Ruby用仮想マシン YARV の実装と評価

SWoPP 2005（2005 8/3）

笹田耕一（発表者）
（農工大/日本Rubyの会）
まつもとゆきひろ（NaCl）
前田敦司（筑波大）
並木美太郎（農工大）

Agenda

- 背景
- Ruby処理系の課題
- YARVの設計
  - 最適化手法
  - 簡単なVM生成系の紹介
- YARVの評価
- まとめと今後の課題

背景：スクリプト言語とRuby

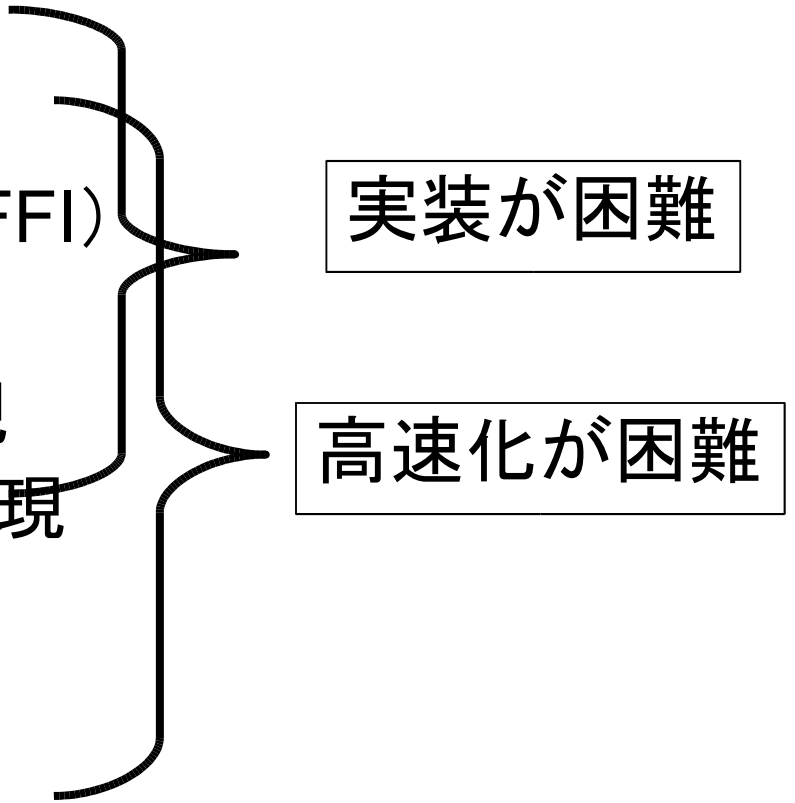
- スクリプト言語の利用シーンの増加
  - Perl, Python, PHP, Groovy, …
  - プログラミングのしやすさ
  - マシン性能の向上
- オブジェクト指向スクリプト言語Ruby
  - 利用の容易さから高い評価
    - 世界中で広く利用
    - とくに海外でブレイク中（Ruby on Rails など）
  - 日本発（主開発者：まつもとゆきひろ（共著者））

背景：現状のRubyの問題点

- 処理速度が遅い
  - スクリプト言語には速度は要らない
    → 用途が多様化　性能が必要な局面も
  - 構文木を再帰関数でたどる処理系
    → あまり最適化できない、例外処理は setjmp/longjmp
- その他のRuby VM
  - 多数提案されているが、高性能 ”Ruby” VM は皆無
    - まつもとVM、ByteCodeRuby、…
- その他の言語用VM
  - Ruby とのセマンティックスギャップ

Ruby処理系の課題

- パーサ作成が困難
- スレッドのサポート
- Ruby C API のサポート(FFI)
- 例外処理機能の実現
- ブロック・クロージャの実現
- オブジェクト指向機能の実現
- プリミティブ型が無い
- 静的解析が困難



本研究の目的

- Ruby 処理系の開発
  - 現在の Ruby 処理系を置き換え可能なレベル
    - Ruby “もどき” 処理系の開発ではない
  - 高い実行性能
  - 高いポータビリティ
- 処理系の開発効率の向上

Ruby向け仮想機械の開発

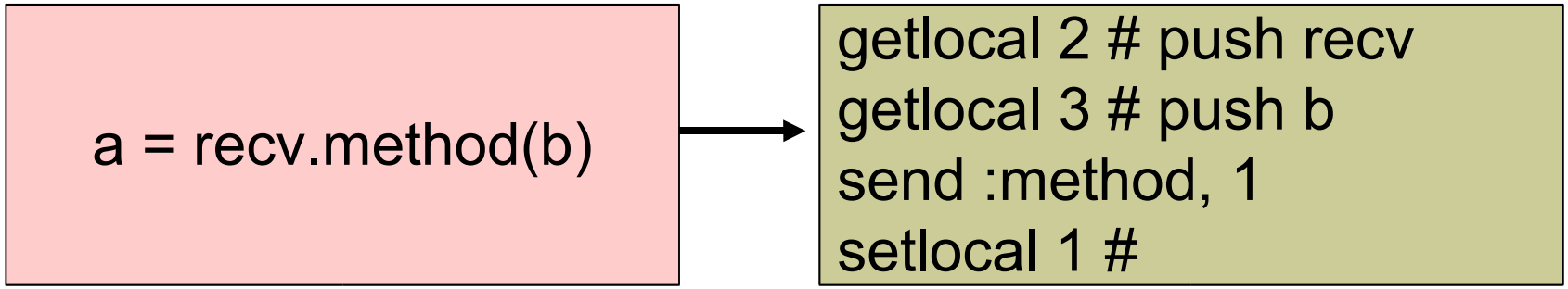
- YARV: Yet Another RubyVM
  - Rubyプログラムを表現するYARV命令セット
  - コンパイラ
  - 仮想機械（VM）
    - スタックマシン
    - 各種最適化を適用
  - C言語で実装
- 簡単なVM生成系を利用した効率的な開発
- オープンソースソフトウェアとして公開中
  - http://www.atdot.net/yarv/

