基于 Sail 的指令集模拟器加速技术研究
——Pydrofoil: 加速基于 Sail 的指令集模拟器

摘要（Abstract）

https://arxiv.org/html/2503.04389

Pydrofoil，是一种多阶段编译器，可以从以高层次、面向验证的 ISA 规范语言 Sail 表达的处理器指令集架构（ISA）生成指令集模拟器（ISS）。在我们的基准测试中，Pydrofoil 相较于 Sail 生成的基于 C 的 ISS 显示出超过 230 倍的加速。

- ISS 实际上是一个解释器循环，即时编译（JIT）编译器在加速这些方面已被证明是有效的，尽管主要针对动态类型语言。
- ISS 工作负载非常不典型，主要由密集的位操作组成。传统编译器优化对于通用编程语言在加速此类工作负载方面的影响有限。我们开发了适合的领域特定优化。
- 仅靠追踪 JIT 编译器或提前编译（AOT）单独使用，即使结合领域特定的优化，也不足以生成高性能的 ISS。Pydrofoil 因此采用了一种混合方法，将 AOT 编译器与基于元追踪 PyPy 框架的追踪 JIT 相结合。AOT 和 JIT 使用特定领域的优化。我们的基准测试表明，结合 AOT 和 JIT 编译器的性能提升显著高于单独使用任一编译器。

研究动机和意义（Motivation）

- 导致 Sail 生成的 ISS 运行缓慢的瓶颈是什么？
- JIT 编译能否显著加速 Sail 生成的 ISS？
- 我们能在多大程度上接近针对 RISC-V 手动调优的工业级 ISS 的性能？

结果（Result）

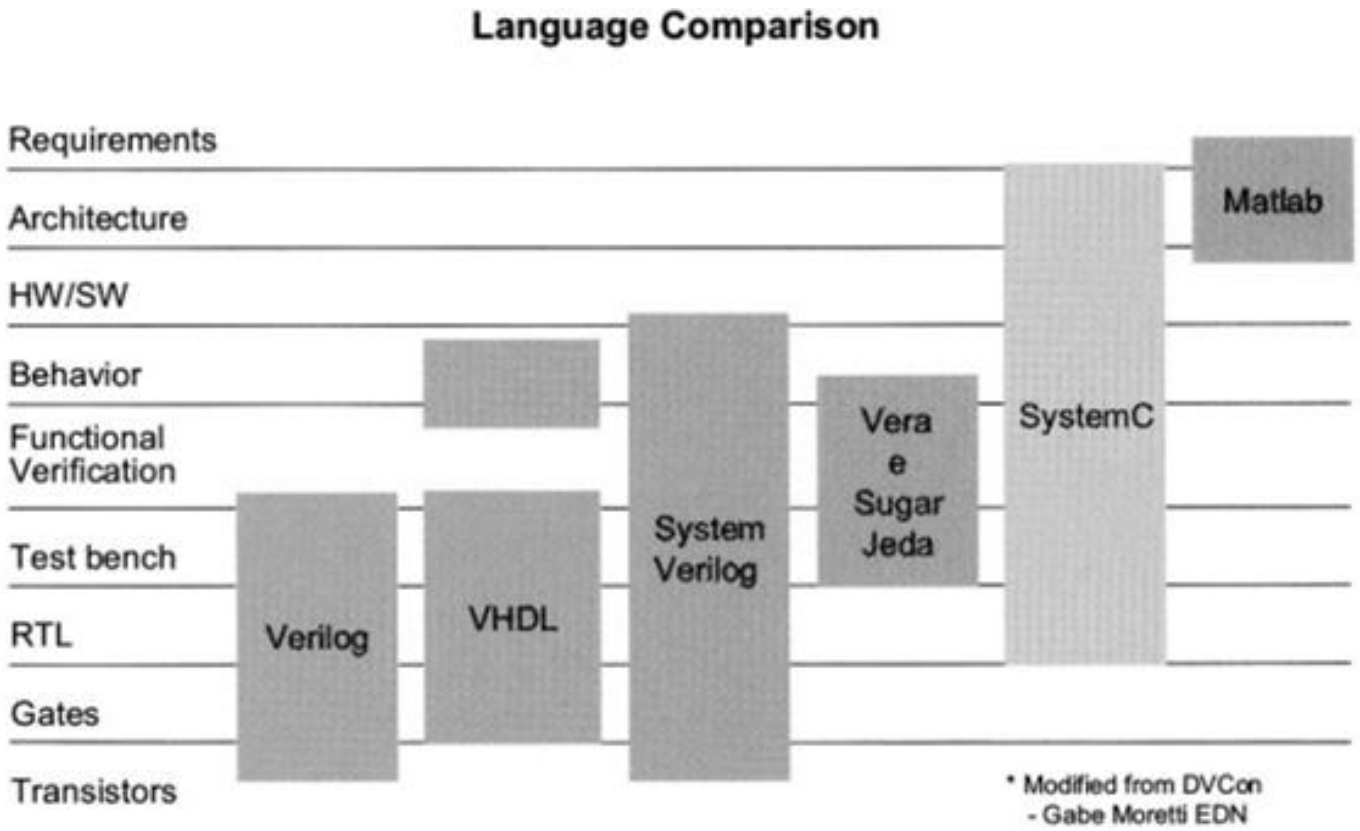
- Pydrofoil 相较于 Sail 生成的 ISS 展现了超过 230x 的加速
- 与手写和优化的 QEMU 相比则表现出 26.7x 的减速

