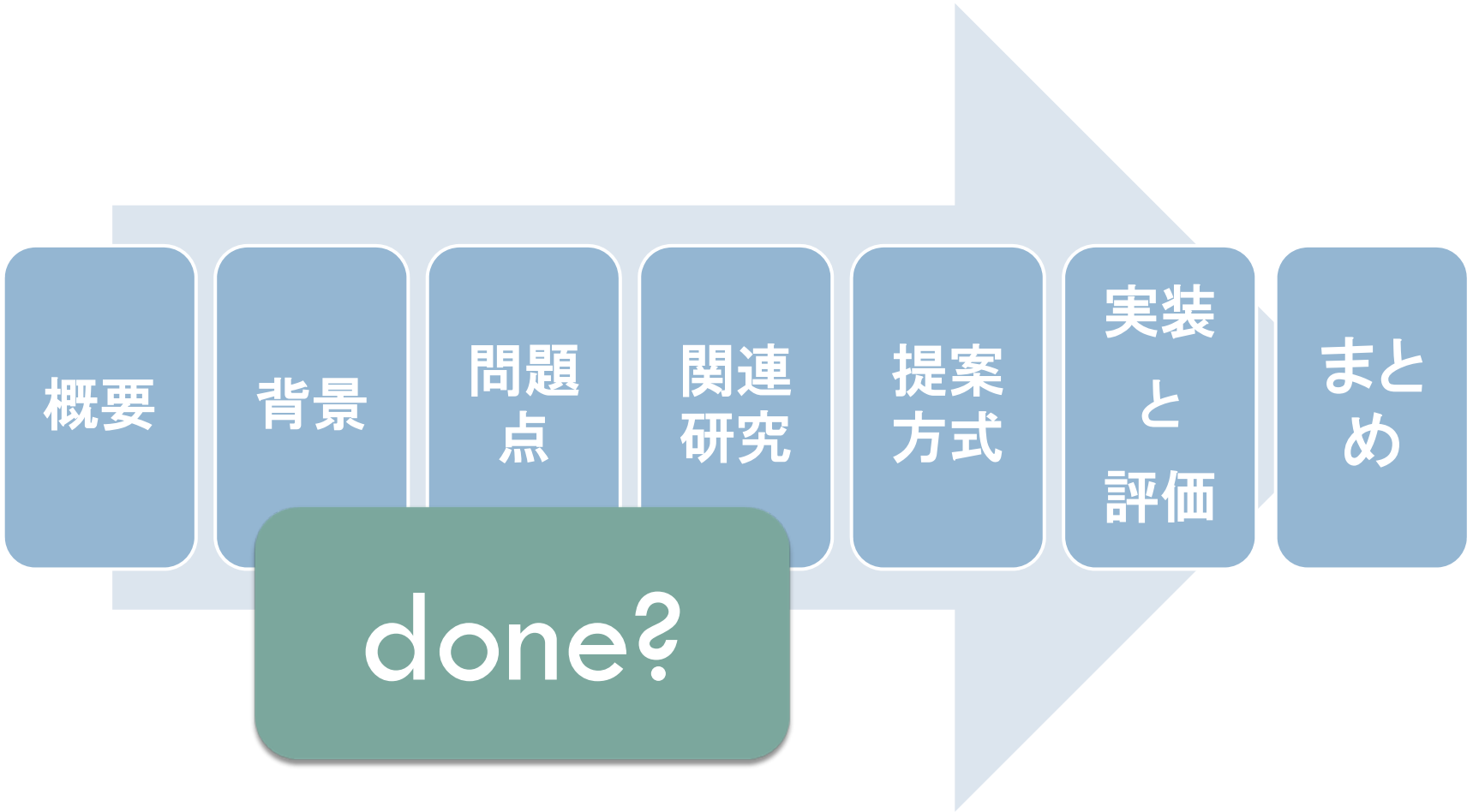
64bit CPU環境の

RUBY処理系での軽量な浮動小数点数表現

東京大学大学院　情報理工学系研究科　創造情報学専攻
笹田耕一

agenda

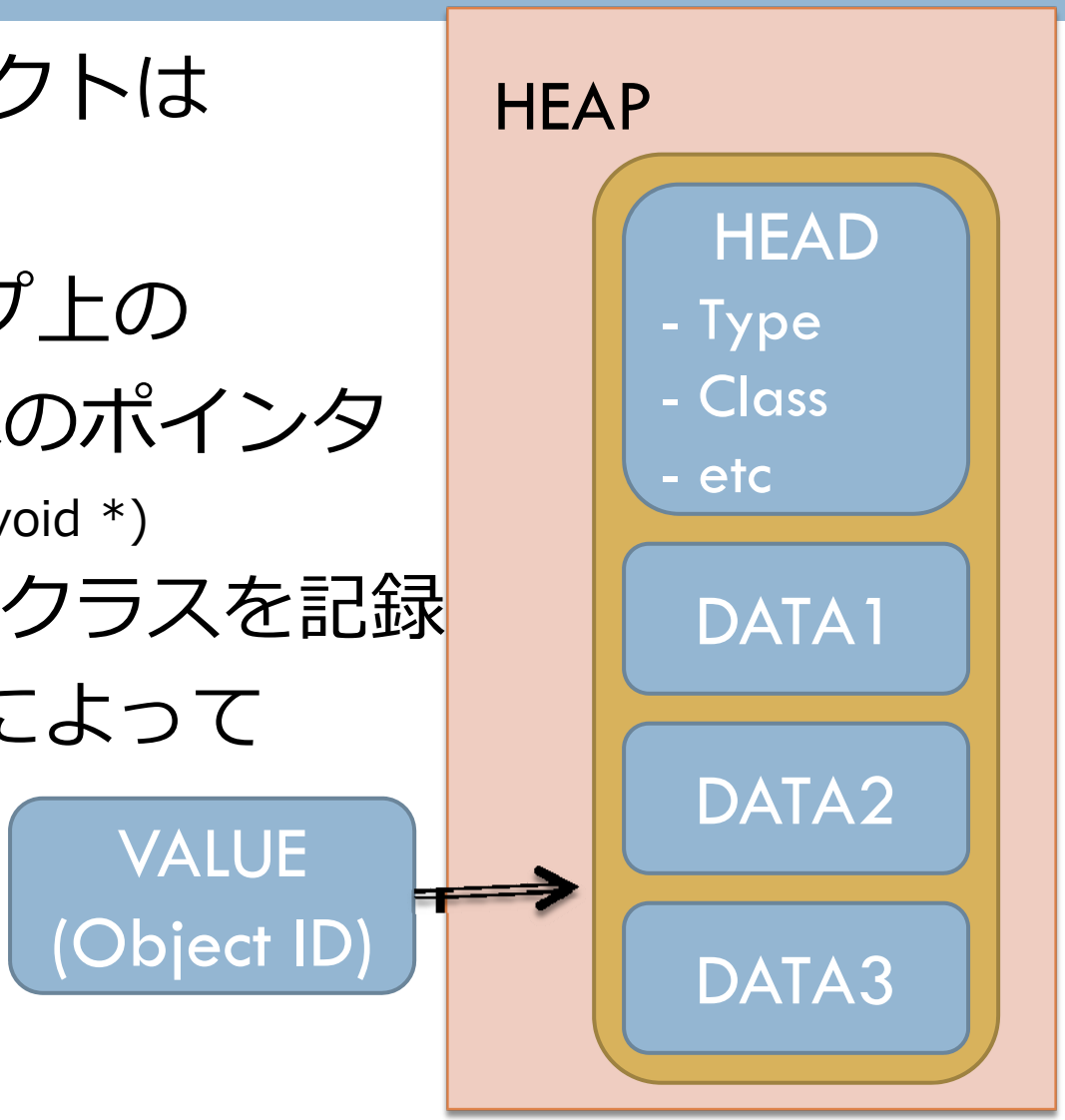


概要

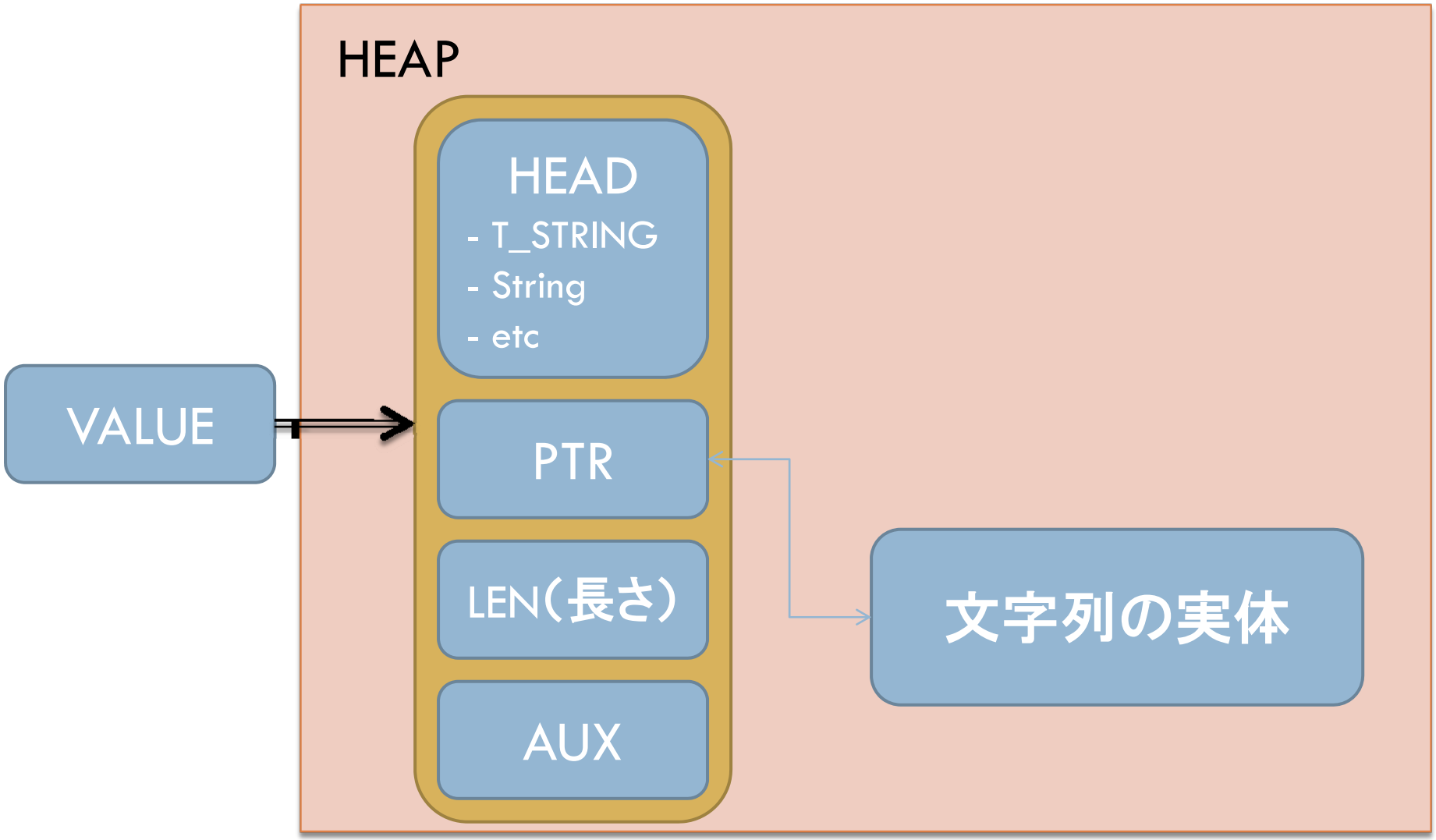
- 問題点
  - **Rubyの浮動小数点数計算は遅い**
  - 主なオーバヘッドは**メモリ管理**（Alloc + GC）
- 解決策
  - **64bit CPU**において、**64bit倍精度浮動小数点数**（Floatオブジェクト）を**タグ付きで表現**
  - **精度を犠牲にしない**方式
  - **埋め込み方を工夫して軽量な埋め込み**
- 結果
  - **速くなった！**　やった！
  - 実装も容易