Ruby処理系の課題の対策

- パーサ作成が困難　→ 既存のものを利用
- スレッドのサポート →　今後の課題
- Ruby C API のサポート（FFI） → サポート
- 例外処理機能の実現 → 作成（表・大域ジャンプ併用）
- ブロック・クロージャの実現 → 環境は出来るだけスタックに
- オブジェクト指向機能の実現 → インラインメソッドキャッシュ
- プリミティブ型が無い → 特化命令
- 静的解析が困難 → 他で性能を稼ぐ

設計：YARV命令セット

- 現時点では 52 命令
  - できるだけシンプルに
    → VM生成系によって自動的に複雑・高速な命令に
- プリミティブ型がないため、数値演算命令無し



設計：コンパイラ

- Rubyパーサ
  - Rubyプログラム → 構文木
  - 既存のものを利用
- コンパイラ
  - 構文木 → YARV 命令列
  - 後に述べる最適化用命令に変換

設計：VM

- 利用できる既存の仕組みはそのまま利用
  - オブジェクト管理・GC
- VM関数で命令を繰り返し実行
- スタックマシン
  - 値用スタック
  - 制御フレーム用スタック
  - スコープ変更 → 制御フレーム用スタック push
    - メソッド呼び出し
    - ブロック呼び出し（yield）
    - クラス定義文

設計 : 例外処理

- Rubyレベルでは表引きで処理
  - 例外処理部分突入時にはコスト無し
  - Ruby スタックの巻き戻しを行い探索
- setjmp/longjmp による例外処理機構と併用
  - VM関数のネストを自然に表現可能
  - マシンスタックの巻き戻しを実現

設計 : 例外処理

YARV実行のC関数コールグラフ

VM handler 関数(setjmp)
VM関数
C関数
VM handler 関数(setjmp)
VM 関数
C関数

表を引いて例外ハンドラ探索

見つからなければ longjmp

例外を発生するには longjmp

最適化

- コンパイル時最適化
- ダイレクトスレッデッドコード
- ☆特化命令
- ☆オペランド・命令融合
- インライン（メソッド）キャッシュ
- ☆静的スタックキャッシング
- プロファイラ
- ネイティブコンパイラ（不完全）

最適化：特化命令

- 特定 selector、特定の引数の数のメソッド呼び出しを特定の命令で置き換え
  - 例）a + b
    - send :+,1　　→　　opt_plus
- 型（クラス）のチェック、再定義のチェックを行い、可能ならばその処理を行う

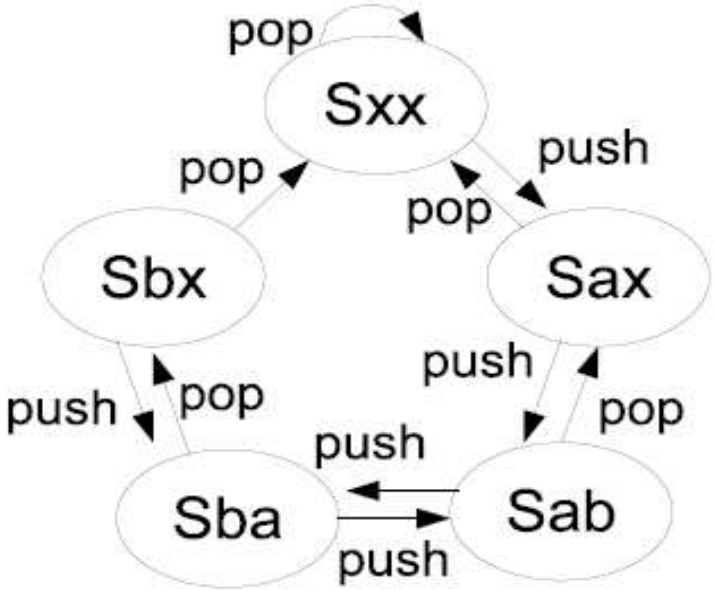
```
opt_plus:
      if(a と b は整数？)
            if 整数についての + メソッドは再定義されていないか？
                  return a+b
            else return 通常のメソッド呼び出し（a.+(b)）
```

最適化：オペランド・命令融合

- オペランド融合
  - insnA op1 → insnA_op1
  - op1 が頻出する場合、ひとつの命令に融合
  - オペランドフェッチコスト減・命令の部分評価
- 命令融合
  - insnA insnB → insnA_insnB
  - insnA, insnB という並びが頻出する場合、ひとつの命令に融合（a.k.a superinstruction）
  - 命令ディスパッチコスト減・Cコンパイラの最適化

最適化：静的スタックキャッシング

- 5状態、2レジスタの静的スタックキャッシング（SC）
  - スタックトップの2値をレジスタに格納
  - 5状態なので、同じ命令につき5命令必要
  - putobject（スタックにひとつ積む）
    - putobject_xx_ax
    - putobject_ax_ab
    - putobject_bx_ba
    - putobject_ab_ba
    - putobject_ba_ab



VM生成系

- VMの記述は煩雑
  - 各命令ごとのスタックのpush/pop, pc 増加, etc
  - オペランド・命令融合のための命令の用意
  - スタックキャッシュ用命令の用意（基本の5倍必要）
- VM記述から各命令の自動生成
  - 共通の処理は手で書く必要なし
  - 最適化のための新しい命令はすべて自動で生成
    - なにをどう融合するか、などの指示を与えて生成
- 関連：vmgen など

VM生成系：命令記述と生成例

- 値の宣言と、ロジックの記述（C言語）
  - （1）命令オペランド（2）スタックオペランド
    （3）スタックへプッシュする値

```
DEFINE_INSN
mult_plusConst
(int c)        // 命令オペランド定義
(int x, int y) // スタックオペランド定義
(int ans)      // 命令の返り値定義
{
  // C言語による命令ロジック記述部分
  ans = x * y + c;
}
```

→

```
mult_plusConst:{
  int c = *(PC+1);
  int y = *(SP-1);
  int x = *(SP-2);
  int ans;
  PC += 2;
  SP -= 2;
{
  ans = x * y + c;
}
  *(SP) = ans; SP += 1;
  goto **PC;}
```