背景

指令集模拟器（ISS）用途广泛

- 许多硬件特性首先使用模拟器进行原型设计，例如新的流水线设计、缓存布局、缓存一致性协议等。
- 模拟是低成本的，常用于评估新软件技术的有效性，例如编译器优化，因为模拟器通常相比真实硬件具有更好的可观察性。
- 模拟的一个常见用途是设计探索。对特定系统进行大量配置，结果用于指导进一步的研究和开发

DSLs used in computer architecture




```systemc
#include "systemc.h"

SC_MODULE(adder)          // module (class) declaration
{
  sc_in<int> a, b;        // ports
  sc_out<int> sum;

  void do_add()           // process
  {
    sum.write(a.read() + b.read()); //or just sum = a + b
  }

  SC_CTOR(adder)          // constructor
  {
    SC_METHOD(do_add);    // register do_add to kernel
    sensitive << a << b;  // sensitivity list of do_add
  }
};
```

nML

A Structural Processor Modeling Language for Retargetable Compilation and ASIP Design

```
opn alu_instr (op : alu, al : alu_left_opd, ar : alu_right_opd)
{
    action {
      stage EX1:
        A = al.value;
        B = ar.value;
        switch (op) {
          case add : C = add(A, B, AS) @alu;
          case sub : C = sub(A, B, AS) @alu;
          case and : C = A & B @alu;
          case or  : C = A | B @alu;
        }
        S = C @sh;
        AR = S;
    }
    syntax : AR " = " al op ar;
    image  : op::ar::al;
}
```

