

# CLI の PHP プログラムを 限界まで高速化してみる

nsfisis (いまむら)

Ya8 2024

# 自己紹介

---

nsfisis (いまむら)



@ デジタルサーカス株式会社

# 高速化の対象

---

PHP で書かれた  
自作の WebAssembly ランタイム  
10 倍速くする

WebAssembly の簡単な説明

自作ランタイムの紹介

高速化

# WebAssembly の概要

# WebAssembly とは

---

## WebAssembly (Wasm)

ブラウザなどで実行できる  
ポータブルな仮想命令セット

# WebAssembly の出自

---

元々のモチベーション：ブラウザ上での高速な処理  
動的な JavaScript だと限界がある

間にいくつかの技術が生まれたり消えたりし、  
最終的に WebAssembly が策定された

## Emscripten

C/C++ のソースコードを Wasm に変換

C/C++ で書かれた膨大な資産を  
ブラウザの上で動かせる



# Wasm の活用例

---

PHP の処理系は C で書かれている

Emscripten を使って Wasm に変換できる

Wasm に変換するとブラウザ上で動かせる

# 自作ランタイムの紹介

# 処理系の紹介

---

```
>>> php -d memory_limit=4G -d opcache.enable_cli=on -d  
opcache.jit=on -d opcache.jit_buffer_size=1G examples/p  
hp-on-wasm/php-wasm.php
```

```
Decoding...
```

```
Instantiating...
```

```
Executing...
```

```
Hello, World!
```

```
Exit code: 0
```

# 処理系の紹介

---

今回作った Wasm 処理系

の上に、Wasm に変換された PHP 処理系

の上に、`echo "Hello, World!\n";`

# 処理系の紹介

---

普通の PHP 処理系

の上に、今回作った Wasm 処理系

の上に、Wasm に変換された PHP 処理系

の上に、`echo "Hello, World!\n";`

# 処理系の紹介

---

普通の PHP 処理系

の上に、今回作った Wasm 処理系

の上に、Wasm に変換された PHP 処理系

の上に、`echo "Hello, World!\n";`

多段になりすぎて実行に 30 秒かかる

メモリは 2.2 GiB 使う

高速化

# 初期ベンチマーク

---

実行時間 : 33.327 s

メモリ使用量 : 2.2 GiB



# ボトルネックを探す

# ボトルネックを探す

## 初期化

# 実行

<unknown>

<unknown>

# ボトルネックを探す

---

初期化処理が遅すぎる

メモリの初期化が遅すぎる

Wasm のメモリ表現を最適化

# メモリ表現の最適化

---

```
$data = array_fill(  
    0,  
    $size * 64 * 1024,  
    0,  
);
```

0 が  $\$size * 64 * 1024$  個並んだ巨大な配列

今回  $\$size$  は 2048

合計 132,644,864 要素

# メモリ表現の最適化

---

```
$data = array_fill(  
    0,  
    $size,  
    str_repeat("\0", 64 * 1024),  
);
```

64 KiB の string が \$size 個並んだ配列

# ベンチマーク

---

実行時間 : 33.327 -> 29.274 s

メモリ使用量 : 240 MiB (ほぼ 10 分の 1)

# ボトルネック探しの課題

---

Wasm の命令を実行する処理がすべて 1 つの関数に集約されている

```
function execInstr($instr) {  
  if ($instr instanceof Instrs\Numeric\F32Abs) {  
  
    ...  
  } elseif ($instr instanceof Instrs\Numeric\F32Add) {  
  
    ...  
  } elseif ($instr instanceof Instrs\Numeric\F32Ceil) {  
  
    ...  
  }  
}
```

どの命令が遅いかわからない

# ボトルネック探しの課題

Wasm の命令を実行する処理がすべて 1 つの関数に集約されている

```
function execInstr($instr) {  
  if ($instr instanceof Instrs\Numeric\F32Abs) {  
  
    ...  
  } elseif ($instr instanceof Instrs\Numeric\F32Add) {  
  
    ...  
  } elseif ($instr instanceof Instrs\Numeric\F32Ceil) {  
  
    ...  
  }  
}
```

どの命令が遅いかわからない

命令ごとに関数を分ける



# execInstr の最適化

```
function execInstr($instr) {  
  if ($instr instanceof Instrs\Numeric\F32Abs) {  
  
    ...  
  } elseif ($instr instanceof Instrs\Numeric\F32Add) {  
  
    ...  
  } elseif ($instr instanceof Instrs\Numeric\F32Ceil) {  
  
    ...  
  }  
}
```

elseif が多すぎる

クラスの判定が過剰な回数おこなわれる

# execInstr の最適化

```
function execInstr($instr) {  
  switch ($instr::class) {  
    case Instrs\Numeric\F32Abs::class:  
      ...  
    case Instrs\Numeric\F32Add::class:  
      ...  
    case Instrs\Numeric\F32Ceil::class:  
      ...  
  }  
}
```

switch-case に  
クラスの判定が 1 回になる

# ベンチマーク

---

実行時間 : 29.274 s -> 14.666 s

# execInstr の分離

命令ごとに個別の関数を用意

```
function execInstr($instr) {  
  return match ($instr::class) {  
    Instrs\Numeric\F32Abs::class =>  
      $this->execInstrNumericF32Abs($instr),  
    Instrs\Numeric\F32Add::class =>  
      $this->execInstrNumericF32Add($instr),  
    Instrs\Numeric\F32Ceil::class =>  
      $this->execInstrNumericF32Ceil($instr),  
    ...  
  };  
}
```

