Cコンパイラの末尾呼び出しを活用した効率的命令ディスパッチ手法の研究開発

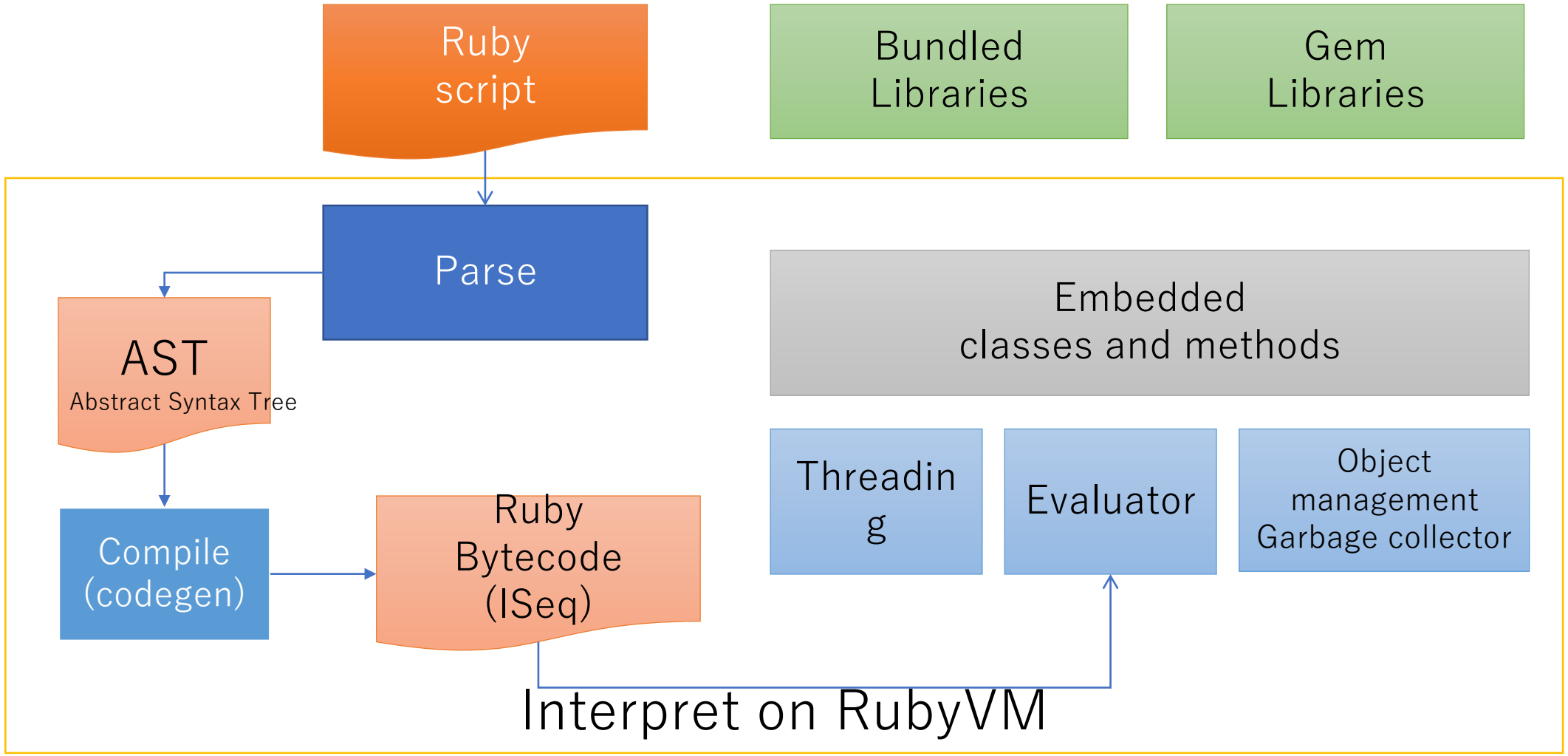
クックパッド

笹田耕一

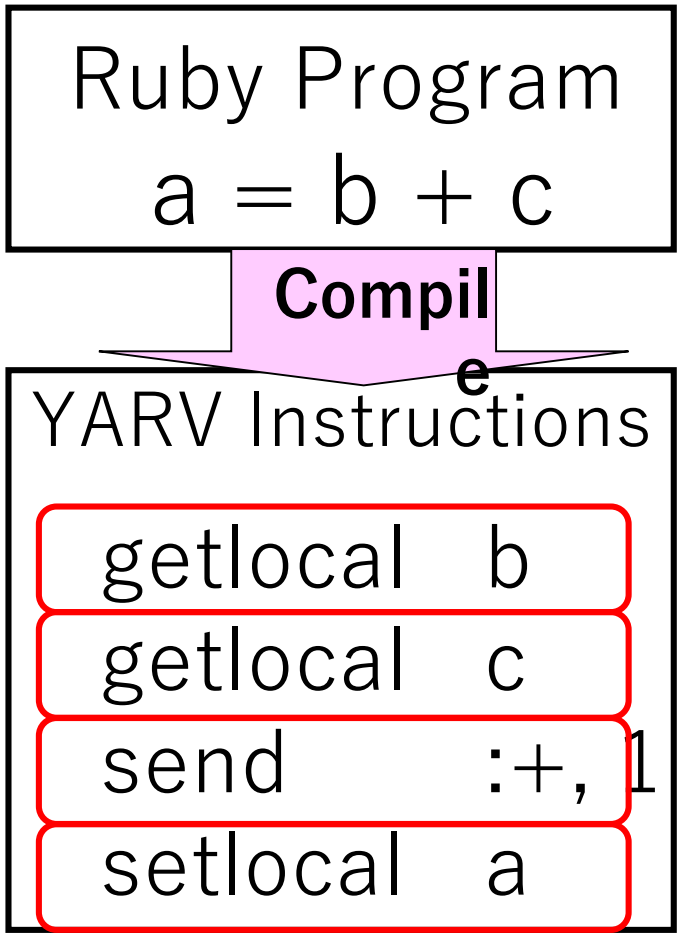
概要

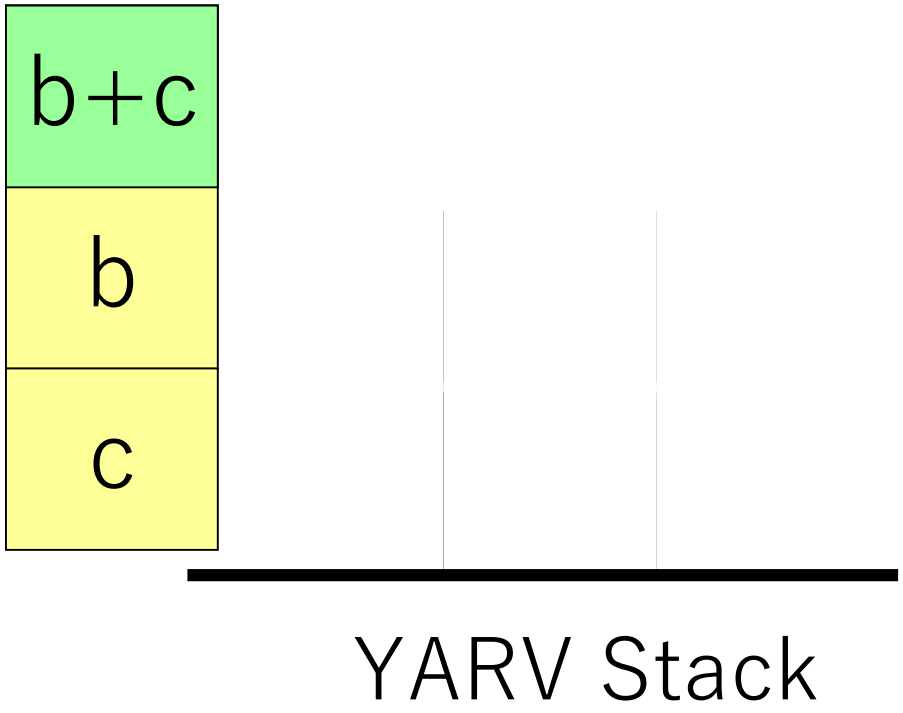
- Context Threading という VM の高速化技術を試しています
- C コンパイラによる末尾呼び出しの最適化をうまく活用できることを発見しました
- けど、そんなにも性能は上がらなかった感じなので、Ruby 2.7 には少なくとも入らないだろうな

背景


Ruby
script
Bundled
Libraries
Gem
Libraries
Parse
AST
Abstract Syntax Tree
Embedded
classes and methods
Threadin
g
Evaluator
Object
management
Garbage collector
Compile
(codegen)
Ruby
Bytecode
(ISeq)
Interpret on RubyVM

VM – スタックマシン





今日は、スタックマシンをたくさん使うよ。

Ruby Bytecode 実例

```
# Ruby
script
  a = 10
  if a > 1
    p :ok
  else
    p :ng
  end
```

```
0000 putobject          10
0002 setlocal           a, 0
0005 getlocal           a, 0
0008 putobject          1
0010 send               <callinfo!mid:>, argc:1, ARGS_SIMPLE>,
                        <callcache>, nil
0014 branchunless       27
0016 jump               18
0018 putself
0019 putobject          :ok
0021 send               <callinfo!mid:p, argc:1, FCALL|ARGS_SIMPLE>,