4.4 Instruction-set Grammar 75

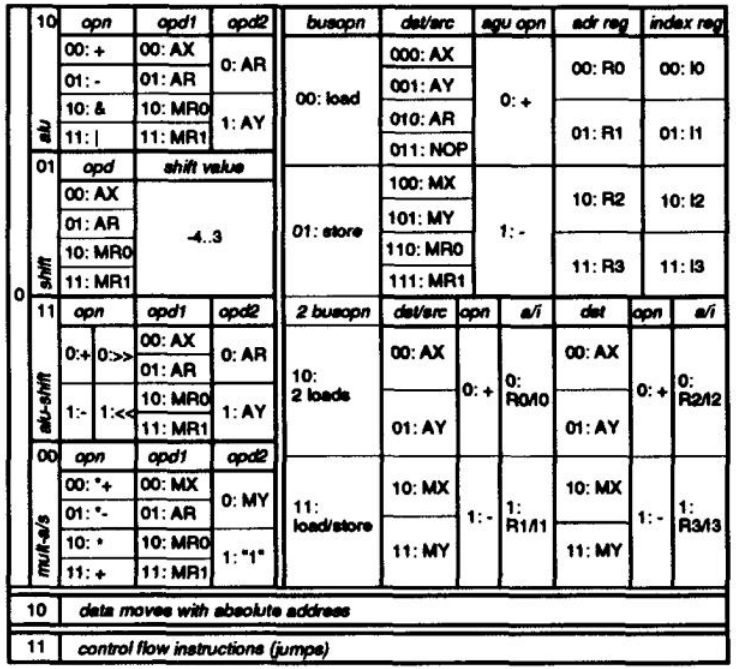


Figure 2: The instruction set table.

```
opn arith_instr (alu_instr | as_sh_instr | shift_instr
                 | macc_rnd_nop_instr)
{
    image : "00"::alu_instr  |
            "01"::as_sh_instr  |
            "10"::shift_instr  |
            "11"::macc_rnd_nop_instr;
}
```

https://ieeexplore.ieee.org/document/470354

ArchC

ArchC: a systemC-based architecture description language

```
ac_format Type_I =
   "[%imm4:1 %imm3:6 %imm2:4 %imm1:1 | %imm8:4 %imm7:4
%imm6:4 | %csr:12] %rs1:5 %funct3:3 %rd:5 %op:7";

ac_instr<Type_I> ADDI, SLTI, SLTIU, XORI, ORI, ANDI;

ADDI.set_asm("ADDI %reg, %reg, %exp", rd, rs1,
imm4+imm3+imm2+imm1);
   ADDI.set_asm("MV %reg, %reg", rd, rs1);
   ADDI.set_decoder(funct3 = 0x0, op = 0x13);
```

```
239     // Instruction ADDI behavior method.
240     void ac_behavior(ADDI) {
241       int imm;
242       imm = (imm4 << 11) | (imm3 << 5) | (imm2 << 1) | imm1;
243       dbg_printf("ADDI r%d, r%d, %d\n", rd, rs1, imm);
244       if ((rd == 0) && (rs1 == 0) && (imm == 0)) {
245         dbg_printf("NOP executed!");
246       } else {
247         int sign_ext;
248
249         sign_ext = sign_extend(imm, 12);
250         ac_Sword rs1_value = RB[rs1];
251         RB[rd] = rs1_value + sign_ext;
252         dbg_printf("RB[rs1] = %d\n", RB[rs1]);
253         dbg_printf("imm = %d\n", sign_ext);
254         dbg_printf("Result = %d\n\n", RB[rd]);
255       }
256     }
```