背景
Rubyのオブジェクト表現

- Rubyのオブジェクトは
  VALUE型
- VALUE型はヒープ上の
  構造(5 words)へのポインタ
  * sizeof(word) == sizeof(void *)
- ヘッダにタイプ、クラスを記録
- DATA1-3はTypeによって
  使い方が異なる



背景
例:文字列オブジェクトの場合

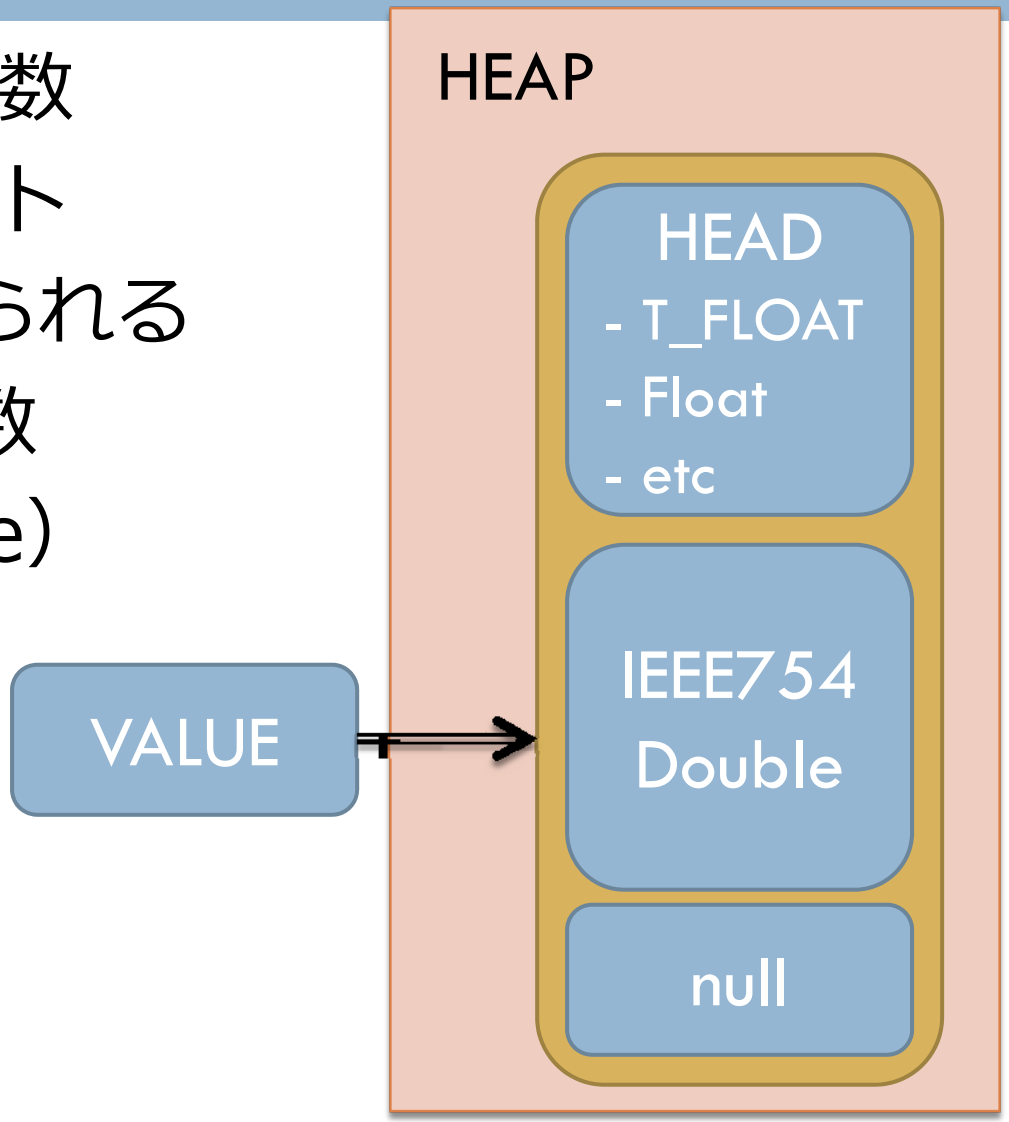


背景
VALUE埋め込みオブジェクト

- 整数など、大量に出てくるオブジェクトは VALUE埋め込みオブジェクトとして表現
- ポインタなら、下位２ビットは0のはず
  → 0でないVALUE型の値は特別扱い
  　VALUE埋め込み（タグ付き）オブジェクト
- 例
  - Fixnum（下位１ビットが1）
  - Symbol（下位8ビットが0x0e）
- よく使われる手法
  - ○ 生成が速い
  - × 制限が多い