命令オペランド
フェッチ

スタックオペランド
フェッチ

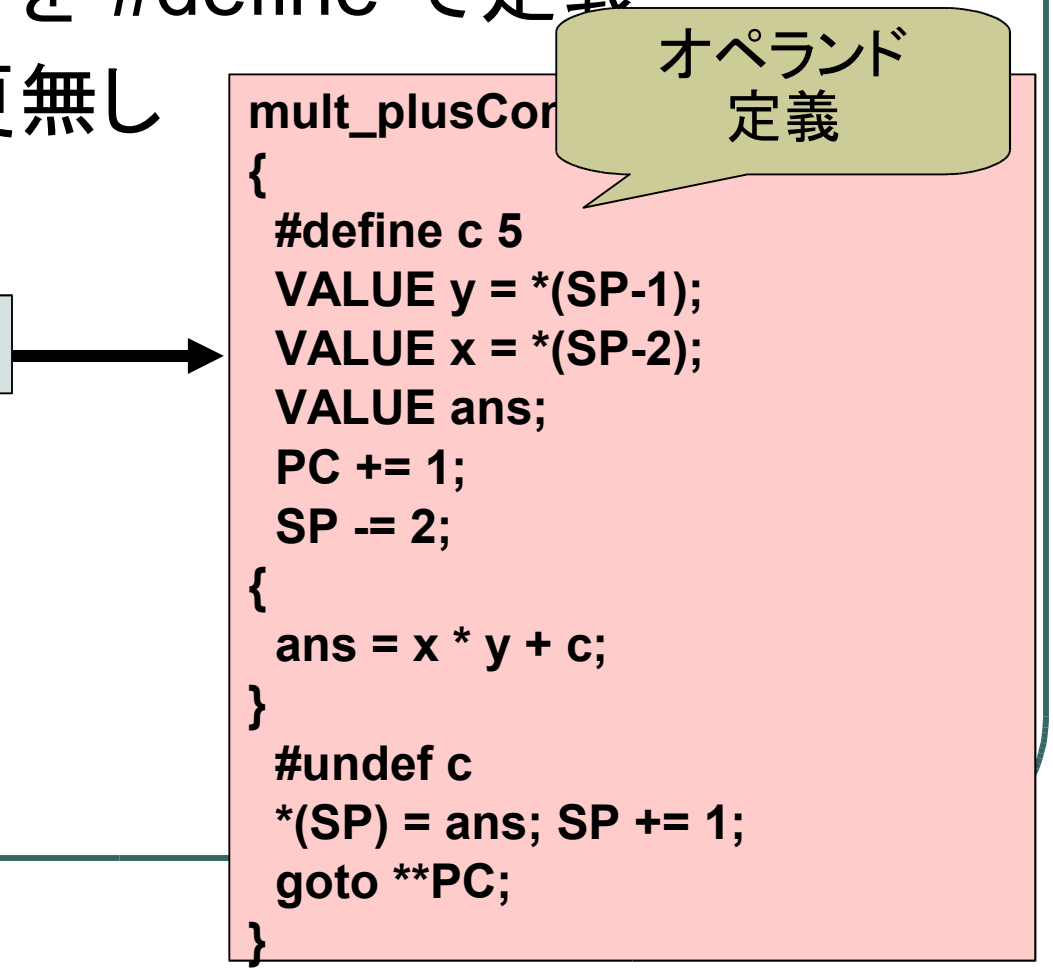
PC/SP
設定

結果を
push

VM生成系:オペランド融合命令の生成

- 融合命令オペランドを #define で定義
- ロジック部分は変更無し

mult_plusConst 5



VM生成系：命令融合した命令の生成

dup + mult_plusConst

```
UNIFIED_dup_mult_plusConst:
{
  int c_1 = *(PC+1);
  int v_0 = *(SC-1);
  int ans;
  PC += 3;
  SP -= 1;
{ // dup
  #define v v_0
  #define v1 v1_0
  #define v2 v2_0
  v1 = v2 = v;
  #undef v
  #undef v1
  #undef v2
} // cont ->
```

利用する値を適切に定義

命令を連結

```
// -> cont
{ // mult_plusConst
  #define c c_1
  #define x v1_0
  #define y v2_0
  ans = x * y + c;
  #undef c
  #undef x
  #undef y
}
  *(SP) = ans; SP += 1;
  goto **PC;
}
```

VM生成系：スタックキャッシング用命令生成

- プッシュ・ポップ処理を SC用レジスタアクセスに
- コンパイラは命令列を状態遷移に則り変換

mult_plusConst_ab_ax

```
mult_plusCon
{
  int c = *(PC+1);
  int y = SC_regB;
  int x = SC_regA;
  int ans;
  PC += 2;
{
  ans = x * y + c;
}
  SC_regA = ans;
  goto **PC;
}
```