ACL2 x86

```
(def-inst x86-bt-0F-A3
  ;; TO-DO: Speed this up!

  ;; 0F A3: BT r/m16/32/64, r16/32/64

  ;; If the bitBase is a register operand, the bitOffset can be in the
  ;; range 0 to [15, 31, 63] depending on the mode and register size.
  ;; If the bitBase is a memory address and bitOffset is a register
  ;; operand, the bitOffset can be:

  ;; Operand Size   Register bitOffset
  ;;      2          -2^15 to 2^15-1
  ;;      4          -2^31 to 2^31-1
  ;;      8          -2^63 to 2^63-1

  :parents (two-byte-opcodes)

  :returns (x86 x86p :hyp (x86p x86))

  :guard-hints (("Goal" :in-theory (e/d (segment-base-and-bounds)
                                        ())))

  :prepwork
  ((local
    (defthm dumb-signed-byte-p-guard-lemma
      (implies (not (mv-nth 0
                            (x86-effective-addr
                             proc-mode p4 temp-rip rex-byte
                             r/m mod sib num-imm-bytes x86)))
               (signed-byte-p 48
                              (mv-nth 2
                                      (x86-effective-addr
                                       proc-mode p4 temp-rip rex-byte
                                       r/m mod sib num-imm-bytes x86))))))

   (local
    (in-theory (e/d ()
                    (acl2::mod-minus
                     signed-byte-p
                     unsigned-byte-p)))))

  :modr/m t

  :body

  ;; Note: opcode is the second byte of the two-byte opcode.

  (b* ((p2 (prefixes->seg prefixes))
       (p4? (equal #.*addr-size-override*
                   (prefixes->adr prefixes)))

       (seg-reg (select-segment-register proc-mode p2 p4? mod r/m sib x86))

       ((the (integer 1 8) operand-size)
        (select-operand-size
         proc-mode nil rex-byte nil prefixes nil nil nil x86))

       (bitOffset (rgfi-size operand-size
                             (reg-index reg rex-byte #.*r*)
                             rex-byte
                             x86))

       ((mv flg0
            (the (signed-byte 64) addr)
            (the (unsigned-byte 3) increment-RIP-by)
            x86)
        (if (equal mod #b11)
            (mv nil 0 0 x86)
          (let ((p4? (equal #.*addr-size-override*
                            (prefixes->adr prefixes))))
            (x86-effective-addr proc-mode p4?
                                temp-rip
                                rex-byte
                                r/m
                                mod
                                sib
                                0 ;; No immediate operand
                                x86))))
       ((when flg0) (!!ms-fresh :x86-effective-addr-error flg0))

       ((mv flg (the (signed-byte #.*max-linear-address-size*) temp-rip))
        (add-to-*ip proc-mode temp-rip increment-RIP-by x86))
       ((when flg) (!!ms-fresh :rip-increment-error temp-rip))
```

ARM ASL

ABS: Absolute value.

ADC: Add with carry.

ADCS: Add with carry, setting flags.

ADD (extended register): Add extended and scaled register.

ADD (immediate): Add immediate value.

ADD (shifted register): Add optionally-shifted register.

ADDG: Add with tag.

ADDPT: Add checked pointer.

ADDS (extended register): Add

Operation

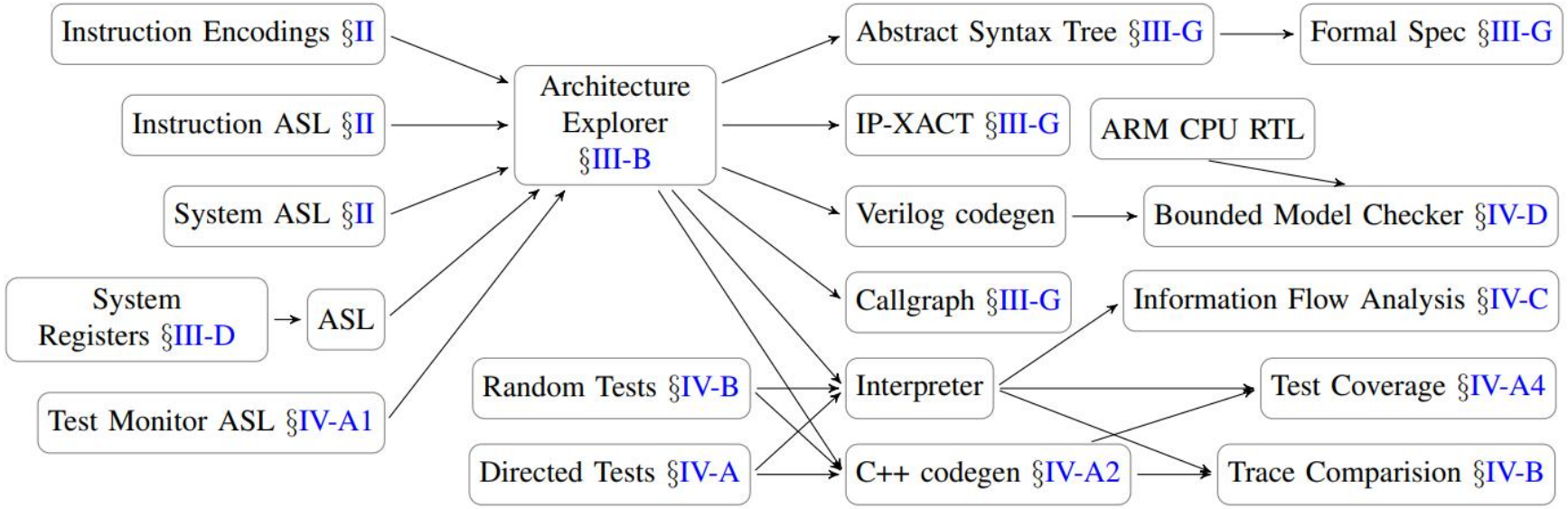
```
constant bits(datasize) operand1 = if n == 31 then SP[datasize] else X[n, datasize];
constant bits(datasize) operand2 = ExtendReg(m, extend_type, shift, datasize);
bits(datasize) result;

(result, -) = AddWithCarry(operand1, operand2, '0');

if d == 31 then
    SP[64] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

https://developer.arm.com/documentation/ddi0602/2025-03/Base-Instructions

ARM ASL



	ARMv8-A				ARMv8-M	
	AArch32	AArch64	Shared	Support	Spec	Support
Instrs.	18318	5757			4998	
Integer	23		352		246	
Float Point			1179		953	76
Exceptions	1474	1611	235		781	
Registers	310	446	398		2011	461
Memory	1584	1169	393		369	481
Debug	675	537	1103			
Instr. Fetch				199	367	128
Test Monitor	-	-	-	1323	-	1893
Misc.	1647	1137	2984	1678	415	1434
Total	24315	10657	5489	3200	9898	4990

(a) Size of ASL specification (lines of code)

	v8-A	v8-M
Registers	586	186
Fields	3951	622
Constant	985	177
Reserved	940	208
Impl. Defined	70	10
Passive	1888	165
Active	68	62
Operations	112	10

(b) Size of System Register specification

ISS acceleration

GenC

From High Level Architecture Descriptions
to Fast Instruction Set Simulators