                        <callcache>, nil
0025 jump               34
0027 putself
0028 putobject          :ng
0030 send               <callinfo!mid:p, argc:1, FCALL|ARGS_SIMPLE>,
                        <callcache>, nil
0034 leave
```

スタックマシンの実行を解説

番地		スタックの状況（空でスタート）	解説
0000 putobject	10	# [10]	スタックに 10 をプッシュ
0002 setlocal	a, 0	# []	a にスタックトップの 10
0005 getlocal	a, 0	# [10]	a の値をスタックにプッシ
0008 putobject	1	# [10, 1]	1 をスタックにプッシュ
0010 send	<callinfo!mid:>, argc:1, ARGS_SIMPLE>,		
	<callcache>, nil		10.>(1) というメソッド呼び出し
		# [true]	その結果（true）をプッシュ
0014 branchunless	27	# []	スタックトップの値が fa
0016 jump	18	# []	無条件に 18 へジャンプ
0018 putself		# [self]	self をスタックへプッシュ
0019 putobject	:ok	# [self, :ok]	:ok をスタックへプッシュ
0021 send	<callinfo!mid:p, argc:1, FCALL|ARGS_SIMPLE>,		
	<callcache>, nil		p(:ok) を実行
		# [:ok]	結果の :ok をスタックにプッシュ
0025 jump	34	# [:ok]	無条件に 34 へジャンプ
0027 putself			
0028 putobject	:ng		
0030 send	<callinfo!mid:p, argc:1, FCALL|ARGS_SIMPLE>,		
	<callcache>, nil		
0034 leave		# []	このコードを終了する（積んであった