SCレジスタアクセス（pop）

SCレジスタアクセス（push）

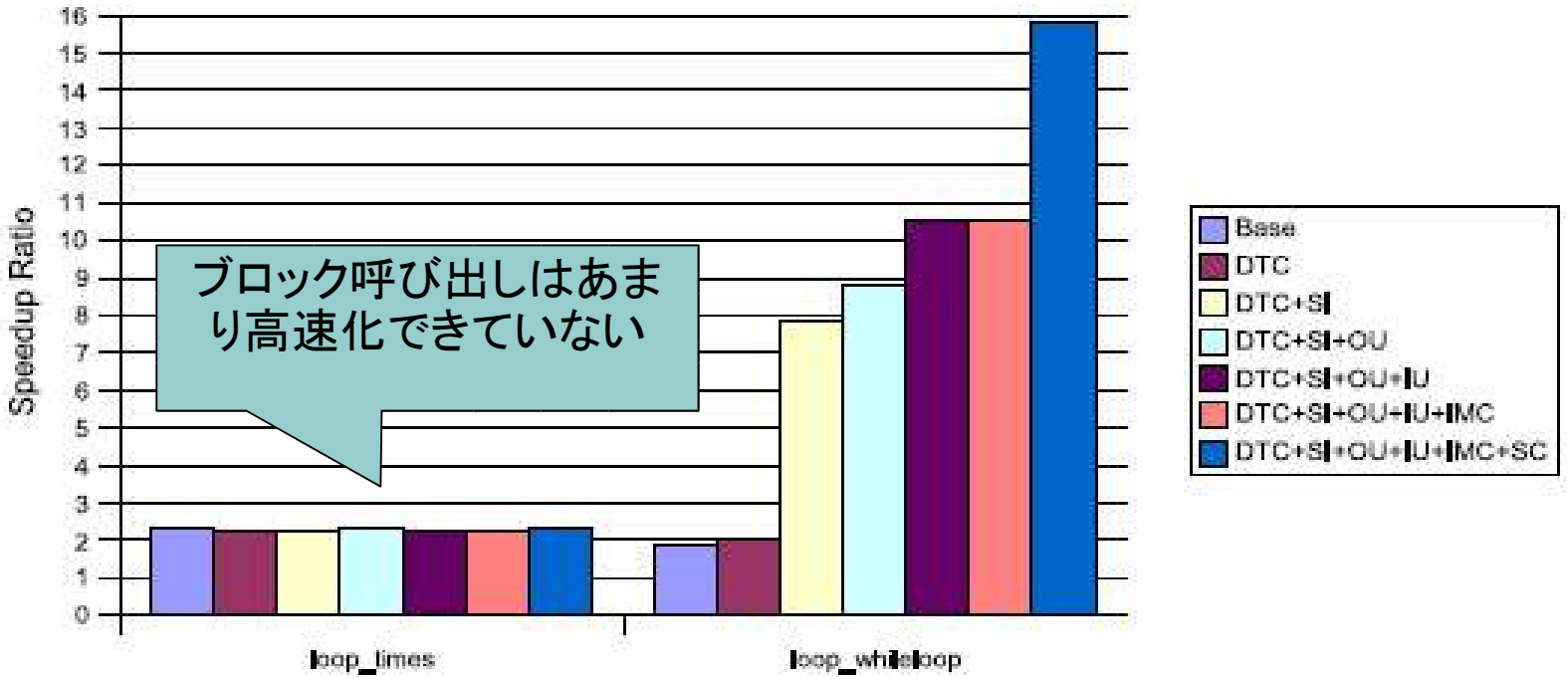
評価：評価環境

- 評価環境

  （1）Pentium-M 753（1.2GHz，L1I/D 32KB，L2S 2MB）、メモリ1GB，WindowsXP + cygwin，GCC 3.4.4

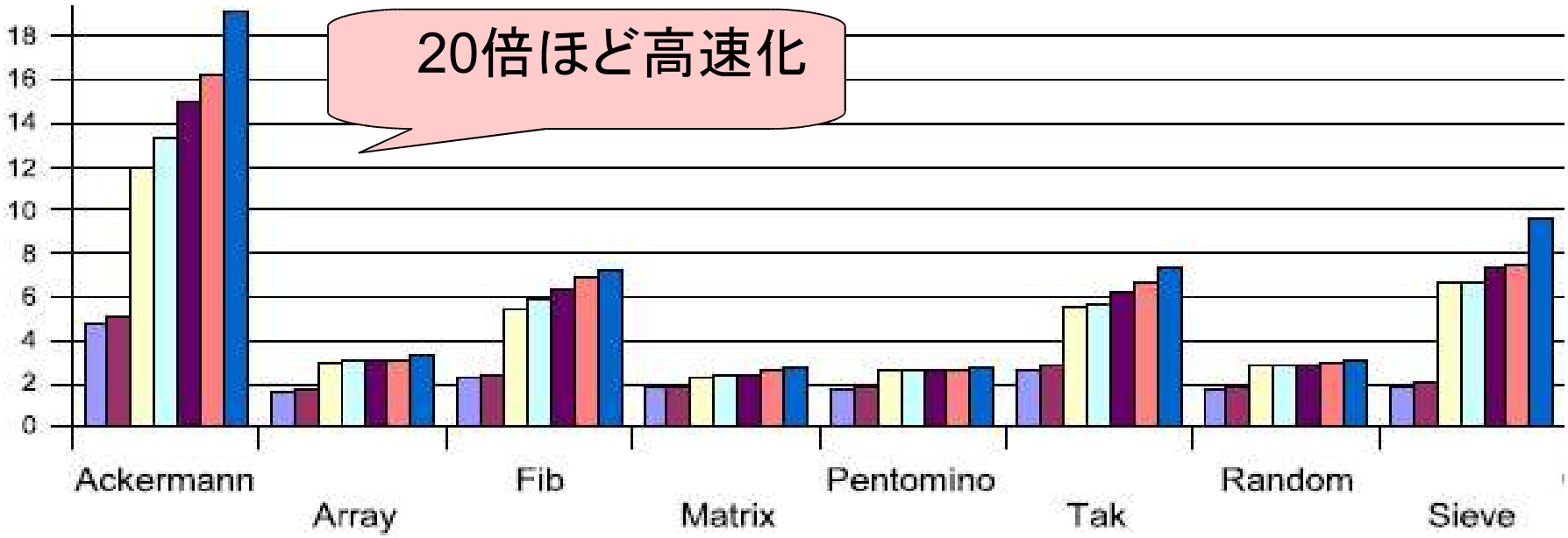
- 比較言語処理系（すべて（1）上）
  - Ruby 1.9.0 (2005-03-04)
  - Perl 5.8.6
  - Python 2.4.1
  - Gauche 0.8.4（Scheme 処理系）