```
1  AC_ISA(arm)
2  {
3      ac_format Type_DPI1    = "%cond:4 %op!:3 %func1!:4 %s:1 %rn:4 %rd:4 %shift_amt:5 \
4                                %shift_type:2 %subop1!:1 %rm:4";
5      ac_format Type_MBXBLX = "%cond:4 %op!:3 %func1!:4 %s:1 0xfff:12 %subop2!:1      \
6                                %func2!:2 %subop1!:1 %rm:4";
7      ...
8      ac_instr<Type_DPI1> and1, eor1, sub1 ... ;
9      ac_instr<Type_MBXBLX> bx, blx2;
10     ...
11
12     ISA_CTOR(armv5e) {
13
14     and1.set_decoder(op=0, subop1=0, func1=0);                                    ①
15     and1.set_behaviour(and1);
16     and1.set_asm("and%[cond]%sf %reg, %reg, %reg", cond, s, rd, rn, rm, ...);     ②
17
18
19     bx.set_decoder(op=0,subop1=1,subop2=0,func1=0x9,s=0,func2=0);
20     bx.set_behaviour(bx);
21     bx.set_asm("bx%[cond] %reg", cond, rm);
22     bx.set_end_of_block();                                                        ③
```

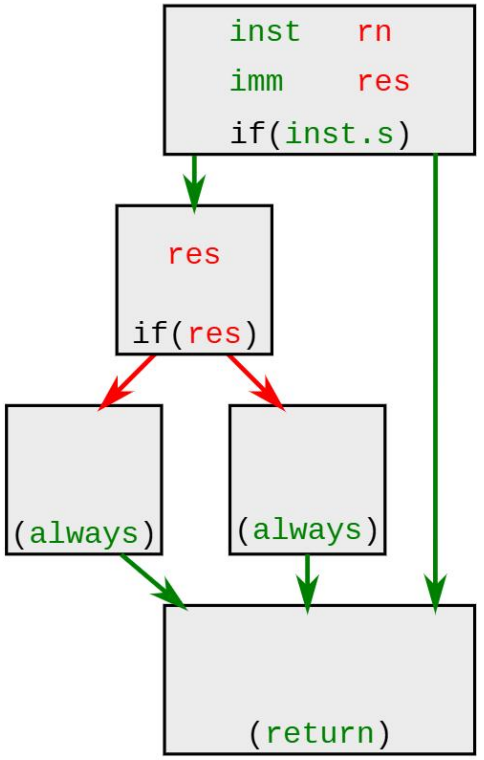
```
1  uint32 pc_check(uint8 reg_index) ...
2  uint32 decode_imm(uint8 type, uint8 shft, uint32 val, uint8 c_i, uint8 &c_o) ...   ①
3  void update_ZN_flags(uint32 value) ...
4
5  execute(and1)
6  {
7      uint32 val;                                                                    ②
8      uint32 imm32;
9      uint32 decode_input = read_register_bank(RB, inst.rm) + pc_check(inst.rm);     ③
10     uint8 carry_in = read_register(C);
11     uint8 c;
12     imm32 = decode_imm(inst.shift_type, inst.shift_amt, decode_input, carry_in, c);④
13     uint32 src1m = read_register_bank(RB, inst.rn) + pc_check(inst.rn);
14     val = src1m & imm32;
15     if(inst.s)                                                                     ⑤
16     {
17             update_ZN_flags(val);
18                 write_register(C, c);
19     }
20     write_register_bank(RB, inst.rd, val);                                         ⑥
21 }
```

https://era.ed.ac.uk/bitstream/handle/1842/14162/Wagstaff2015.pdf?sequence=1&isAllowed=y

1. Fixedness Analysis

