ実装（改良版）では，特化した型ディスパッチを持つ実装と同様に，定数オペラン

```
function f(msg) {
  var ppfix = "=";
  return ppfix + msg + ppfix;
}
```

(a) JavaScript のプログラム

```
string r1 "="
add r2 r1 r2
add r1 r2 r1
```

(b) 合成命令を使わないコンパイル

```
string r1 "="
add r2 r1 r2
addregstr r1 r2 "="
```

(c) 合成命令を使ったコンパイル

図 10: コンパイラによる合成命令の利用

Fig. 10 Compilation using superinstructions.

ドのデータ型に特化した型ディスパッチを行う．さらに，型ディスパッチ後に合成元命令の対応する演算処理にジャンプする．**図 9** にこの実装による add 命令と addfixreg 命令の本体である add.inc と addfixreg.inc の一部を示す．add.inc では，型ディスパッチにより選択されるコードに TLadd_fixnum_fixnum のようなラベルを付けておく．そして，addfixreg.inc では，型ディスパッチの後，それらの中の適切なラベルにジャンプする．

5. コンパイラ

eJS コンパイラは，VM に実装された合成命令を使ってコンパイルする．eJS コンパイラは定数伝播の最適化を行う．これにより，演算命令の両方のオペランドが定数のときは定数ロード命令に置き換える．しかし，通常の eJSVM には定数をオペランドにとる演算命令は存在しないので，演算命令の両方のオペランドが定数のときにしか命令を置き換えない．

本研究では，これを拡張して演算命令の片方のオペランドの定義元の可能性が単一の定数ロード命令に決定でき，かつ，その定数ロード命令と演算命令を合成した命令が VM に実装されているときは，演算命令を合成命令で置き換えるようにする．このとき，定数ロード命令は消さずに残しておく．これは，定数ロード命令の結果が他の命令でも使われている可能性があるからである．他のどの命令でも使われなかった定数ロード命令は，後の最適化フェーズで冗長な命令として取り除かれる．

例えば**図 10**(a) のプログラムでは，変数 ppfix はレジスタに割り付けられ，図 10(b) のようにコンパイルされる．ここで，各命令の最初のオペランドは出力レジスタである．VM に右オペランドに文字列定数をとる addregstr 命令だけが実装されている場合，その命令を使って，図 10(c) のようになる．

合成命令は型ディスパッチが定数オペランドのデータ型に特化しており高速に実行できる．そのため，定数ロード命令が取り除けなくとも，合成命令を積極的に使うようにコンパイルする．

```
function triple(a, b, c) {
  return "(" + a + "," + b + "," + c + ")";
}
function main() {
  for (i = 0; i < 500000; i++)
    triple("a", "b", "c");
}
main();
```

図 11: triple ベンチマーク

Fig. 11 triple benchmark.

6. 性能評価

4 章で述べた各実装による VM を生成する eJSTK を実装し，生成された eJSVM の性能を評価した．ベンチマークプログラムには，文献 [2] と同様に eJSVM の評価に適するようにループ回数などを調整した SunSpider ベンチマーク[*3] のサブセットと，文字列定数のロード命令を合成するプログラムとして，**図 11** に示す triple を用いた．

ベンチマークプログラム毎に**表 2** に示す合成命令を実装した VM を生成し実験した．これは，次のようにして選択した．まず，考えられるあらゆる合成命令を実装した eJSVM で，命令実行回数を数えながら，ベンチマークプログラムを実行した．その結果，実行回数が全命令の実行回数の 1%以上を占めた合成命令を選択した．型ディスパッチを共有する実装（改良版）で，左オペランドが定数の合成命令と右オペランドが定数の合成命令を追加するときは，実行回数が多かった命令を優先して型ディスパッチを省略するようにした．