すべての計算が、
「スタックから値を取り出し」、「スタックに積む」ことで実現されていることがわかればOK

バイトコードをどう実行するか？
C の switch/case による素直な実装

（code[i] に命令番号が入っているとする）

```
while (1) {
  switch (code[pc]) {                  # 命令を取り出す
    case insnA:                        # switch/case で分岐
      do_A();                          # 命令を実行
      break;                           # 最初に戻る
    case insnB: do_B(); break;
    …
  }
}
```

素直な方法の問題点

- 命令を取り出す（フェッチ）
- switch/case で分岐
  - **分岐先を調べるのに時間がかかる**
  - 間接分岐で遅い（分岐予測が効きづらい）
- 命令を実行
  - ここはどうやってしょうがない
- 最初に戻る
  - **分岐アドレス同一化→ BTB を汚す可能性**

CPU の用語

- メモリキャッシュ
- 命令キャッシュ
- 分岐予測
- BTB (Branch Target Buffer)

Direct Threading (DT)

・code[i] に命令番地（&&label, gcc 拡張）が入っているとする
・つまり、事前に命令番号の列→命令番地の列へ変換が必要

```
 insnA:
    do_A();                          #  命令を実行
    goto *code[pc];                  #  次の命令番地をフェッチ
                                     #  次の命令番地に間接ジャンプ
 insnB:                              #    jmp *$rax 1命令
    do_B();
    goto *code[pc];
}
```

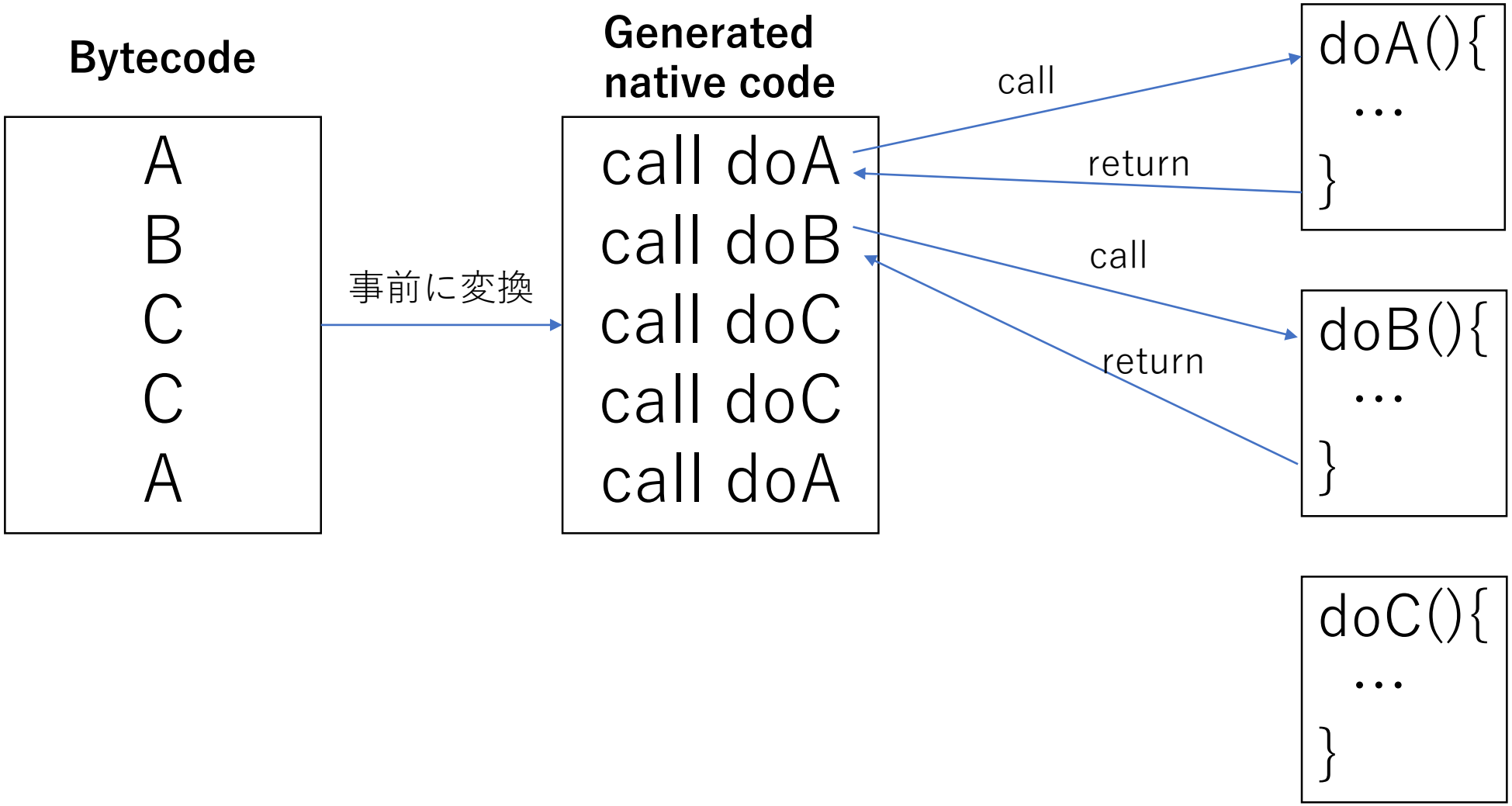
DTの利点と問題点

- 命令を実行
- 次の命令番地をフェッチ
  - Good: 次の分岐先番地を調べるコストが0
- 次の命令番地に間接ジャンプ
  - Good: 複数の場所で間接ジャンプするので、BTBを汚さない可能性増
  - Bad: 間接ジャンプなので分岐予測が効きづらい