```
1 uint32 imm = ror(inst.imm, inst.rotate);
2 uint32 rn = read_register(inst.rn)
3 uint32 res = rn + imm;
4 write_register(inst.rd, res);
5 if(inst.s) {
6     if(res) write_Z(0);
7     else write_Z(1);
8 }
```

(a) A simple instruction semantic description for an add immediate instruction



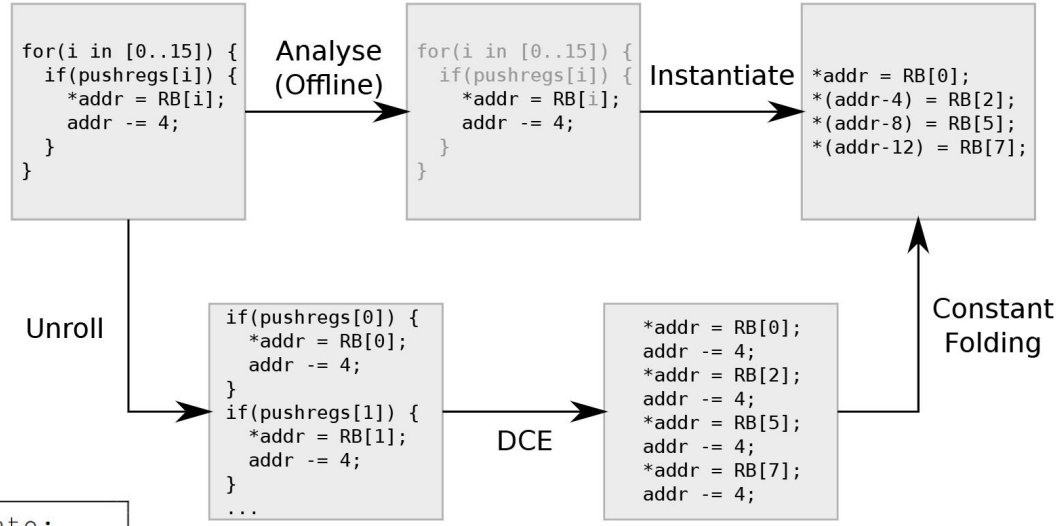
(b) Control Flow Graph for this description