これら，ベンチマークプログラムに特化した合成命令を実装した VM（実験結果の図では「特化」と示す）の他に，特定のアプリケーションには特化させずに，一般的によく使われる合成命令を VM に実装する方法を模した「一般 1」と「一般 2」の VM でも実験した．「一般 1」は表 2 に 1 回以上現れる合成命令を全て実装しており，16 個の合成命令を持つ．「一般 2」は 2 回以上現れる合成命令を実装しており，表 2 でダガー（†）印を付けた 5 個の合成命令を持つ．これらの VM も，4 章で述べた各実装方法で生成した．

実行時間は，Raspberry Pi3 で実行して計測した．その他の実験環境は以下の通りである．

コンパイラ: gcc 8.2.0 (arm-linux-gnueabihf)
最適化オプション: -Os -marm
OS: Raspbian GNU/Linux 9

また，オペランド仕様には，全データ型をサポートする命令を生成するオペランド仕様 any と，add 命令には共に fixnum 型か共に string 型のオペランドだけを，それ以外の演算命令には fixnum 型のオペランドだけをサポートす

[*3] https://webkit.org/perf/sunspider/sunspider.html

表 2: 実装した合成命令
Table 2 Implemented superinstructions.

プログラム	合成命令	実行割合 (%)
3d-cube	addregfix†	5.35
	lessthanregfix†	1.05
access-binary-trees	subregfix†	2.44
	lessthanfixreg	1.66
	mulfixreg	1.63
access-fannkuch	addregfix	6.68
	rightshiftregfix†	1.01
access-nbody	addregfix	1.84
access-nsieve	addregfix	8.10
bitops-3bit-bits-in-byte	bitandregfix	9.99
	bitandfixreg	9.99
	leftshiftregfix†	6.66
	lessthanregfix	3.36
	addregfix	3.34
	rightshiftfixreg	3.33
	rightshiftregfix	3.33
bitops-bits-in-byte	lessthanregfix	10.42
	leftshiftregfix	8.33
	addregfix	5.21
bitops-bitwise-and-func	addregfix	9.09
controlflow-recursive	subregfix	6.38
	lessthanregfix	3.23
	equalregfix	2.34
math-cordic	addregfix	3.90
	lessthanregfix	3.90
	mulregfix	1.20
math-spectral-norm	addregfix	10.28
	divregfix	3.39
	divfixreg	3.39
string-fasta	addregfix	1.11
triple	addregstr	10.71
	addstrreg	3.57
	addregfix	3.57

るように指定したオペランド仕様 fixnum の二つを用いた．fixnum を用いて生成した VM は一部のアプリケーションに特化しているため，いくつかのベンチマークプログラムは実行することができない．

6.1 合成命令の効果

図 12 に合成命令を持たない VM での実行時間で正規化した各 VM でのベンチマークプログラムの実行時間を示す．ベンチマークプログラム名の後の any と fixnum はオペランド仕様を示している．

また，インタプリタのサイズを**図 13** に示す．点線は合成命令を持たない VM のインタプリタのサイズである．反例の実装 1 から実装 5 は，表 1 の実装番号に対応している．インタプリタのサイズには，組込み関数は含んでいない．これは，サイズに厳しい組込みシステムでは必要な組込み関数だけを選択して実装すると考えるためである．さらに，動的にリンクされる libc のサイズも含んでいない．現在の eJSVM の実装では文字列から数値に変換する関数などが実装できていないため libc を利用しているが，これらの関数は短かいコードで実現できる見通しがあるためである．

6.1.1 実行時間

アプリケーションに特化した合成命令を実装した VM でオペランド仕様に any を用いた場合， 実行時間はベンチマークプログラムによって 2 種類の結果になった．bitops-3bit-bits-in-byte と bitops-bits-in-byte は合成命令の追加によって，実行時間が比較的大きく短縮された．例えば，型ディスパッチを共有する実装（改良版）では 10%程度実行時間が短縮された．これらは，合成命令の実行頻度が特に高かったベンチマークプログラムである（表 2 参照）．他のベンチマークプログラムでも実行時間は短縮されたが，その程度は小さく，実装によっては僅かに遅くなったプログラムもあった． eJS の想定されている利用方法である，オペランド仕様に fixnum を用いて命令が受けとるデータ型を制限した VM も同じ傾向を示しているが，オペランドのデータ型を制限しない VM よりも効果が高かった．