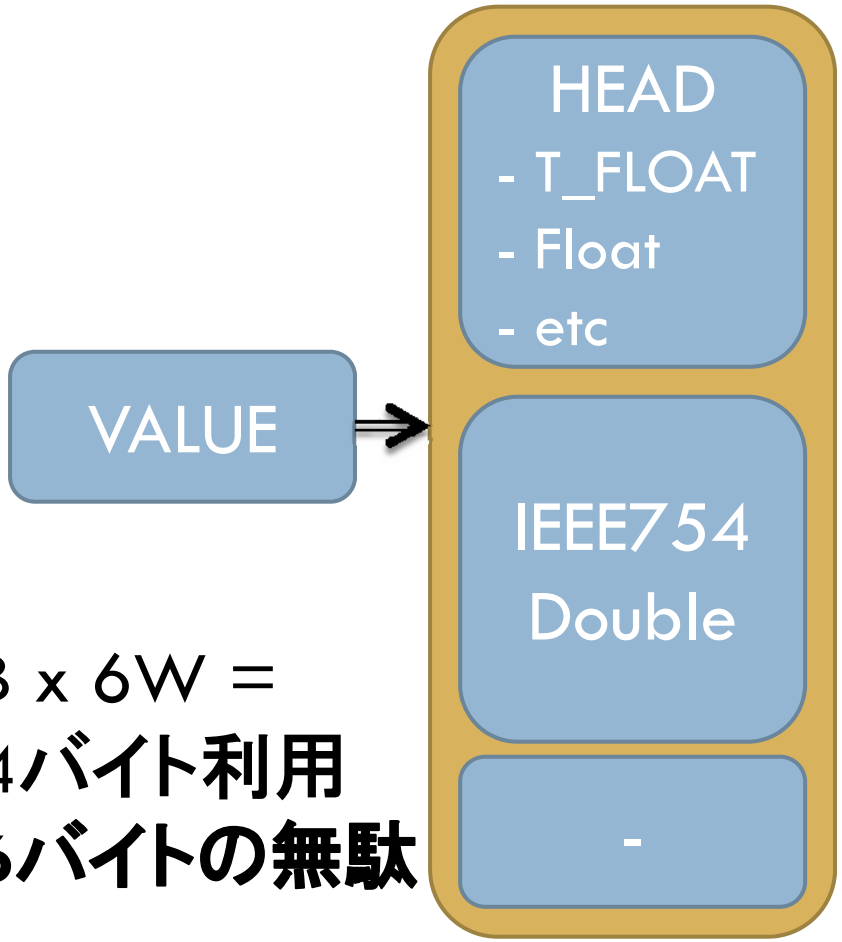
背景
Rubyの浮動小数点数表現

- Rubyの浮動小数点数
  →Floatオブジェクト
- ヒープに割り当てられる
- 倍精度浮動小数点数
  （IEEE754/double）



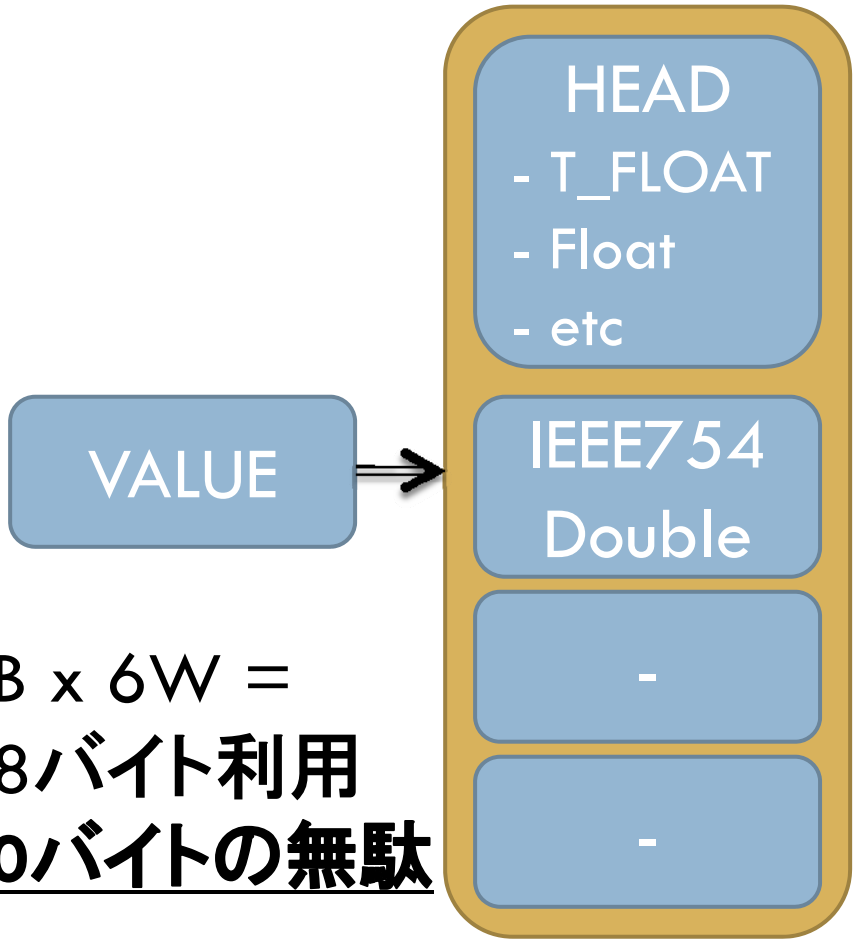
問題点
メモリ利用効率

32bit CPUの場合



4B x 6W =
24バイト利用
16バイトの無駄

64bit CPUの場合



8B x 6W =
48バイト利用
40バイトの無駄

問題点
実行効率(こちらが本題)