```
1:  function InsnImplFixedness(action)
2:      for all b ← BB ∈ action do
3:          b.dyn_in ← []
4:          b.dyn_out ← []
5:          b.ctrlflow ← invalid
6:          b.mark_variable_accesses_as_fixed()
7:      wl ← [action.entry_block]
8:      while wl is not empty do
9:          b ← wl.pop_front()
10:         result ←BBFixedness(b)
11:         if result = False then
12:             wl.insert(b.successors)
13: function BBFixedness(block)
14:     for all p ← BB ∈ block.predecessors do
15:         if p.ctrlflow = dynamic ∨ ¬p.final_stmt.is_fixed then
16:             block.ctrlflow ← dynamic
17:         block.dyn_in ← block.dyn_in ∪ p.dyn_out
18:     dyn_now ← block.dyn_in
19:     for all s ← Statement ∈ block.statements do
20:         if s writes a dynamic value to a variable v then
21:             dynamic_now ← v
22:         if s reads a variable in dyn_now then
23:             mark s as dynamic
24:     block.dyn_out ← dyn_now
25:     if block.ctrlflow changed ||dyn_now ≠ dyn_in then
26:         return False
27:     return True
```

Figure 4.12: Computing 'fixedness' of variable modifying statem
mentation.

2. GenC Jit warm up

Generating LLVM Bitcode For An Instruction




```
1 %1 = sub i32 32, i32 %ror
2 %2 = shl i32 %imm, i32 %1
3 %3 = shr i32 %imm, i32 %ror
4 %4 = or  i32 %2, i32 %3
5 %5 = load i32* %r0_ptr
6 %6 = sub i32 %5, i32 %4
7 store i32* %r0_ptr, i32 %6
```

(a) LLVM bitcode emitted by a Naïve JIT. The LLVM bitcode must be optimised using expensive analysis and transformation passes, otherwise the rotated immediate is computed each time the instruction executes.

```
1 uint32_t imm_l_shift = 32 - rotate;
2 uint32_t imm_l = imm << imm_l_shift;
3 uint32_t imm_r = imm >> rotate;
4 uint32_t imm_val = imm_l | imm_r;
```

(b) A *partial evaluation* JIT knows to compute the value at JIT time...