Context Threading (CT)

- Marc Berndl, et.al.: Context Threading: A Flexible and Efficient Dispatch Technique for Virtual Machine Interpreters (2005)
- 命令番号の列を、call A, call B, … という、実際に動作する機械語列（ネイティブコード）に変換しておく
  - 機械語ちょっとした JIT みたいなもの
  - いわゆるJITとは違い、それ以外の最適化はあまりしない
  - なお、このテクニックだけだと、Subroutine Threading (ST) という

CT の前提の ST


Bytecode
A
B
C
C
A
事前に変換
Generated native code
call doA
call doB
call doC
call doC
call doA
call
return
call
return
doA(){
…
}
doB(){
…
}
doC(){
…
}

STの良いところ・悪いところ

- Good: ほとんどアセンブラを書かなくても良い
- Good: 間接ジャンプがないので、分岐予測が外れない
  - 厳密には return が間接ジャンプ（stack 上に保存された戻り番地へのジャンプ）だが、たいていは**専用ハードウェア**によりキャッシュされるのでノーペナルティ
- Good: 命令を増やしても命令キャッシュミスに悩まされない！

- Bad: 間接ジャンプはまだまだある
  - VM上命令で分岐した場合
  - VM上命令で関数（メソッド）呼び出した場合
  - → そういうのも全部ジャンプ先埋め込んじゃおう！

    → 全部埋め込んだやつが CT （今回はあまり触れません）

STの良いところ・悪いところ

- Bad: goto $reg よりも call/ret は遅い

Intel Skylake

Integer instructions

Instruction	Operands	µops fused domain	µops unfused domain	µops each port	Latency	Recipro-cal through put	Comments
Control transfer instructions							
JMP	short/near	1	1	p6		1-2	
JMP	r	1	1	p6		2	
JMP	m	1	2	p23 p6		2	
CALL	near	2	3	p237 p4 p6		3	
CALL	r	2	3	p237 p4 p6		2	
CALL	m	3	4	2p237 p4 p6		3	
RET		1	2	p237 p6		1	
RET	i		2			2	

https://www.agner.org/optimize/instruction_tables.pdf より一部抜粋

ST の悪いと思っていたところ

- 命令の実体（doA() など）に、どうやって情報を伝えるか？
  - 例えば、スタックポインタなどはどうやって伝えるか？
    - Ruby では、ec, cfp, pc, sp など4要素が対象
    - ただし、cfp->pc, cfp->sp のようにすることで2要素に縮約可能（実際、今はそうしている）
  - 素直にやるなら引数だが、call doA 命令の前に、毎回引数をセットアップするのか？
  - グローバル変数を使う？→スレッドで使えない
  - スレッドローカル変数 (TLS) を使う？→実は TLS は凄く遅い

- この問題がよくわからないので、使えなかった

末尾呼び出しを利用したSTの実現

- 素直に、doA(ec, cfp) と呼び出したらどうなるか on x86_64
  - call 前に、引数をセットアップ
    - `$rdi ← ec # 第一引数`
    - `$rsi ← cfp # 第二引数`
    - `call doA`
    - `$rdi ← ec # 第一引数`
    - `$rsi ← cfp # 第二引数`
    - `call doB`
    - …
  - さすがに同じことやるの無駄じゃないの？

```
doA(ec, cfp) {
  ec, cfp を使ってなにかする
}
```

末尾呼び出しを利用したSTの実現

- なんとかならない？
  - もし、doA() 内部で $rdi, $rsi のレジスタを、触らなければ…？
    - `$rdi ← ec # 第一引数`
    - `$rsi ← cfp # 第二引数`
    - `call doA`
    - `call doB`
    - …
  - しかし、$rdi, $rsi は破壊してもよいことになっている→ NG
- なんとかならないか…？

```
doA(ec, cfp) {
  ec, cfp を使ってなにかする
}
```

末尾呼び出しを利用したSTの実現

- 関数の最後に、$rdi, $rsi を再設定してくれれば良いのでは？
  - doA(), doB() の定義をこんな感じにすればできる
    - `$rdi ← ec # 第一引数`
    - `$rsi ← cfp # 第二引数`
    - `call doA`
    - `call doB`
    - …
- C で、これどうやって実現しよう？

```
doA(ec, cfp) {
  ec, cfp を使ってなにかする
  $rdi = ec; $rsi = cfp;
  return;
}
```

末尾呼び出しを利用したSTの実現

- そこで末尾呼び出し！
  - doA() の最後に tail(ec, cfp) を呼ぶ
    - `$rdi ← ec # 第一引数`
    - `$rsi ← cfp # 第二引数`
    - `call doA`
    - `call doB`
    - …
  - 1. tail 呼び出しのために $rdi, $rsi がセットアップされる
  - 2. tail では何もせずに返るため、$rdi, $rsi がセットアップされて戻る
  - できた！

```
doA(ec, cfp) {
  ec, cfp を使ってなにかする
  return tail(ec, cfp);
}
```