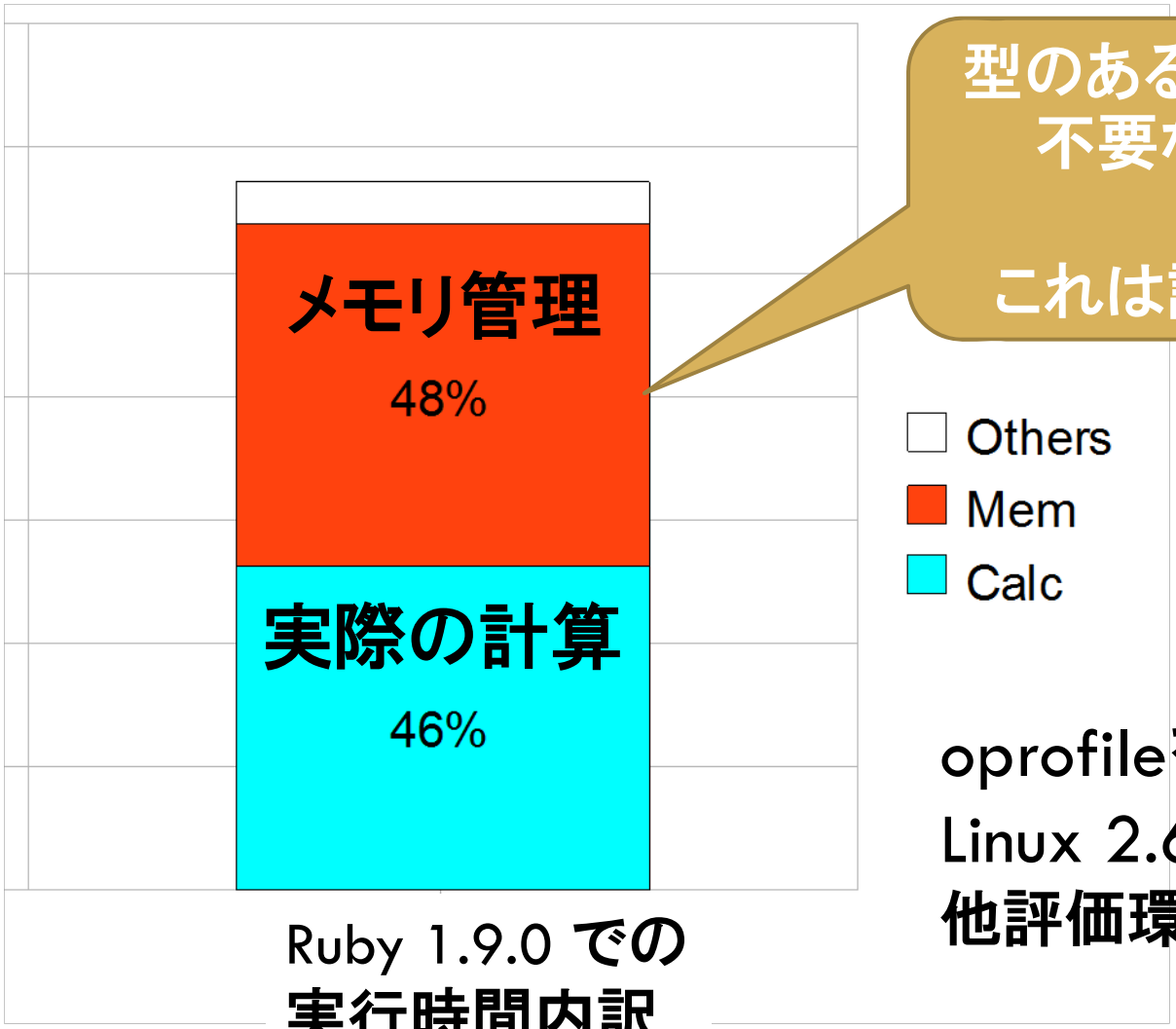
- 数値計算

  →大量の一時的なFloatオブジェクトを生成
- 数値計算のオーバヘッドの大半はメモリ管理
  - **オブジェクト生成**のオーバヘッド
  - **オブジェクト回収（ＧＣ）**のオーバヘッド

問題点
トイプログラム


```
i = 0; f = 0.0
while i<30_000_000
  i += 1
  f += 0.1; f -= 0.1
  f += 0.1; f -= 0.1
  f += 0.1; f -= 0.1
  f += 0.1; f -= 0.1
end
```

問題点
トイプログラムの実行時間内訳



Ruby 1.9.0 での
実行時間内訳

oprofileを利用して算出
Linux 2.6 (x86_64), Xeon
他評価環境は予稿集を参照

関連研究

- 関連研究
  - Ruby の話
    - NArray → Float しか扱わない配列で計算 [12]
    - Inline Ruby → C 関数で記述 [2]
  - 一般的な話
    - プログラム・型解析によりUnbox化 [7], …
      - 正当派
      - 解析が絶望的なRubyでは困難
    - NaN に色々埋め込み（Lisp, Lua）
    - メモリ番地によってヘッダを不要に [13]
      - 必要なメモリ消費量を削減
      - 実行時オーバヘッドの解決になっていない

復習
IEEE 754 倍精度浮動小数点数

64bit 倍精度浮動小数点数

s: 符号(1bit)

e: 指数部(11bit)

m: 仮数部(52bit)

$$-1^{s}2^{e-1024}m \quad \left(m = 1 + \sum_{i=0}^{51} \frac{b_i}{2^{52-i}}\right)$$

検討
VALUE埋め込みオブジェクトにするには？

- VALUE埋め込みオブジェクトにすれば(unboxed)
  メモリ管理オーバヘッド不要
- **64ビットCPUなら埋め込めそう**
  → 64bit CPUを前提に
- 最低、タグを埋め込む 1 bit が必要
  - 仮数部（52bit から 1bit）？
    - 精度が犠牲に
    - Self, Smalltalk の処理系で利用したらしい（裏取れず）
  - 指数部（11bit から 1bit）？
    - 表現可能な範囲が犠牲に
  - 符号は駄目（当たり前）