6.1.2 インタプリタサイズ

インタプリタのサイズは増加したが，その程度は実装した合成命令の数や実装方法によって異なる．ベンチマークプログラムに特化した合成命令を持つ VM の中で最も多くの合成命令を実装した bitops-3bit-bits-in-byte では，オペランド仕様に any を用い場合，1KB から 4KB の増加になった．

6.1.3 アプリケーションに特化した合成命令の得失

一般的によく使われる合成命令を実装した VM（「一般 1」と「一般 2」）でも合成命令を持たない VM に比べて高速化されたが，アプリケーションに特化した合成命令を持つ VM はより高速であり，サイズの増加も小さい傾向にあった．一方で，アプリケーションに特化させるためには，アプリケーションのプロファイルをとり，実装する合成命令を決めたり，アプリケーション毎に VM を生成する必要があり手間がかかる．

しかし，eJS で想定しているアプリケーション開発プロセスでは，既にこの手間はかかっている．eJSTK はアプリケーション毎に各命令がサポートするデータ型を特化させた VM を生成する．そのために，アプリケーションの開発プロセスでアプリケーションのプロファイリングも想定されている．実装する合成命令をアプリケーションに特化させる作業もこの一環として行うことができる．

6.2 合成命令の実現方法の比較

合成命令の実装方法を詳しく比較するために，bitops-3bit-bits-in-byte について，合成命令を実行頻度が高かった

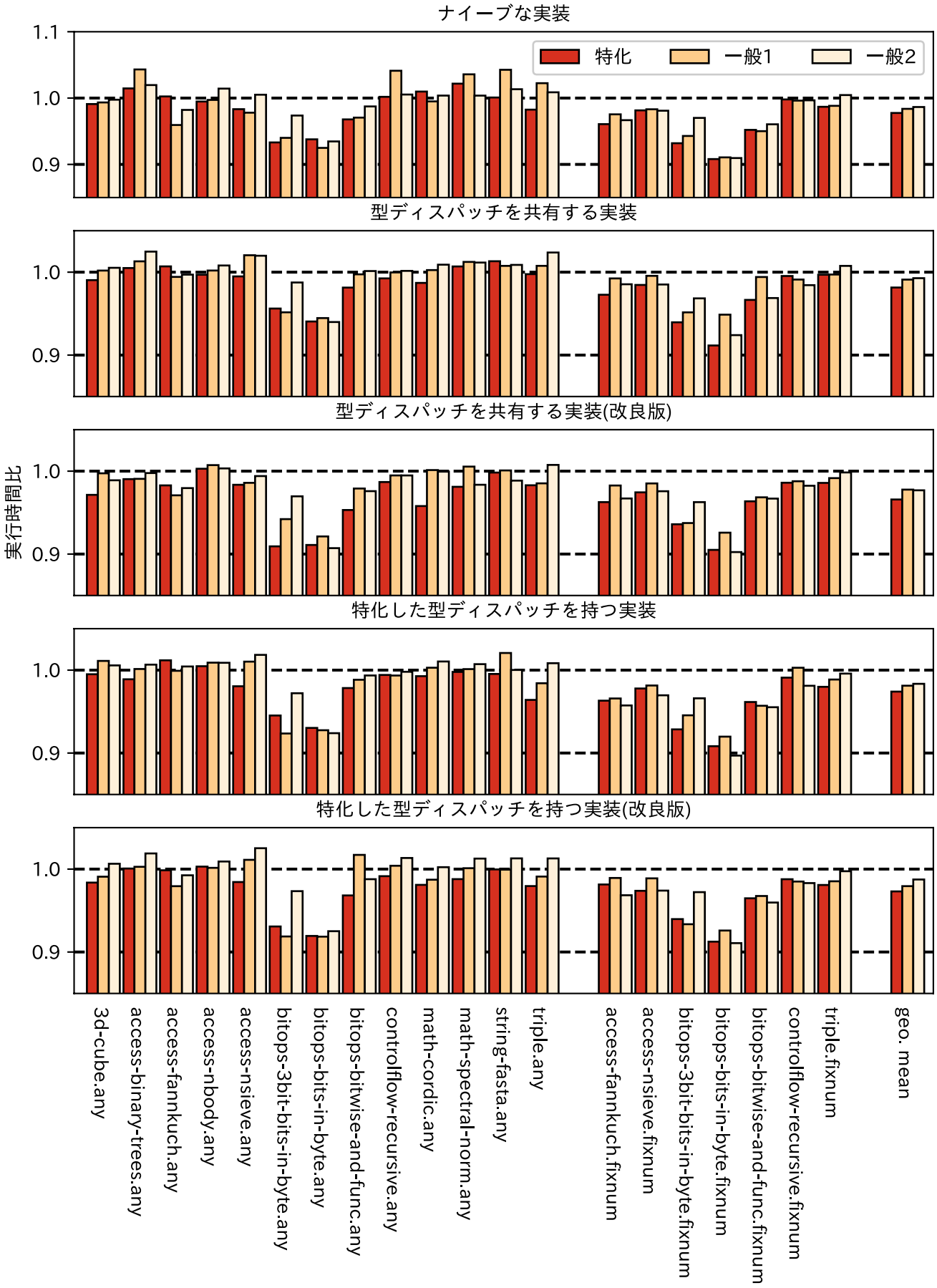


図 12: 標準化した実行時間

Fig. 12 Normalized execution times.