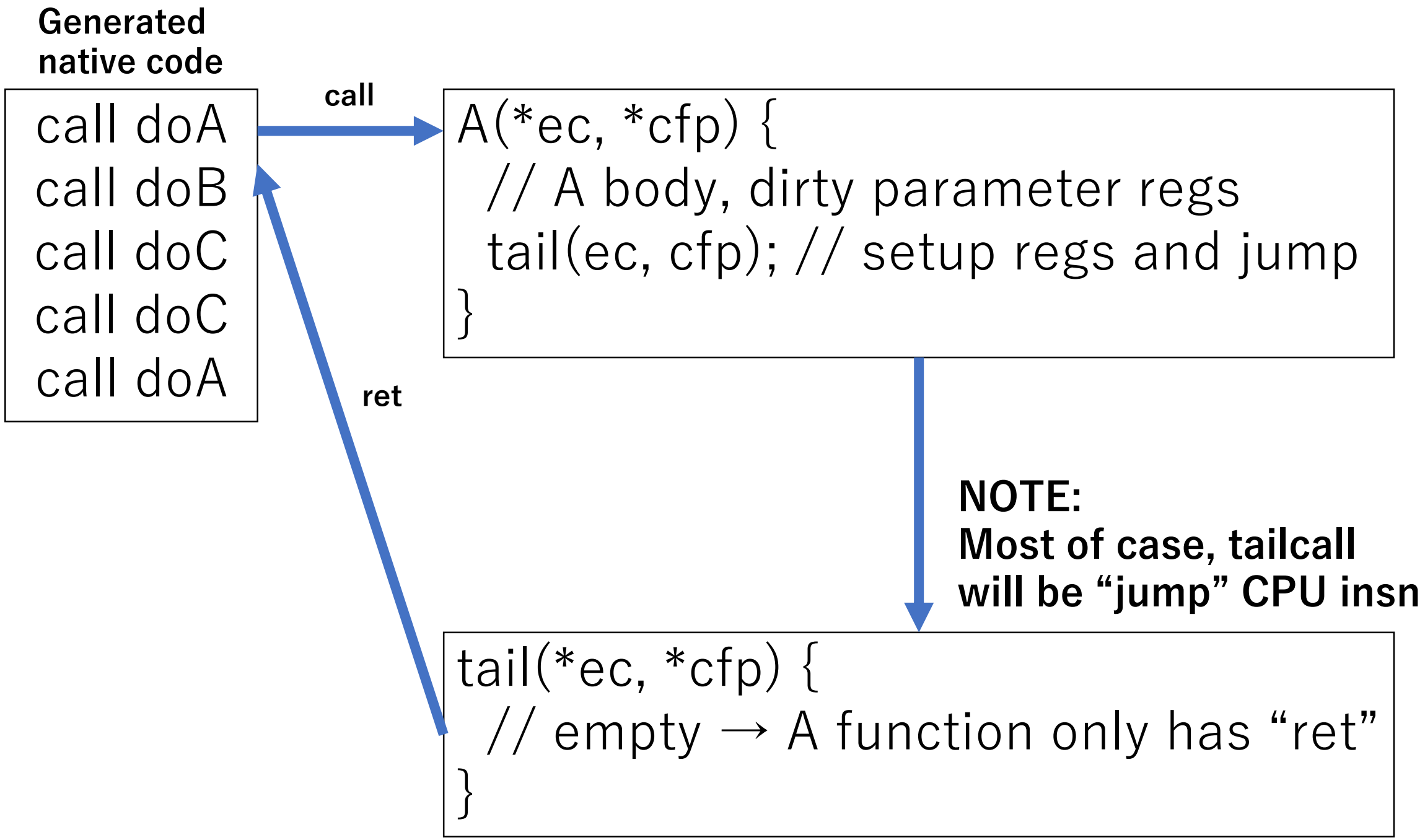
```
tail(ec, cfp) {
  // 何もしない
  return;
}
```

末尾呼び出しを利用したSTの実現

- 疑問点：call/return が増えて遅くなるんでは？
  - gcc -O3 だと、**末尾呼び出しの最適化**により、jmp tail になる（call doA()→ jmp tail → ret）
- 疑問点：jmp tail が一個余計に入るんでは？
  - そう。なので、dynamic に jmp tail 命令を、ret に書き換えることも可能
- 疑問点：何もしない tail って inline 化されちゃうんでは？
  - されちゃう。ので、別コンパイル単位（別ファイル）におく必要がある

```
doA(ec, cfp) {
  ec, cfp を使ってなにかする
  $rdi = ec; $rsi = cfp;
  return tail(ec, cfp);
}
```

```
tail(ec, cfp) {
  // 何もしない
  return;
}
```


Generated native code
call doA
call doB
call doC
call doC
call doA
call
ret
A(*ec, *cfp) {
// A body, dirty parameter regs
tail(ec, cfp); // setup regs and jump
}
NOTE:
Most of case, tailcall will be “jump” CPU insn
tail(*ec, *cfp) {
// empty → A function only has “ret”
}

末尾呼び出しを利用したSTの実現

- 疑問点：本当に動くの？
  - 動いた！　が、罠がいくつかあった
- 罠：-fstack-protector-strong
  - スタックが壊れてないかチェックする機構
  - なんと、$rsi を関数から返るときに破壊

  → このオプションとは共存できない

```
doA(ec, cfp) {
  ec, cfp を使ってなにかする
  $rdi = ec; $rsi = cfp;
  return tail(ec, cfp);
}
```

```
tail(ec, cfp) {
  // 何もしない
  return;
}
```