評価:繰り返し

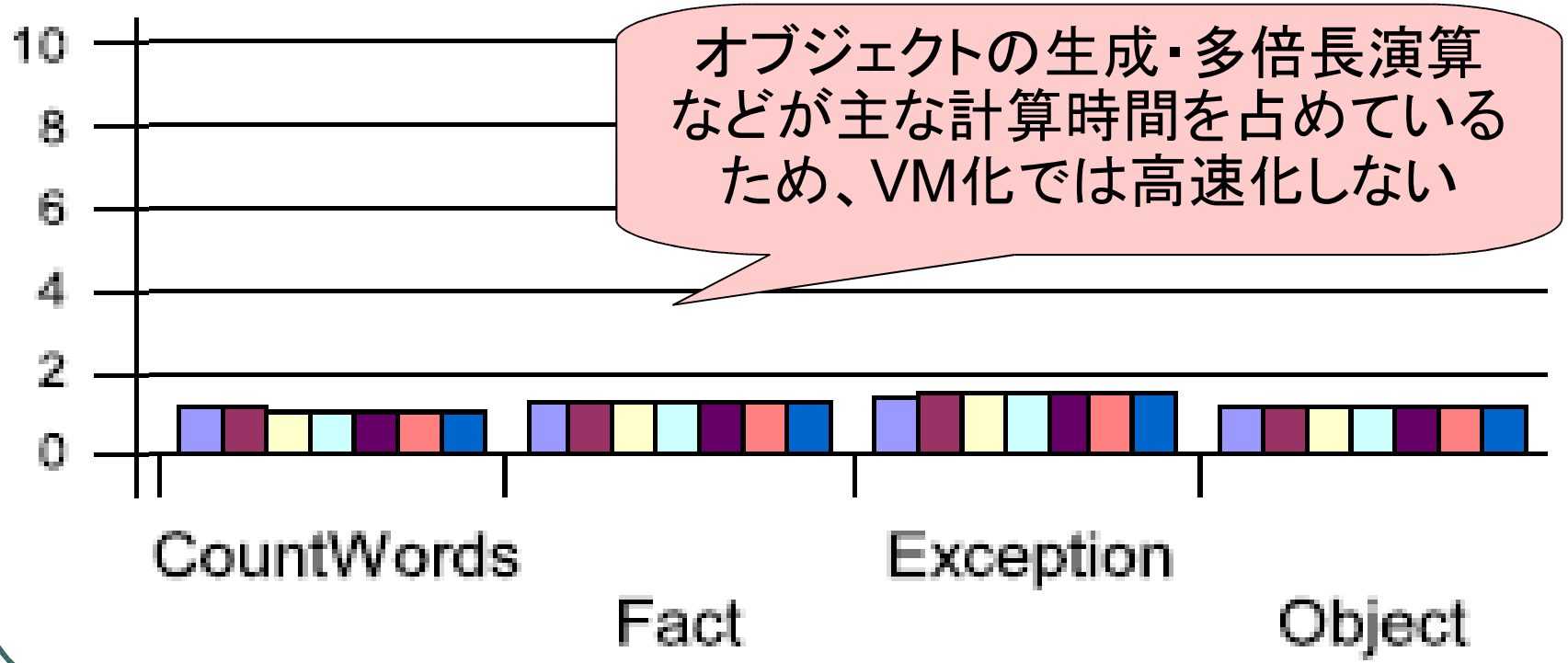


Base: VM化しただけのもの　**OU**: Operand Unification　**IMC**: Inline Method Cache
DTC: Direct Threaded Code　**IU**: Instruction Unification　**SC**: Stack Caching
SI: Specialized Instruction