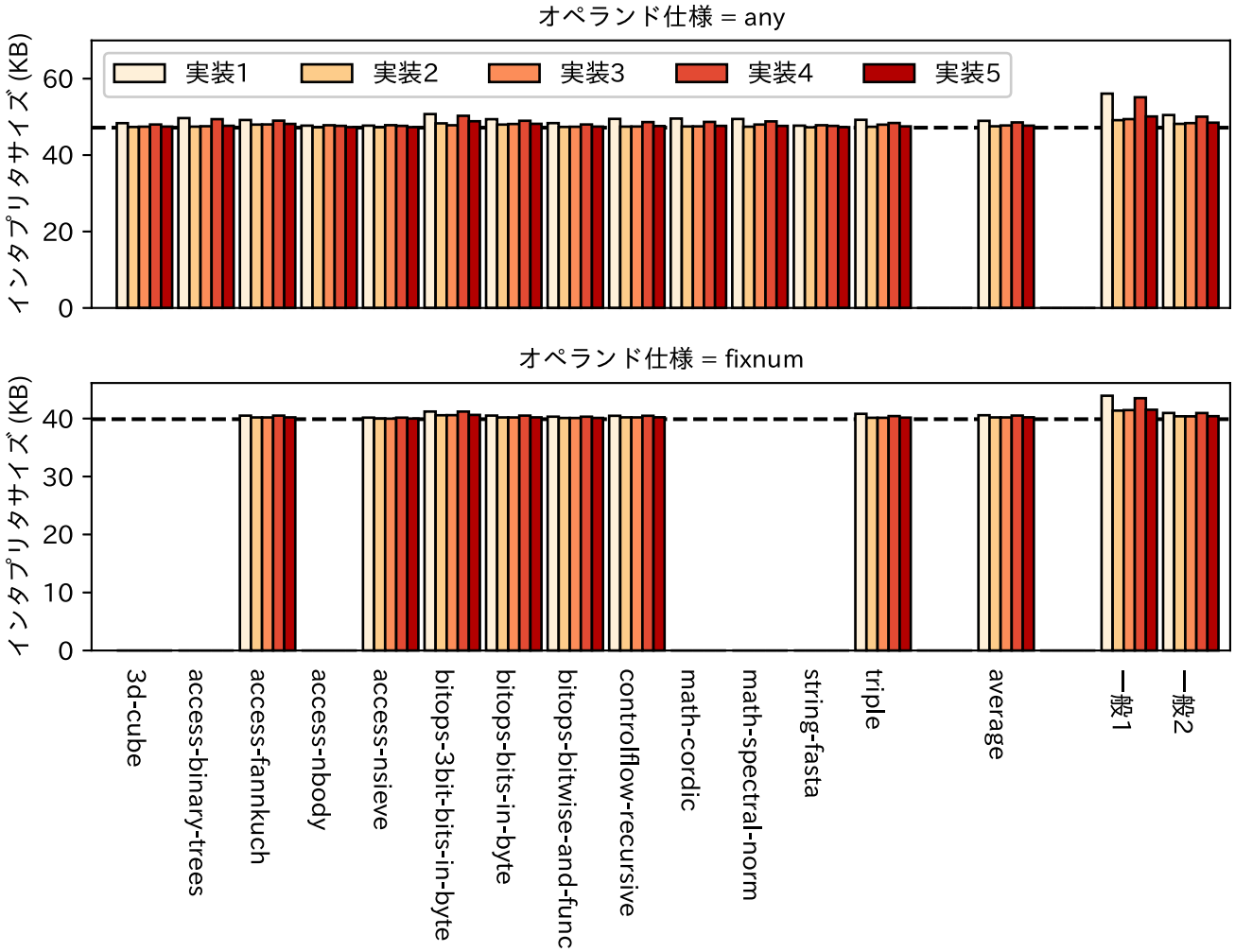


図 13: インタプリタのサイズ （点線は合成命令を持たないインタプリタのサイズ）

Fig. 13 Interpreter sizes. Dashed line indicates the size of the interpreter that has no superinstructions.

順に 7 個まで 1 個ずつ追加して，実行時間とインタプリタのサイズを図 **14** にプロットした． 各折線の左上に近い端点が合成命令を 1 個追加した場合で，折線の順に合成命令を追加している．実行時間は合成命令を持たない VM の実行時間で正規化しており，インタプリタのサイズは合成命令を持たない VM からの増分を示している．

どの実現方法でも，合成命令の数を増やすと実行時間は短縮され，インタプリタのサイズは増加する傾向にあるものの，実現方法によってその程度は大きく異なっている．また，合成命令の数に対して単調に変化しないこともあった．これは，コードの変化が VM をコンパイルした gcc の最適化に影響したためと推測している．

6.2.1 コードの共有によるインタプリタサイズ増加の抑制

オペランド仕様に any を用いた場合，ナイーブな実装と特化した型ディスパッチを持つ実装ではインタプリタサイズの増加が大きい．これは型ディスパッチと演算処理を複製しているためである．ただし，特化した型ディスパッチを持つ実装では，合成命令の型ディスパッチが最適化されているため，ナイーブな実装と比べるとサイズの増加は抑えられている．型ディスパッチを持つ実装（改良版）は，演算処理を共有しており，その分だけサイズの増加は改良前に比べて抑えられた．型ディスパッチを共有する実装やその改良版のサイズは合成命令の数に対して単調に変化していないが，傾向としては他の実装方法よりサイズの増加が抑えられていた．

オペランド仕様に fixnum を用いた場合，合成元命令でもオペランドのデータ型が制限されていることを利用していい型ディスパッチを最適化している．これにより型ディスパッチは十分に小さくなっている．そのため，ナイーブな実装と特化した型ディスパッチを持つ実装ではインタプリタのサイズに差がない．また，特化した型ディスパッチを持つ実装（改良版）も型ディスパッチを共有する実装と差がない．結果として，演算処理を共有するグループと共有しないグループの 2 グループに分かれた．