# ベンチマーク

---

実行時間 : 14.666 s -> 14.644 s

# ボトルネックは？

---

プロファイラが役に立たなくなってくる  
単純な命令が天文学的な回数実行される

# ボトルネックは？

---

ではどうするか

推測と計測のサイクルを回す

# ボトルネックは？

---

推測と計測のサイクルを回す

ボトルネックが減ってくると

遅いという確証を得るのが難しくなる

実際にやってみて測るしかない



# メモリアロケーションを減らす

---

メモリアロケーションを減らす

Strong typedef されたインデックス型

# メモリアロケーションを減らす

---

```
final readonly class MemIdx
{
    public function __construct(
        public int $value,
    ) {
    }
}
```

実質的にはただの `int` と同じ

メモリがもったいない

# ベンチマーク

---

実行時間 : 14.644 s -> 14.185 s

メモリ使用量 : 240 MiB -> 187 MiB

# メモリアロケーションを減らす

---

Wasm のプリミティブ

i32、i64、f32、f64

個別にクラスを用意

クラスのアロケーションコスト

# メモリアロケーションを減らす

---

Wasm のプリミティブ

i32、i64、f32、f64

型は事前にチェックされるので、

int と float で事足りる

# ベンチマーク

---

実行時間 : 14.185 s -> 8.300 s

メモリ使用量 : 187 MiB -> 187 MiB

# ナイーブな実装を最適化

---

memory.init と table.init 命令の最適化

仕様書をなぞった実装になっていて

明らかに非効率 (詳細は割愛)

# ベンチマーク

---

実行時間 : 8.300 s -> 6.650 s



# メモリアロケーションを減らす

---

スタックに積まれるもの

値、 コールフレーム、 ラベル

# メモリアロケーションを減らす

---

```
final class Stack
{
    /**
     * @param list<StackEntry> $entries
     */
    public function __construct(
        public array $entries,
    ) {
    }

    ...
}
```

# メモリアロケーションを減らす

---

```
abstract class StackEntry { ... }
```

```
final class Value extends StackEntry { ... }
```

```
final class Frame extends StackEntry { ... }
```

```
final class Label extends StackEntry { ... }
```

# メモリアロケーションを減らす

---

```
final class Stack
{
    /**
     * @param list<int|float|Ref|Frame|Label> $entries
     */
    public function __construct(
        public array $entries,
    ) {
    }

    ...
}
```

# ベンチマーク

---

実行時間 : 6.650 s -> 4.587 s

# 安全装置を切る

---

安全装置を切る

# 安全装置を切る

---

安全装置を切る

assert() を無効化する

# ベンチマーク

---

実行時間 : 4.587 s -> 3.407 s



# メモリアロケーションを減らす

---

命令の実行結果を表す

ControlFlowResult クラス

return、br 命令の結果

# メモリアロケーションを減らす

---

```
abstract readonly class ControlFlowResult {}

final class Return extends ControlFlowResult {}

final class Br extends ControlFlowResult
{
    public function __construct(
        public int $label,
    ) {
    }
}
```

# メモリアロケーションを減らす

---

```
// => -1
final class Return extends ControlFlowResult {}

// => $label
final class Br extends ControlFlowResult
{
    public function __construct(
        public int $label,
    ) {
    }
}
```

# ベンチマーク

---

実行時間 : 3.407 s -> 3.156 s

# メモリ表現の最適化

---

メモリ表現の最適化がまだ足りない

# メモリ表現の最適化

---

```
$data = array_fill(  
    0,  
    $size,  
    str_repeat("\0", 64 * 1024),  
);
```

64 KiB の string が \$size 個並んだ配列

# メモリ表現の最適化

---

```
function loadI32(int $n) {  
    $bytes = $this->sliceNBytes($n, 4);  
    return unpack('l', $bytes)[1];  
}
```

バイナリ列の切り出しと `unpack()` のコストが重い

# メモリ表現の最適化

---

```
int32_t loadI32(void* rawData, size_t n) {  
    return *(int32_t*)(rawData + n);  
}
```

Cならこう書ける



# メモリ表現の最適化

---

FFI: foreign function interface

アロケーションとポインタのキャストができる

# メモリ表現の最適化

---

```
// $this->rawData = $this->ffi->new(  
//     "uint8_t[$this->memorySize]",  
// );  
  
function loadI32(int $n) {  
    $dataAsInt32 = $this->ffi->cast(  
        "int32_t[$this->memorySize / 4]",  
        $this->rawData,  
    );  
    return $dataAsInt32[$n / 4];  
}
```

# ベンチマーク

---

実行時間 : 3.156 s -> 2.852 s

メモリ使用量 : 187 MiB -> 308 MiB

# ベンチマークまとめ

---

実行時間 : 33.327 s -> 2.852 s (11.7 倍)

メモリ使用量 : 2.2 GiB -> 308 MiB (7.4 分の 1)

10 倍以上速くなった

メモリアロケーションを減らす

すべての改善でベンチマークを取る

推測・適用・計測のサイクルを回す