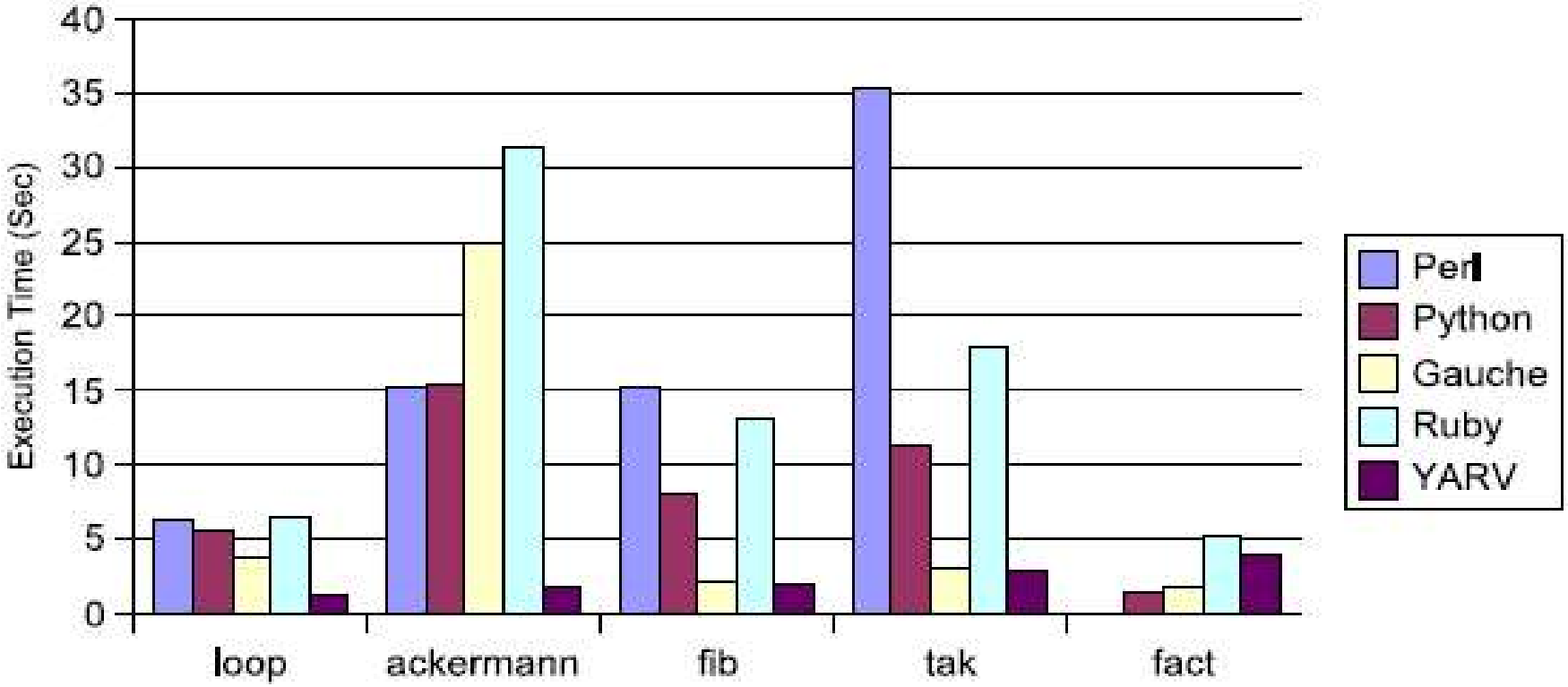
評価: 速くなった例


20倍ほど高速化
18
16
14
12
10
8
6
4
2
0
Ackermann
Array
Fib
Matrix
Pentomino
Tak
Random
Sieve

評価：あまり変わらない例



評価：他言語との比較


Execution Time (Sec)
40
35
30
25
20
15
10
5
0
loop
ackermann
fib
tak
fact
Perl
Python
Gauche
Ruby
YARV

まとめ

- Ruby向け仮想機械YARVを開発
  - Rubyを動かすための処理系をデザインし実装
    - C言語で実装
  - いろいろな最適化手法を適用
    - 最大24倍高速化（ackermann関数）
    - 他言語処理系と比較しても高速
  - VM生成系を利用して容易に最適化を適用
    - 単純な命令から融合操作・SC命令の半自動生成

今後の課題

- より大規模なRubyアプリケーションでの評価
- Rubyとのマージ
- ブロック付きメソッド呼び出しの高速化
- Rubyのネイティブスレッド対応
  - スレッドローカルGCなど
- RubyのMulti-VM インスタンス対応

おわり

ご清聴ありがとうございました。

ささだ　こういち
ko1@atdot.net

謝辞

yarv-dev/yarv-devel ML 参加者の方々には
いつも有益なアドバイスを頂いております。感謝いたします。

本プロジェクトは IPA 情報処理推進機構
未踏（2005）・未踏ユース（2004）の支援を受けています。

1.5分でわかる Ruby

```
recv.method(arg)
```

```
class Foo
  def
  end
end
class Foo
  # ここは実行文
  end
end
```

```
# 例外
begin
  #
end
```

```
eval(s) # evil
1+2 # 1.+(2)
```

```
# ブロック付
# メソッド呼び出し
# lexical closureも
# 作成可能
```

```
; scheme なら
(iter
  (lambda ()
    ...))
```