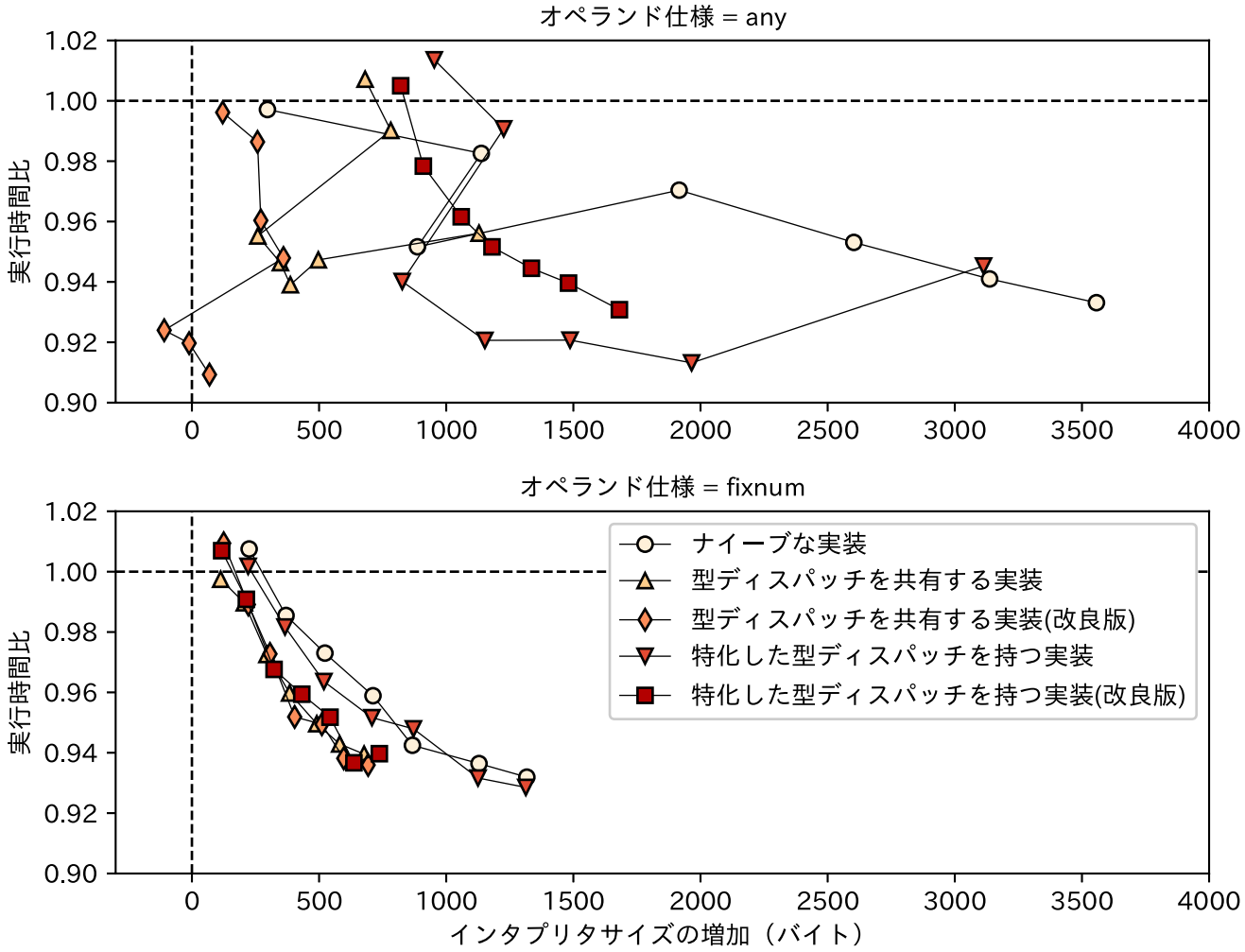


図 14: bitops-3bit-bits-in-byte の実行時間とインタプリタサイズ
Fig. 14 Execution times for bitops-3bit-bits-in-byte and interpreter sizes.