末尾呼び出しを利用したST実装の工夫

- pc, sp をレジスタに
  - x86_64 のABIでは、レジスタ渡しに 6 レジスタまで利用可能
  - ec, cfp, pc, sp をレジスタに

```
# DTでの nop
   0x5555557dc870 <vm_exec_core+1888>:  lea    0x8(%r14),%rdx # PC (cfp->pc) の操作
   0x5555557dc874 <vm_exec_core+1892>:  mov    %r14,%rax
   0x5555557dc877 <vm_exec_core+1895>:  mov    %rdx,(%r15)
   0x5555557dc87a <vm_exec_core+1898>:  mov    %rdx,%r14
   0x5555557dc87d <vm_exec_core+1901>:  jmpq   *0x8(%rax)     # 次の命令に

# ST での nop
call insn_func_nop
…
00000000000013e0 <insn_func_nop>:
    13e0:       48 83 c2 08             add    $0x8,%rdx      # PC の操作
    13e4:       e9 00 00 00 00          jmpq   rb_insn_tail
…
000000000001a830 <rb_insn_tail>:
   1a830:       f3 c3                   repz retq
```

末尾呼び出しを利用したST実装の工夫

- いくつかの命令を call ではなく直接埋め込み

```
case BIN(putobject):
  {
      VALUE obj = orig_iseq[i+1];

      if ((int32_t)obj == (int64_t)obj) {
          if ((int32_t)obj >= 0) {
              unsigned char mov_to_eax[5] = {
                  0xb8, 0, 0, 0, 0, // mov    $0x??,%eax
              };
              ((int32_t*)&mov_to_eax[1])[0] = (int32_t)obj;
              j += emit_code(&encoded[j], mov_to_eax, sizeof(mov_to_eax));
          }
          else {
              unsigned char mov_to_rax[7] = {
                  0x48, 0xc7, 0xc0, // mov    $0x??,%rax
              };
              ((int32_t*)&mov_to_rax[3])[0] = (int32_t)obj;
              j += emit_code(&encoded[j], mov_to_rax, sizeof(mov_to_rax));
          }
      }
      else {
          unsigned char mov_to_rax[10] = {
              0x48, 0xb8, // movabs $0x??, %rax
          };
          ((VALUE *)&mov_to_rax[2])[0] = obj;
          j += emit_code(&encoded[j], mov_to_rax, sizeof(mov_to_rax));
      }

      const unsigned char code[] = {
          0x48, 0x83, 0xc1, 0x08,                         // add    $0x8,%rcx
          0x48, 0x83, 0xc2, 0x10,                         // add    $0x10,%rdx
          0x48, 0x89, 0x41, 0xf8,                         // mov    %rax,-0x8(%rcx)
      };
      j += emit_code(&encoded[j], code, sizeof(code));
  }
  break;
```

末尾呼び出しを利用したST実装の工夫

- 命令の追加
  - いままでは、ある命令の中で型ごとに分岐
  - 命令を増やしてもあまりペナルティはなさそうなので、型ごとに命令を分岐（実験のために、特定の型だけを想定）

末尾呼び出しを利用したST実装の工夫

- いくつかの簡単な命令をインライン化
  - call/return が不用
  - Cで関数をコンパイル→貼り付け

```
#if USE_SUBR_OPTIMIZE
    case BIN(putobject):
      {
        VALUE obj = orig_iseq[i+1];

        if ((int32_t)obj == (int64_t)obj) {
          if ((int32_t)obj >= 0) {
            unsigned char mov_to_eax[5] = {
                0xb8, 0, 0, 0, 0, // mov    $0x??,%eax
            };
            ((int32_t*)&mov_to_eax[1])[0] = (int32_t)obj;
            j += emit_code(&encoded[j], mov_to_eax, sizeof(mov_to_eax));
          }
          else {
            unsigned char mov_to_rax[7] = {
                0x48, 0xc7, 0xc0, // mov    $0x??,%rax
            };
            ((int32_t*)&mov_to_rax[3])[0] = (int32_t)obj;
            j += emit_code(&encoded[j], mov_to_rax, sizeof(mov_to_rax));
          }
        }
        else {
          unsigned char mov_to_rax[10] = {
              0x48, 0xb8, // movabs $0x??, %rax
          };
          ((VALUE *)&mov_to_rax[2])[0] = obj;
          j += emit_code(&encoded[j], mov_to_rax, sizeof(mov_to_rax));
        }

        const unsigned char code[] = {
            0x48, 0x83, 0xc1, 0x08,                 // add    $0x8,%rcx
            0x48, 0x83, 0xc2, 0x10,                 // add    $0x10,%rdx
            0x48, 0x89, 0x41, 0xf8,                 // mov    %rax,-0x8(%rcx)
        };
        j += emit_code(&encoded[j], code, sizeof(code));
      }
      break;
#endif
#if USE_SUBR_OPTIMIZE
    case BIN(getlocal_WC_0):
      {
        unsigned long idx = (unsigned long)orig_iseq[i+1];

        if (idx >= 3 && idx < 10) {
          unsigned char idx_code = 0 - 8 * idx;
          const unsigned char code[] = {
              0x48, 0x8b, 0x46, 0x20,               // mov    0x20(%rsi),%rax
              0x48, 0x83, 0xc1, 0x08,               // add    $0x8,%rcx
              0x48, 0x83, 0xc2, 0x10,               // add    $0x10,%rdx
              0x48, 0x8b, 0x40, idx_code,           // mov    ?? (%rax),%rax
              0x48, 0x89, 0x41, 0xf8,               // mov    %rax,-0x8(%rcx)
          };
          j += emit_code(&encoded[j], code, sizeof(code));
        }
        else {
          j += emit_call(&encoded[j], func_ptr);
        }
      }
      break;
#endif
#if USE_SUBR_OPTIMIZE
    // simple copy instructions
    case BIN(putself):
      {
        const unsigned char code[] = {
            0x48, 0x8b, 0x46, 0x18,               // mov    0x18(%rsi),%rax
            0x48, 0x83, 0xc1, 0x08,               // add    $0x8,%rcx
            0x48, 0x83, 0xc2, 0x08,               // add    $0x8,%rdx
            0x48, 0x89, 0x41, 0xf8,               // mov    %rax,-0x8(%rcx)
        };
        j += emit_code(&encoded[j], code, sizeof(code));
      }
      break;
    case BIN(nop):
      {
        const unsigned char code[] = {
            0x48, 0x83, 0xc2, 0x08,               // add    $0x8,%rdx
        };
        j += emit_code(&encoded[j], code, sizeof(code));
      }
      break;
    case BIN(putnil):
      {
        const unsigned char code[] = {
            0x48, 0xc7, 0x01, 0x08, 0x00, 0x00, 0x00,       // movq   $0x8,(%rcx)
            0x48, 0x83, 0xc2, 0x08,               // add    $0x8,%rdx
            0x48, 0x83, 0xc1, 0x08,               // add    $0x8,%rcx
        };
        j += emit_code(&encoded[j], code, sizeof(code));
      }
      break;
    case BIN(pop):
      {
        const unsigned char code[] = {
            0x48, 0x83, 0xe9, 0x08,               // sub    $0x8,%rcx
            0x48, 0x83, 0xc2, 0x08,               // add    $0x8,%rdx
        };
        j += emit_code(&encoded[j], code, sizeof(code));
      }
      break;
    case BIN(dup):
      {
        const unsigned char code[] = {
            0x48, 0x8b, 0x41, 0xf8,               // mov    -0x8(%rcx),%rax
            0x48, 0x83, 0xc2, 0x08,               // add    $0x8,%rdx
            0x48, 0x83, 0xc1, 0x08,               // add    $0x8,%rcx
            0x48, 0x89, 0x41, 0xf8,               // mov    %rax,-0x8(%rcx)
        };
        j += emit_code(&encoded[j], code, sizeof(code));
      }
      break;
    case BIN(swap):
      {
        const unsigned char code[] = {
            0x48, 0x8b, 0x41, 0xf8,               // mov    -0x8(%rcx),%rax
            0x4c, 0x8b, 0x41, 0xf0,               // mov    -0x10(%rcx),%r8
            0x48, 0x83, 0xc2, 0x08,               // add    $0x8,%rdx
            0x4c, 0x89, 0x41, 0xf8,               // mov    %r8,-0x8(%rcx)
            0x48, 0x89, 0x41, 0xf0,               // mov    %rax,-0x10(%rcx)
        };
        j += emit_code(&encoded[j], code, sizeof(code));
      }
      break;
#endif
```