動的 ○プログラミングはしやすい
×静的解析が困難

任意の時点で
eval 可能

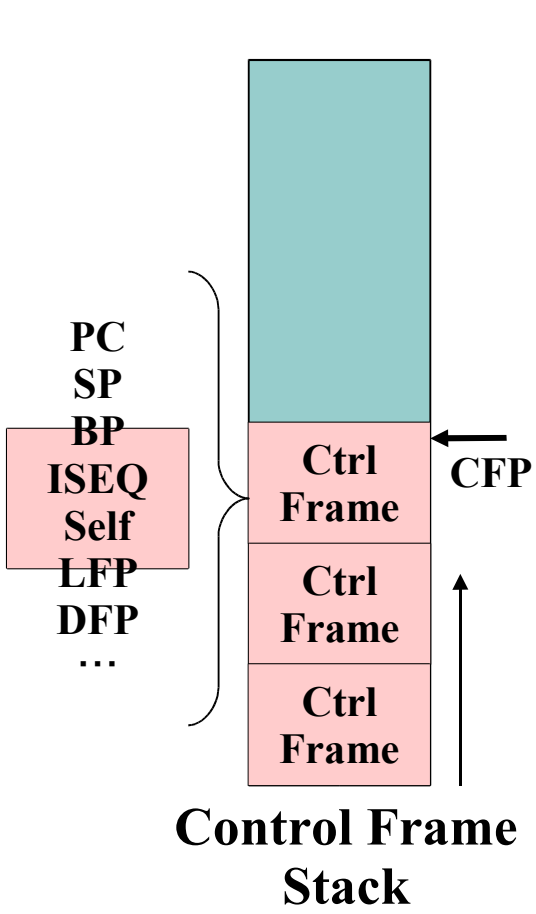
メソッドは
再定義可能

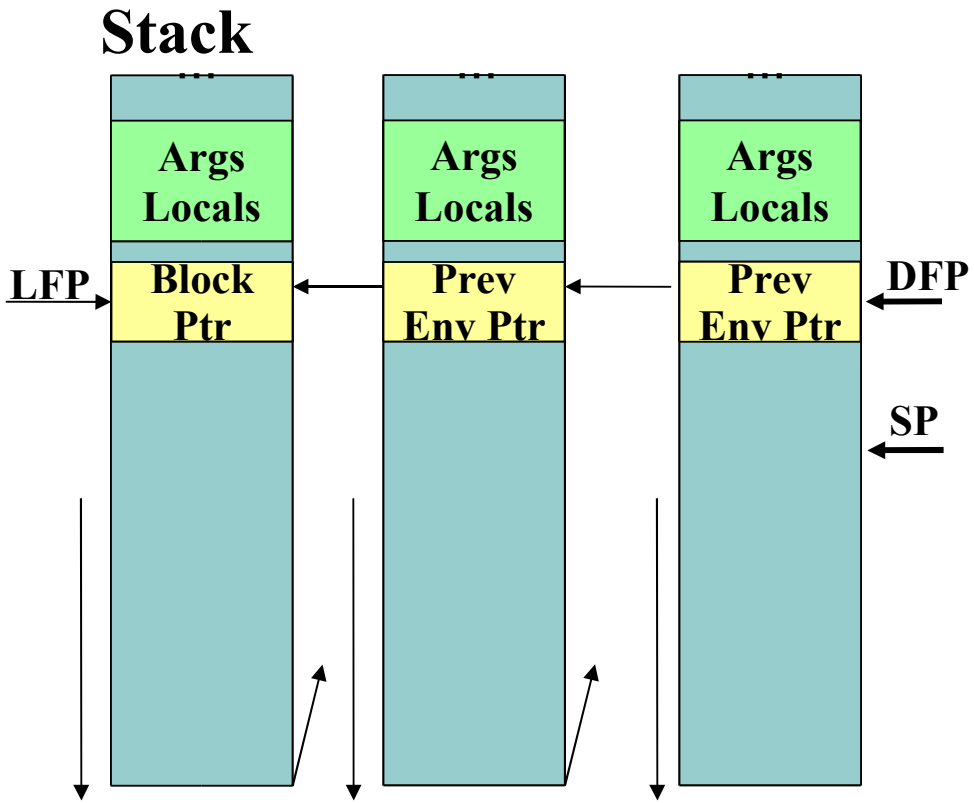
演算子も
すべてメソッド

設計: Registers

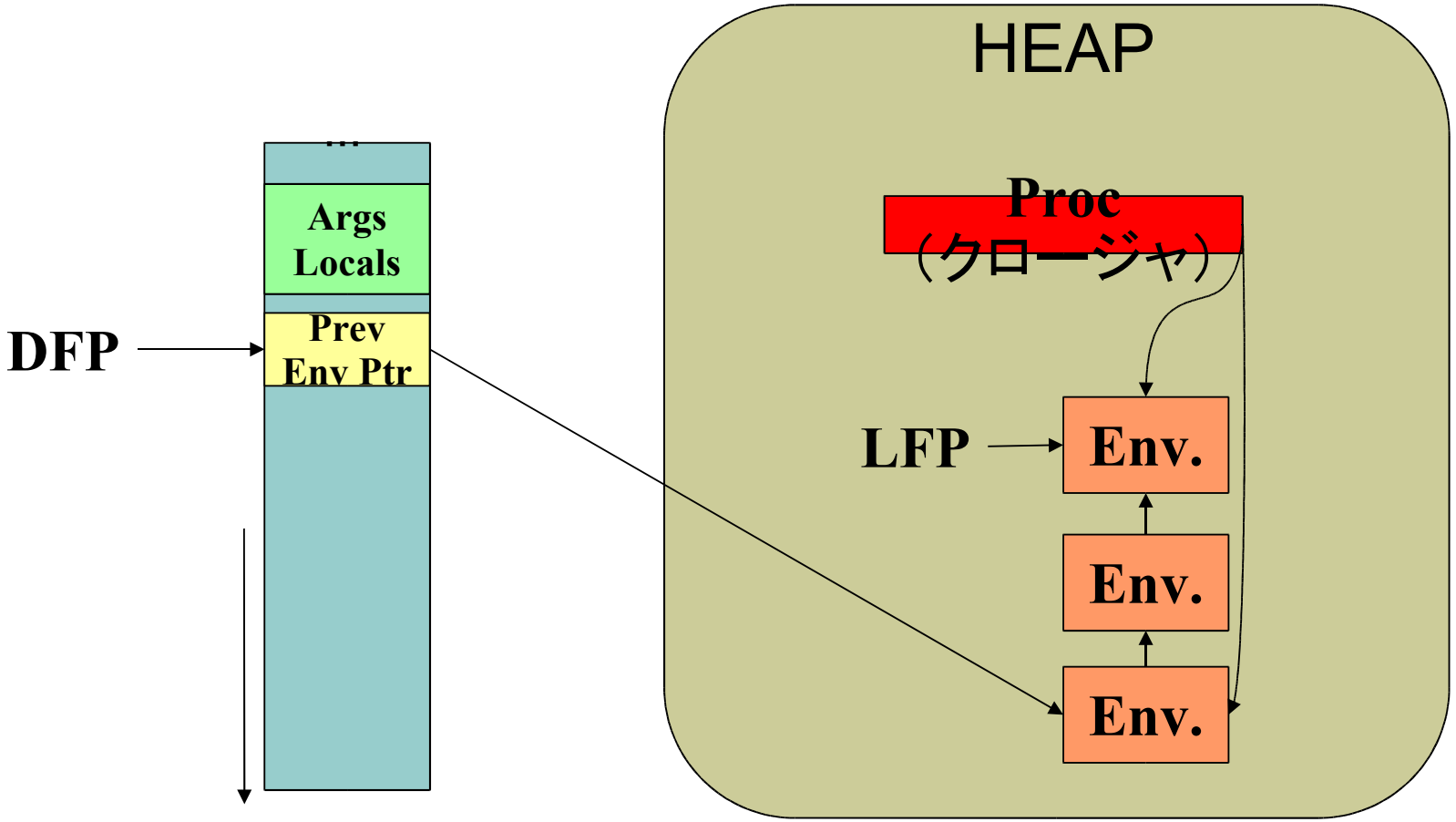
- PC: Program Counter
- SP: Stack Pointer
- CFP: Control Frame Pointer
- LFP: Local Frame Pointer
- DFP: Dynamic Frame Pointer

設計：スタックフレーム・環境





設計：スタックフレーム・環境（クロージャの実行）



実装

- CFP・PC をマシンレジスタに
  - GCC拡張： VALUE *pc asm(“esi”)
  - x86_64 の場合は SC 用レジスタも
- GCC コンパイルオプション-fcrossjumping
  - できるだけコードを共有するためのオプション
  - ジャンプだらけになるため無効
- その他、いくつかアセンブラを挿入

VM生成系:コード増加量に関する考察

- 生成結果
  - 52命令 → 総命令数875命令
  - VM関数: 5万行弱
  - オペランド・命令融合は組み合わせが爆発
- 増加に関する懸念
  - I-cache に載らなくなるのでは？ → 今のところ無い
  - コンパイル時間が長くなるのでは？→ 1分かかる
- 不必要な命令削除の必要性
  - Ertl らの研究(参考文献6))など参考に今後の課題