みんなもう
64bit CPU
だよね？

提案手法
精度を犠牲にしないVALUE埋め込みFloat

- 仮数部から1bit捻出
  ただし、範囲外は従来通りヒープに保存
  → 精度および範囲を制限しない
- 仮数部 e の範囲（0 ～ 2047）から捻出
  - だいたい、01000000000b～10111111111b あたりをよく利用（ 2^{-512}～2^{511} ）
    - **数値計算で範囲外の数はあまり出現しない（らしい）**
      → 現実的な解
  - この範囲だったらタグをb61に埋め込み
  - そうでなければ、ヒープに保存
    - NaN, ±Inf はヒープに保存される範囲に存在
  - ブログでこの方法自体は紹介されていたらしい [3]

提案手法
実際のプログラム

- 1) eの範囲チェック（512～1535）
- 2) 数値のエンコード（埋め込み）
- 3) 数値のデコード（値の取り出し）

提案手法
タグ付き表現可能かのチェック

- e: 1000000000b～0111111111b
- よく見ると b62 != b61 だったら埋め込み可
- Rubyでは、タグは下位ビットに埋め込む
  → **3bit 左に rotate すればよい**
  下位ビットの演算に持ち込む（64bit計算不要）
- b63 b62 b61 b60 … b0 → 3 bit rotate
  b60 … b0 b63 b62 b61
  このとき、**下位2bitが異なれば埋め込み可能**
- 埋め込めるときには b61 を 1 に（タグ）
- 値を取り出すときは逆

提案手法
エンコードのコード


```
VALUE rb_float_new(double dbl) {
  VALUE v1 = BIT_DOUBLE2VALUE(dbl);
  VALUE v2 = BIT_ROTL(v1, 3);
  if ((v2 + 1) & 0x02) // 下位2bitの判定
    return v2 | 0x01;     // タグ埋め込み
  else {
    if (dbl == 0) // 0 の場合
      return ruby_float_zero;
    else                  // ヒープから取り出し
      return rb_float_new_in_heap(dbl);
  }}
```

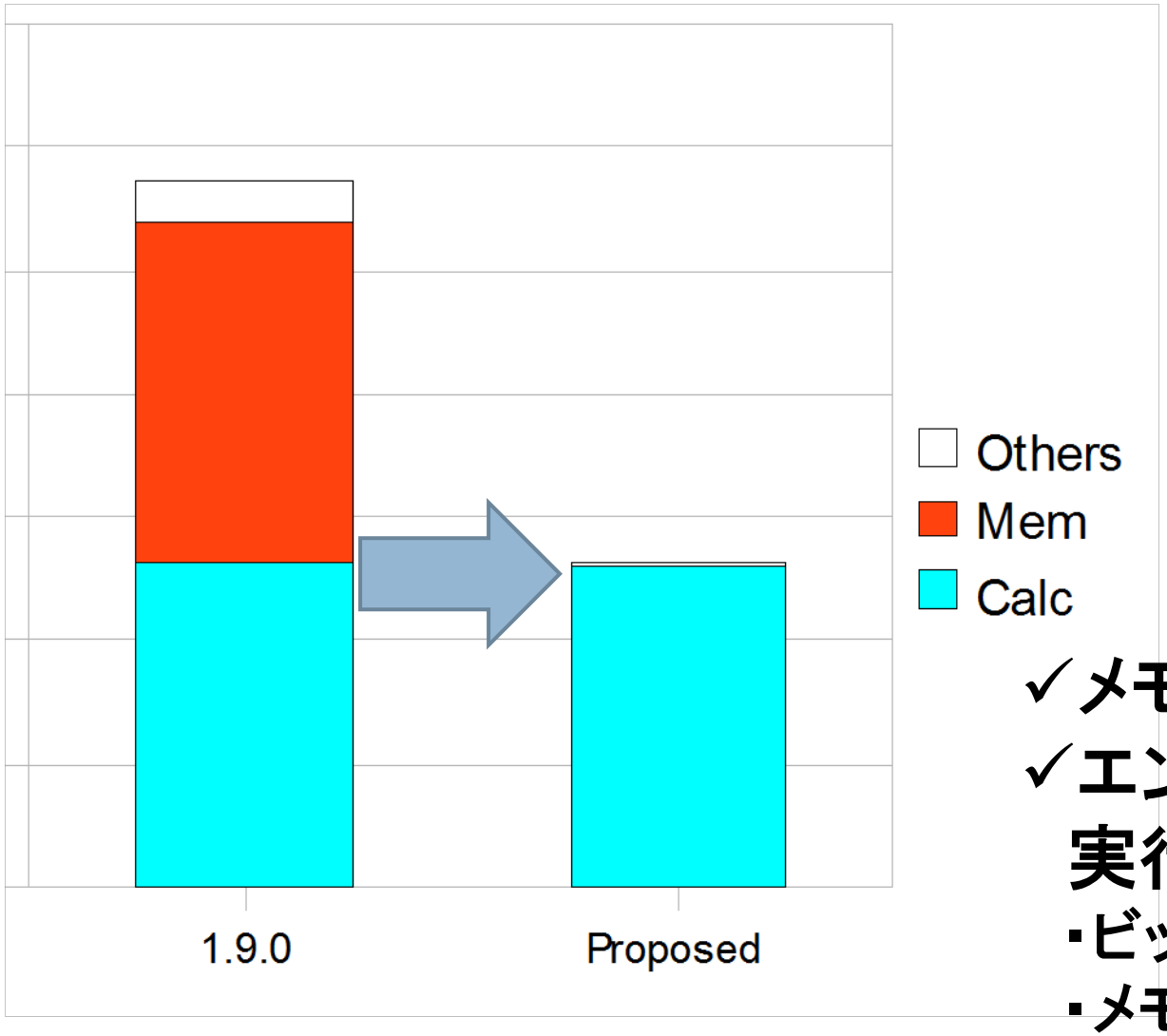
提案手法
デコードのコード


```
double RFLOAT_VALUE(VALUE v) {
  if (v & 1) {
    VALUE v1 = v ^ ((v >> 1) & 1);
    VALUE v2 = BIT_ROTR(v1, 3);
    return BIT_VALUE2DOUBLE(v2);
  }
  else
    return RFLOAT(v)->float_value;
}
```