100行くらいのアセンブラ

評価：フィボナッチ

```
def fib n
  if n < 2
    n
  else
    fib(n-1) + fib(n-2)
  end
end

fib(37)
```

	実行時間 (s)
Master	1.13
Proposal	0.85

```
876.550221      task-clock (msec)         #    1.000 CPUs utilized
              2      context-switches          #    0.002 K/sec
              0      cpu-migrations            #    0.000 K/sec
          1,546      page-faults               #    0.002 M/sec
  3,230,313,777      cycles                    #    3.685 GHz                      (30.18%)
 10,601,611,221      instructions              #    3.28  insn per cycle           (37.94%)
  1,636,349,497      branches                  # 1866.806 M/sec                    (37.94%)
      2,892,903      branch-misses             #    0.18% of all branches         (38.19%)
  3,270,018,693      L1-dcache-loads           # 3730.555 M/sec                    (38.65%)
        377,713      L1-dcache-load-misses     #    0.01% of all L1-dcache hits    (38.79%)
          4,882      LLC-loads                 #    0.006 M/sec                    (31.03%)
            168      LLC-load-misses           #    3.44% of all LL-cache hits     (31.03%)
<not supported>      L1-icache-loads
         81,719      L1-icache-load-misses                                        (31.03%)
  3,304,954,152      dTLB-loads                # 3770.410 M/sec                    (31.03%)
         24,067      dTLB-load-misses          #    0.00% of all dTLB cache hits   (31.03%)
          2,551      iTLB-loads                #    0.003 M/sec                    (30.78%)
          1,194      iTLB-load-misses          #   46.81% of all iTLB cache hits   (30.32%)
<not supported>      L1-dcache-prefetches
<not supported>      L1-dcache-prefetch-misses

    0.876710034 seconds time elapsed
```

```
1249.288522      task-clock (msec)         #    1.000 CPUs utilized
              2      context-switches          #    0.002 K/sec
              0      cpu-migrations            #    0.000 K/sec
          1,540      page-faults               #    0.001 M/sec
  4,636,477,038      cycles                    #    3.711 GHz                      (30.84%)
 12,375,549,381      instructions              #    2.67  insn per cycle           (38.53%)
  1,667,763,942      branches                  # 1334.971 M/sec                    (38.53%)
     11,676,347      branch-misses             #    0.70% of all branches         (38.53%)
  4,165,508,205      L1-dcache-loads           # 3334.304 M/sec                    (38.53%)
        375,707      L1-dcache-load-misses     #    0.01% of all L1-dcache hits    (38.42%)
          5,287      LLC-loads                 #    0.004 M/sec                    (30.74%)
            143      LLC-load-misses           #    2.70% of all LL-cache hits     (30.74%)
<not supported>      L1-icache-loads
        143,364      L1-icache-load-misses                                        (30.74%)
  4,197,109,130      dTLB-loads                # 3359.600 M/sec                    (30.74%)
             10      dTLB-load-misses          #    0.00% of all dTLB cache hits   (30.74%)
         76,943      iTLB-loads                #    0.062 M/sec                    (30.74%)
            670      iTLB-load-misses          #    0.87% of all iTLB cache hits   (30.74%)
<not supported>      L1-dcache-prefetches
<not supported>      L1-dcache-prefetch-misses

    1.249460020 seconds time elapsed
```