6.2.2 型ディスパッチの最適化による実行の高速化

オペランド仕様に any を用いた場合，何からの方法で型ディスパッチのコストを減らした実装が，それ以外の実装（ナイーブな実装と型ディスパッチを共有する実装）よりも高速だった．特に，特化した型ディスパッチを持つ実装の合成命令を 7 個実装したときの実行時間を例外と考えれば，特化した型ディスパッチを持つ実装と型ディスパッチを共有する実装（改良版）が高速だったと言える．なお，特化した型ディスパッチを持つ実装（改良版）は予想に反して改良前と比べて遅かったが，ベンチマークプログラムによっては高速になることもあった（図 12 参照）．

オペランド仕様に fixnum を用いた場合は，型ディスパッチの最適化の有無とは関係なく，演算処理を共有するグループの方が共有しないグループより高速だった．オペランド仕様を利用して型ディスパッチは十分最適化されていたためである．

6.2.3 命令ディスパッチの削減による高速化

ナイーブな実装や型ディスパッチを共有する実装でも，ある程度の高速化は得られた．これは命令ディスパッチの回数と，レジスタを実現するメモリへのアクセスが削減されたためと考えられる．

6.2.4 その他の実行速度に関係する要素

オペランド仕様に fixnum を用いた場合は，演算処理を共有すると共有しない場合より僅かに遅かった．これらの実装の差異は，合成命令から共有された演算処理への goto 文の有無である．

7. 関連研究

合成命令はよく知られた VM の高速化手法である．Probsting [3] は，C 言語のインタプリタの高速化のために合成命令を導入した．Probsting は，プリミティブな命令の仕様と，どの命令を合成するかを指定したリストから合成命令を持つインタプリタを生成する．Ertl らの Vmgen [6] も同様に，プリミティブな命令のテンプレートと合成命令の仕様から，合成命令を持つ VM を生成する．Vmgen は組込みシステム用の Java VM の開発に用いられている [7], [8].

本研究では，これらの研究と同様に eJSVM に合成命令を導入した．しかし，Probsting のインタプリタや Vmgen が生成する VM は，中間コードで変数の型が決まることを前提としており，VM 生成系が型ディスパッチの最適化を行う必要はない．本研究では型ディスパッチの最適化を行う VM 生成系である eJSTK に合成命令を導入し，定数オペランドの型が決まることを利用した最適化を行った．

デスクトップ計算機上で動作する VM では，動的に複数の命令を合成するのが一般的である．Piumarta ら [9] は，動的に命令を合成することで，RISC 命令セットのインタプリタの性能を大幅に向上した．近年では JIT コンパイルが主流になっている．これらの方法では，VM 命令を合成するとそれを実装する CPU のコードを最適化する余地が増えるので，実行時に最適化する仕組みを持っている．それに対して，組込みシステムを対象とする本研究では，実行前に最適化する．また，本研究では型ディスパッチのコストを削減する最適化に特化している点も異なる．言語に特化した最適化には，Zakirov ら [10] の Ruby VM に合成命令の導入による浮動小数点数の box 化を減らす最適化がある．

本研究では定数ロード命令と演算命令を合成し，オペランドに定数を指定できる演算命令を作った．しかし，もともとオペランドに定数を指定できる CISC 命令セットの VM も考えられる．JerryScript はそのような命令セットを持った組込みシステム向けの JavaScript VM である．JerryScript の命令インタプリタのループ内部は 3 ステップに分かれている．第 1 ステップで命令デコードとオペランドのロードを行う．ロードされたオペランドはインタプリタのローカル変数に格納される．第 2 ステップで命令ディスパッチと命令に対応する計算を行う．第 3 ステップで計算結果を適切な位置に格納する．第 1 ステップや第 3 ステップでは，スタック上のスロットやレジスタなど様々なロケーションにアクセスする可能性があり，複雑なコードになっているが，全ての命令でコードを共有している．本研究でも同じ方式を試作したが，eJS では逆に実行速度が低下してしまった．

本研究では合成命令と合成元命令で，型ディスパッチや演算処理のコードを共有することでインタプリタのサイズが増加するのを抑えた．Peng ら [11] はスタックベースのインタプリタにスタックキャッシングを導入する際に，命令が複製されてインタプリタのサイズが増加するのを抑える目的で，VM 命令を実装するコードの一部を共有する手法を提案している．