実装

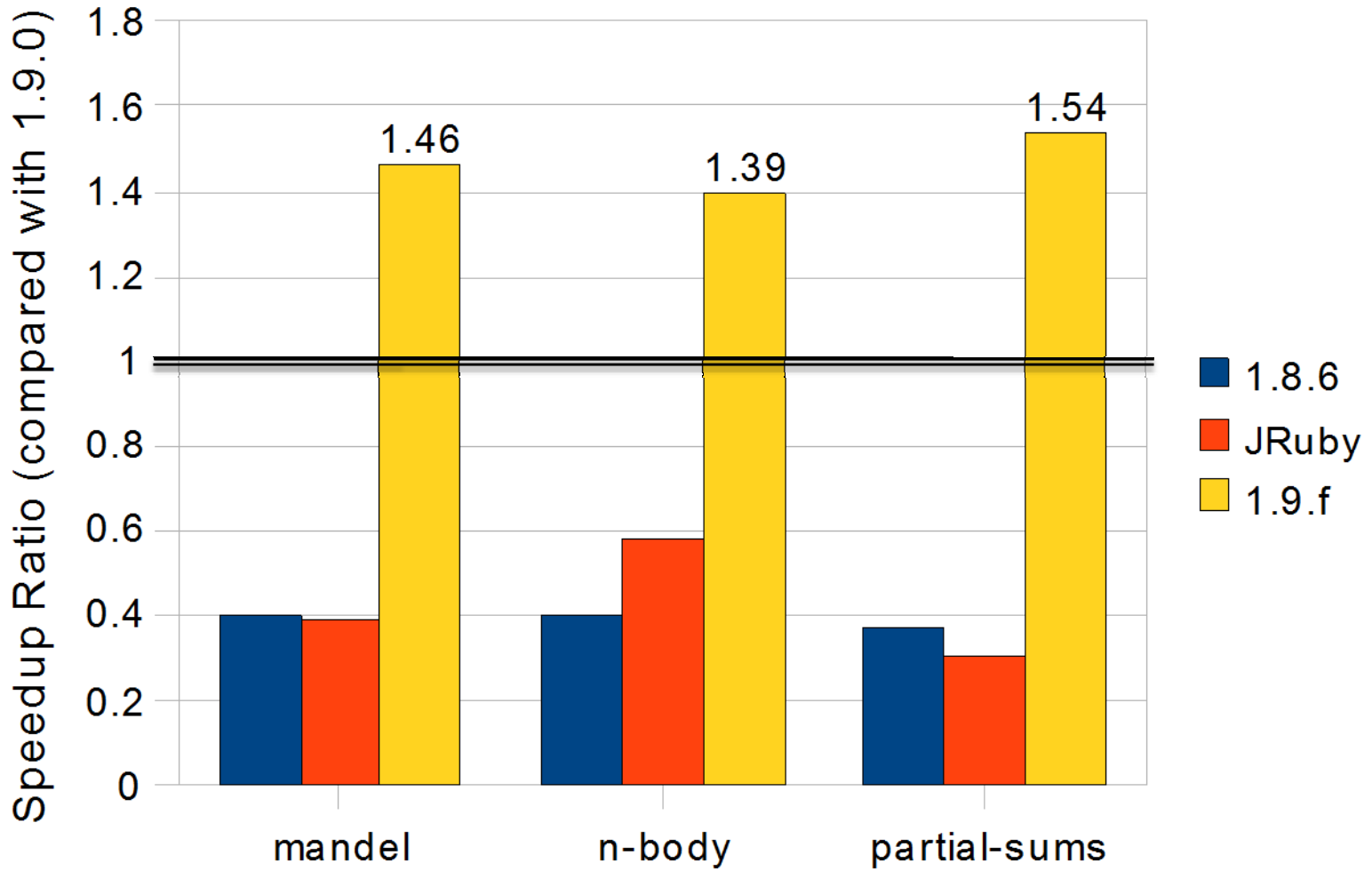
- Ruby 1.9.0-0 に実装
- 実装の変更点はごく一部（**実装が容易**）
  - Float の生成、Float からdouble値の取り出し
  - Fixnum の範囲を犠牲に（63bit → 61bit）
  - Fixnum の実装に依存していた部分の変更
  - 特化命令を埋め込みFloatに対応
- 仕様の変更点もごく一部（**ほぼ影響なし**）
  - Floatに特異メソッドが追加不可
  - FloatのオブジェクトIDの仕様が変更