VM生成系：命令数の爆発

- オペランド融合と命令融合
  - オペランド融合：insnA op1, insnB op2
  - 命令融合：insnA + insnB
  - 生成される命令　4つ
    - insnA_insnB
    - insnA_op1_insnB
    - insnA_insnB_op1
    - insnA_op1_insnB_op1

（大きな声では言えない）Ruby処理系の課題（あるいは愚痴）

- 「ソースコードが仕様です」
  - 「バグもきちんと書かれている」
    → 開発時間の多くはRubyの仕様の調査
- 「プログラミングしやすいいろんな工夫」
  - いろんな機能を処理系が実装
    → 煩雑なVM

（Scheme はいいなぁ）

（でも、それはそれで大量にあるからつまらない）

最適化:コンパイル時の最適化

- 覗き穴最適化
- コンパイル時に利用するデータはGC対象外
  - コンパイル終了後開放
  - Rubyの配列オブジェクトを利用していたら巨大なコードのコンパイルが極端に遅くなってしまったため
    - 世代別GCが無いため

最適化：ダイレクトスレッデッドコード

- GCC拡張の label as value を利用したダイレクトスレッデッドコードを利用
  - 余計な jump 命令の除去
  - 分岐部分を分散するため、分岐予測効果向上
- その他の処理系では switch/case で

最適化：インラインキャッシュ

- インラインメソッドキャッシュ
  - メソッド検索結果を命令列にキャッシュ
- インラインキャッシュ
  - 検索結果のRuby定数値を命令列にキャッシュ
- キャッシュ時にVM状態カウンタも同時に格納
  - 現在の値と比較し、キャッシュの状態を確認
  - 定義・再定義が発生したときインクリメント

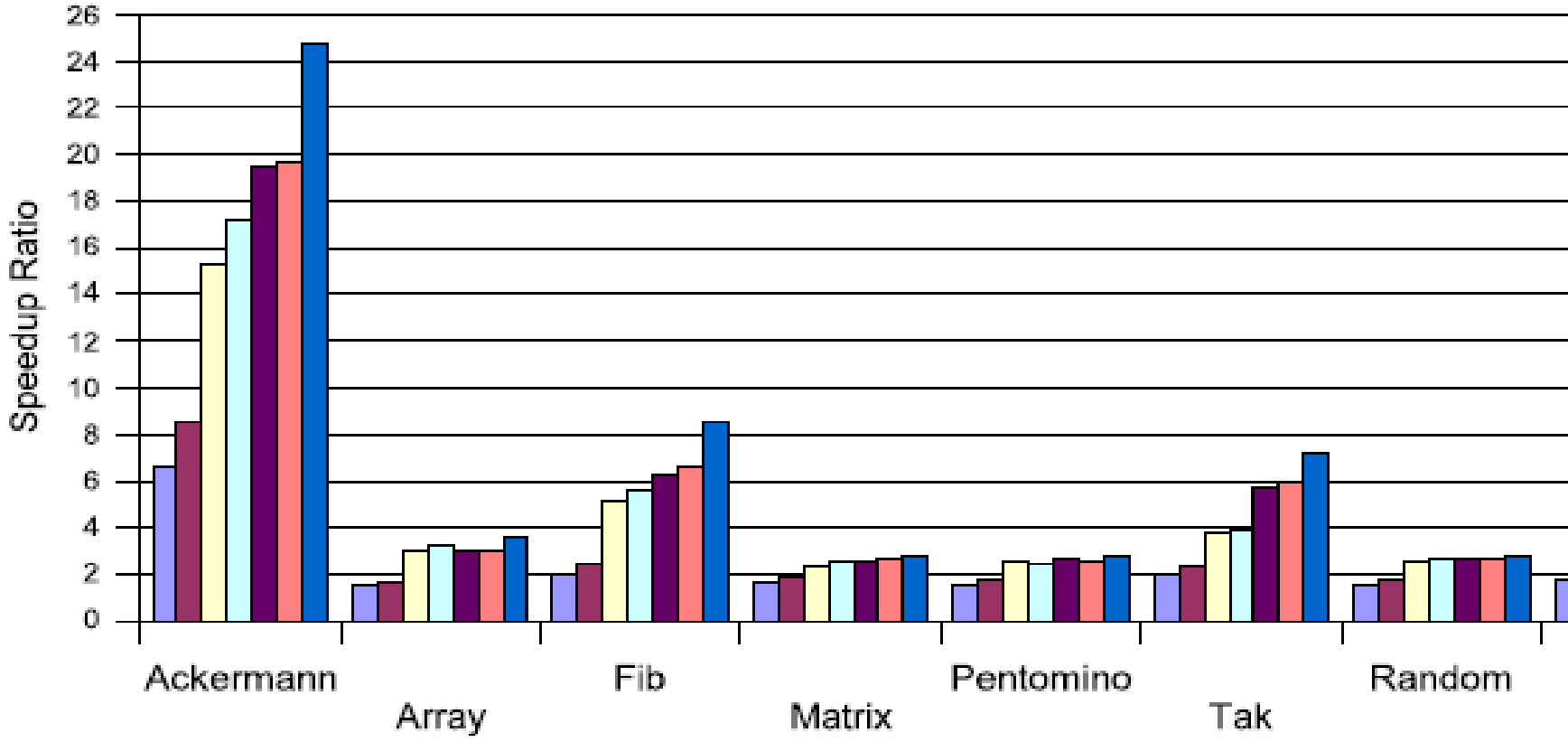
最適化：プロファイラ

- 実行時情報の集計
  - VMレジスタへのアクセス頻度
  - 命令実行頻度
  - オペランド出現頻度
  - 命令連結度（bigram）

最適化：ネイティブコンパイラ

- AOT コンパイラ
  - 命令列 → C 言語へ変換
- JIT コンパイラ
  - ネイティブコードへ変換
  - dynamic superinstruction
  - 昨年度挑戦して、一人では大変だということを実感
    - 劇的な高速化は得られなかった

評価:x86_64


Speedup Ratio
26
24
22
20
18
16
14
12
10
8
6
4
2
0
Ackermann
Array
Fib
Matrix
Pentomino
Tak
Random