8. まとめ

本論文では組込みシステム向けの JavaScript 処理系 eJS に対して，定数ロード命令と演算命令を合成した合成命令を導入した．合成命令の実現方法には，合成命令と合成元命令で実装のコードを共有したり，定数オペランドのデータ型を利用した型ディスパッチの最適化を行う方法の違いにより 5 種類の実装方法を試した． その結果，どの実装方法でも合成命令の導入により高速化が達成された．高速化の程度や，インタプリタのサイズの増加の程度は実装方法により異なった．

謝辞 本研究に協力して頂いた eJS プロジェクトのメンバーや，有益なコメントを頂いた査読者に感謝します．本研究の一部は，JSPS 科研費 16K00103 の助成を受けたものです．

参考文献

[1] Kataoka, T., Ugawa, T. and Iwasaki, H.: A Framework for Constructing JavasSript Virtual Machines with Customized Datatype Representations, *In Proc. SAC 2018*, ACM, pp. 1238–1247 (2018).

[2] Ugawa, T., Iwasaki, H. and Kataoka, T.: eJSTK: Building JavaScript virtual machines with customized datatypes for embedded systems, *Journal of Visual Languages and Computing* (to appear).

[3] Proebsting, T. A.: Optimizing an ANSI C Interpreter with Superoperators, *In Proc. POPL 1995*, ACM, pp. 322–332 (1995).

[4] ECMA International: *Standard ECMA-262 - ECMAScript Language Specification*, 5.1 edition (2011).

[5] Bell, J. R.: Threaded Code, *Communications of ACM*, Vol. 16, No. 6, pp. 370–372 (1973).

[6] Ertl, M. A., Gregg, D., Krall, A. and Paysan, B.: Vmgen - a generator of efficient virtual machine interpreters, *Software: Practice and Experience*, Vol. 32, No. 3, pp. 265–294 (2002).

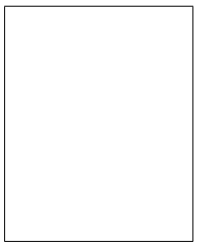
[7] Beatty, A., Casey, K., Gregg, D. and Nisbet, A.: An Optimized Java Interpreter for Connected Devices and Embedded Systems, *In Proc. SAC 2003*, ACM, pp. 692–697 (2003).

[8] Ertl, M. A., Thalinger, C. and Krall, A.: Superinstructions and replication in the Cacao JVM interpreter, *Journal of .NET Technologies*, Vol. 4, pp. 25–32 (2006).

[9] Piumarta, I., Riccardi, F. and Rocquencourt, I.: Optimizing Direct Threaded Code By Selective Inlining, *In Proc. PLDI 1998*, ACM, pp. 291–300 (1998).

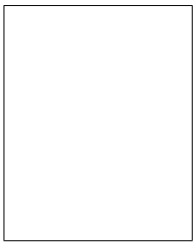
[10] Zakirov, S., Chiba, S. and Shibayama, E.: How to Select Superinstructions for Ruby, *IPSJ Online Transactions*, Vol. 3, pp. 54–61 (online), DOI: 10.2197/ipsjtrans.3.54 (2010).

[11] Peng, J., Wu, G. and Lueh, G.-Y.: Code Sharing among States for Stack-Caching Interpreter, *In Proc. IVME 2004*, ACM, pp. 15–22 (2004).



野中 智矢 （正会員）

1996 年生．2018 年高知工科大学情報学群卒業．



鵜川 始陽 （正会員）

1978 年生．2000 年京都大学工学部情報学科卒業．2002 年同大学大学院情報学研究科通信情報システム専攻修士課程修了．2005 年同専攻博士後期課程修了．同年京都大学大学院情報学研究科特任助手．2008 年電気通信大学助教．2014 年より高知工科大学准教授．博士 (情報学)．プログラミング言語とその処理系に関する研究に従事．情報処理学会 2012 年度山下記念研究賞受賞．