評価
トイプログラム実行時間の改善



✓メモリ管理の時間削減
✓エンコード・デコード
実行時間は見えない
・ビット演算増
・メモリ間接アクセス減

評価(マイクロベンチマーク)
Ruby 1.9.0 と比較した速度向上比



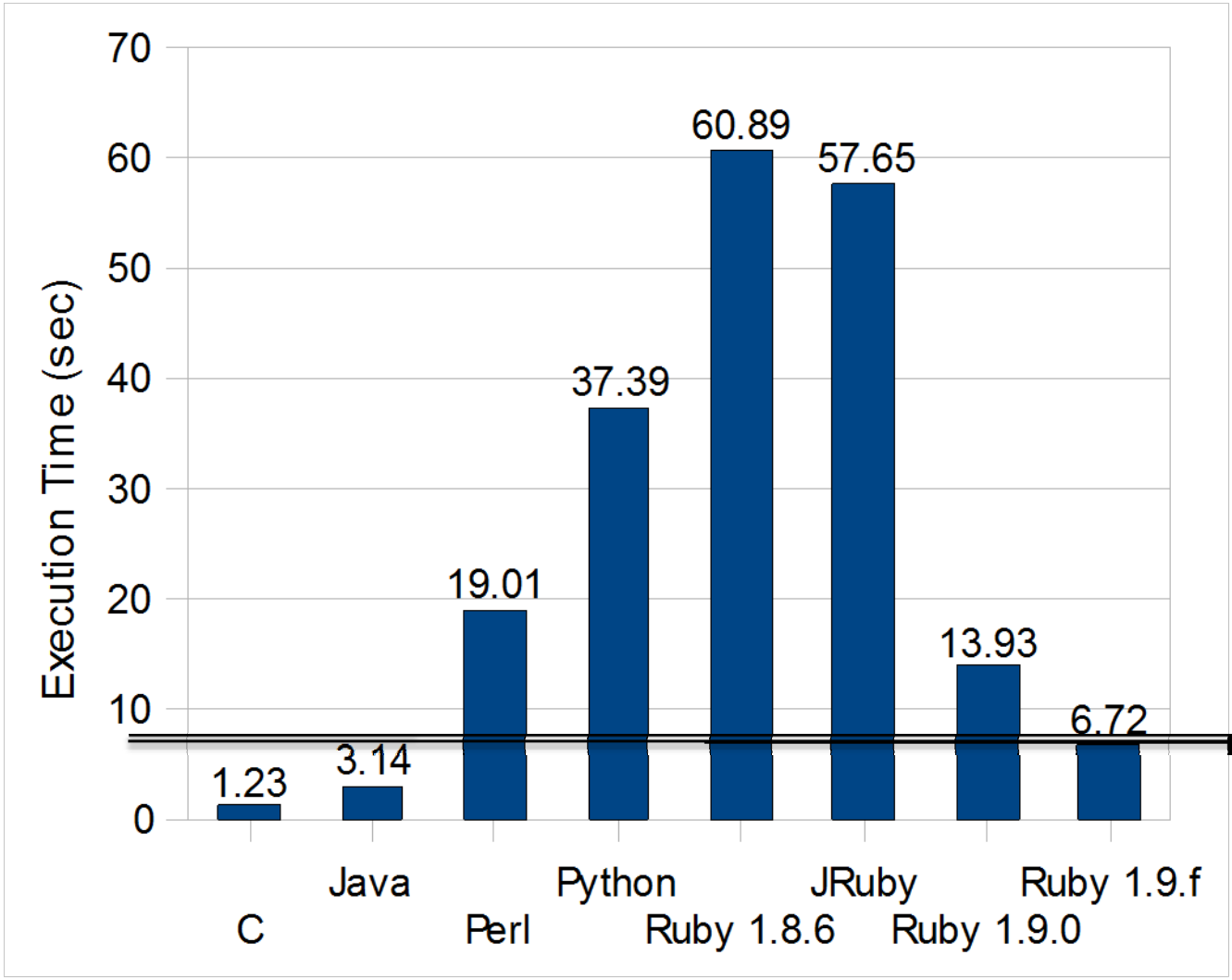
ベンチマークは Comp. Lang. Shootout [4] より

評価
他言語との比較

- 他言語で下記と同じようなコードで評価
- ただし、**C/Java では変数に volatile 付加**

```
i = 0; f = 0.0
while i<30_000_000
   i += 1
    f += 0.1; f -= 0.1
    f += 0.1; f -= 0.1
    f += 0.1; f -= 0.1
    f += 0.1; f -= 0.1
end
```

評価
他言語との比較(実行時間)



欠点・課題

- 欠点
  - 64bit CPUでしか使えない
    - 手元のラップトップで追加の評価が出来ない…
  - 整数計算が**若干**遅い(下位１ビットを譲ったため)
    - 具体的には「２つのオブジェクトが整数か？」というチェックの実行時間の増加（トレードオフ）
- 課題：特化命令で遅くなる場合がある
  - 何らかの工夫が必要

増えた！

「＋」の特化命令

(1) **整数かチェック**
(2) **埋め込みFloatかチェック（詳細略）**
(3) **文字列かチェック**
(4) **一般的な「+」メソッド呼び出し**

方式比較

- 32bit CPUでは優等生的shiro方式
- 64bit CPUでは単純なko1方式

方式比較	(1) shiro	(2) ko1
実行効率	○	◎
メモリ効率	○	◎
可搬性	○	× - 64bit CPU - IEEE754 表現
開発	大変	簡単

まとめ

- 64bit CPU上で精度や範囲を犠牲にしない軽量なFloatオブジェクト表現手法を提案
  - 単純な発想
  - 開発は容易
- Rubyで実装・評価
  - 実行効率改善を確認
  - Rubyの次のリリースに取り込む予定
- 他言語でも多分適用可能

おしまい

ご静聴ありがとうございました。

謝辞

発表した手法はRubyコミュニティ、
とくに三浦英樹氏との議論を参考にしております。
この場を借りてお礼申し上げます。

東京大学大学院　情報理工学系研究科　創造情報学専攻
ささだこういち
ko1@atdot.net