perf stat –ddd の結果
st <-> master

評価：Optcarrot

- あるゲーム機のシミュレータ

	FPS（高い方が速い）
Master	46.4
Proposal	52.4
Proposal+	53.0

そんなに速くない…

```
3842.746149      task-clock (msec)         #    1.000 CPUs utilized
               4      context-switches          #    0.001 K/sec
               0      cpu-migrations            #    0.000 K/sec
          18,709      page-faults               #    0.005 M/sec
  14,481,884,052      cycles                    #    3.769 GHz                      (30.68%)
  42,320,810,357      instructions              #    2.92  insn per cycle           (38.38%)
   9,222,141,605      branches                  # 2399.883 M/sec                    (38.38%)
      10,654,527      branch-misses             #    0.12% of all branches         (38.41%)
  12,814,632,520      L1-dcache-loads           # 3334.759 M/sec                    (38.51%)
     140,452,010      L1-dcache-load-misses     #    1.10% of all L1-dcache hits    (38.51%)
       2,969,299      LLC-loads                 #    0.773 M/sec                    (30.81%)
         287,856      LLC-load-misses           #    9.69% of all LL-cache hits     (30.81%)
<not supported>      L1-icache-loads
      12,377,374      L1-icache-load-misses                                        (30.81%)
  12,898,660,140      dTLB-loads                # 3356.626 M/sec                    (30.81%)
         161,584      dTLB-load-misses          #    0.00% of all dTLB cache hits   (30.81%)
         154,492      iTLB-loads                #    0.040 M/sec                    (30.78%)
          64,487      iTLB-load-misses          #   41.74% of all iTLB cache hits   (30.68%)
<not supported>      L1-dcache-prefetches
<not supported>      L1-dcache-prefetch-misses

    3.843336476 seconds time elapsed
```

```
4331.488939      task-clock (msec)         #    1.000 CPUs utilized
               5      context-switches          #    0.001 K/sec
               0      cpu-migrations            #    0.000 K/sec
          18,560      page-faults               #    0.004 M/sec
  16,369,051,211      cycles                    #    3.779 GHz                      (30.73%)
  45,627,951,290      instructions              #    2.79  insn per cycle           (38.49%)
   8,307,968,885      branches                  # 1918.040 M/sec                    (38.58%)
      45,682,787      branch-misses             #    0.55% of all branches         (38.67%)
  15,185,108,345      L1-dcache-loads           # 3505.748 M/sec                    (38.68%)
     178,826,801      L1-dcache-load-misses     #    1.18% of all L1-dcache hits    (38.61%)
       2,968,733      LLC-loads                 #    0.685 M/sec                    (30.76%)
         109,344      LLC-load-misses           #    3.68% of all LL-cache hits     (30.67%)
<not supported>      L1-icache-loads
       3,856,244      L1-icache-load-misses                                        (30.66%)
  15,312,688,143      dTLB-loads                # 3535.202 M/sec                    (30.66%)
         322,839      dTLB-load-misses          #    0.00% of all dTLB cache hits   (30.66%)
       2,422,832      iTLB-loads                #    0.559 M/sec                    (30.66%)
       1,449,688      iTLB-load-misses          #   59.83% of all iTLB cache hits   (30.66%)
<not supported>      L1-dcache-prefetches
<not supported>      L1-dcache-prefetch-misses

    4.331753222 seconds time elapsed
```

perf stat –ddd の結果
st <-> master

末尾呼び出しを利用したST実装の工夫　＋

- jmp tail を、ret に書き換えてしまえばいいのでは？

→ 実行開始時に書き換えてしまおう

```
# ST での nop
call insn_func_nop
…
00000000000013e0 <insn_func_nop>:
    13e0:       48 83 c2 08             add    $0x8,%rdx       # PC の操作
    13e4:       e9 00 00 00 00          jmpq   rb_insn_tail
…
000000000001a830 <rb_insn_tail>:
   1a830:       f3 c3                   repz retq


# ST での nop +
00000000000013e0 <insn_func_nop>:
    13e0:       48 83 c2 08             add    $0x8,%rdx       # PC の操作
                                        ret
```

未評価

- まだバグが…
- icache が問題になりそうな大きいプログラムでは、まだ実行できていないので、真価が計れない

末尾呼び出しを利用したST
問題点

- 命令ボディから引数付き関数呼び出しをすると、ec等を保存しないといけない
- → 関数 prologue, epilogue での待避・復帰のコストが大きい
- → 致命的な問題

- どうしようかなあ、どうにもならないかなぁ
  - レジスタの値を復帰する命令を指定する？
  - global register variable を使う？（gcc 拡張）
  - callee save register を無理矢理増やす？（gcc 拡張）

末尾呼び出しを利用したST
問題点

- メモリ管理
  - GC どうしよう
  - call 32bit 相対縛り
- pc, sp をレジスタに持たせた問題
  - バックトレース生成時に pc が必要だが…
  - GC 時に sp が必要だが…

発展

- Stack caching ときっと相性が良い
- 命令をすごく増やしてちゃんと動くか検証したい

- 費用対効果、あるのかなあ？

試すには

- https://github.com/ko1/ruby/tree/subr_tailcall