```
1 %5 = load %r0_ptr
2 %6 = sub %5, 0xa000000a
3 store %r0_ptr, %6
```

(c) ... and then emit LLVM code using the computed value

Pydgin

Pydgin，一个基于 meta-tracing 和 PyPy 的 ISS
仅模拟用户模式执行

- 省略了地址转换
- 自修改代码
- 中断处理
- 使用 Pydgin 架构描述语言，该语言缺乏 Sail 的抽象级别。

Pydgin: a (Py)thon (D)SL for (G)enerating (In)struction set simulators.

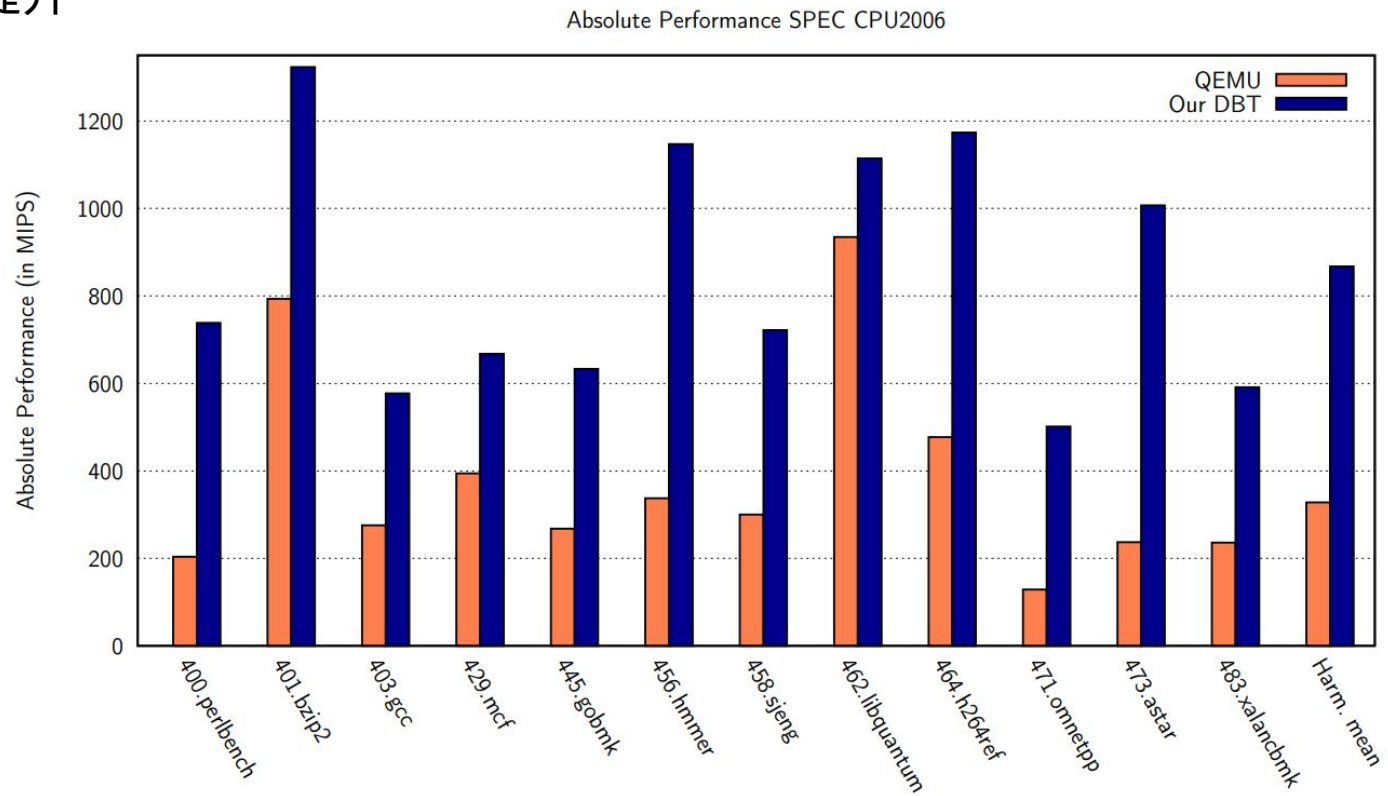
```
144     while s.running:
145       jitdriver.jit_merge_point(
146         pc        = s.fetch_pc(),
147         max_insts = max_insts,
148         state     = s,
149         sim       = self,
150       )
151       # constant-fold pc and mem
152       pc  = hint( s.fetch_pc(), promote=True )
153       old = pc
154       # we use trace elidable iread instead of just read
155       inst_bits = mem.iread( pc, 4 )
156       inst, exec_fun = self.decode( inst_bits )
157
158       self.pre_execute()
159       exec_fun( s, inst )
160       self.post_execute()
161
162       if s.fetch_pc() < old:
163         jitdriver.can_enter_jit(
164           pc        = s.fetch_pc(),
165           max_insts = max_insts,
166           state     = s,
167           sim       = self,
168         )
```

```
160   def execute_addi( s, inst ):
161     s.rf[ inst.rd ] = sext_xlen( s.rf[inst.rs1] + inst.i_imm )
162     s.pc += 4
163
```

GenSim

GenSim 工具链使用 GenC 规范语言和 Captive 动态二进制翻译器（DBT）
实现了相对于 QEMU 的 2 倍 的性能提升

- 硬件加速的客户机内存地址转换
- partial evaluation
- 针对 ISS 工作负载优化的 DBT
- 并行化
- 使用具有性能导向构造的 DSL



https://dl.acm.org/doi/pdf/10.1145/2597809.2597810

2. Sail and RISC-V Sail model

SAIL 语言

Sail 是 ML 家族中的一种领域特定语言 (DSL) [30, 42]，旨在促进处理器的规范和验证。

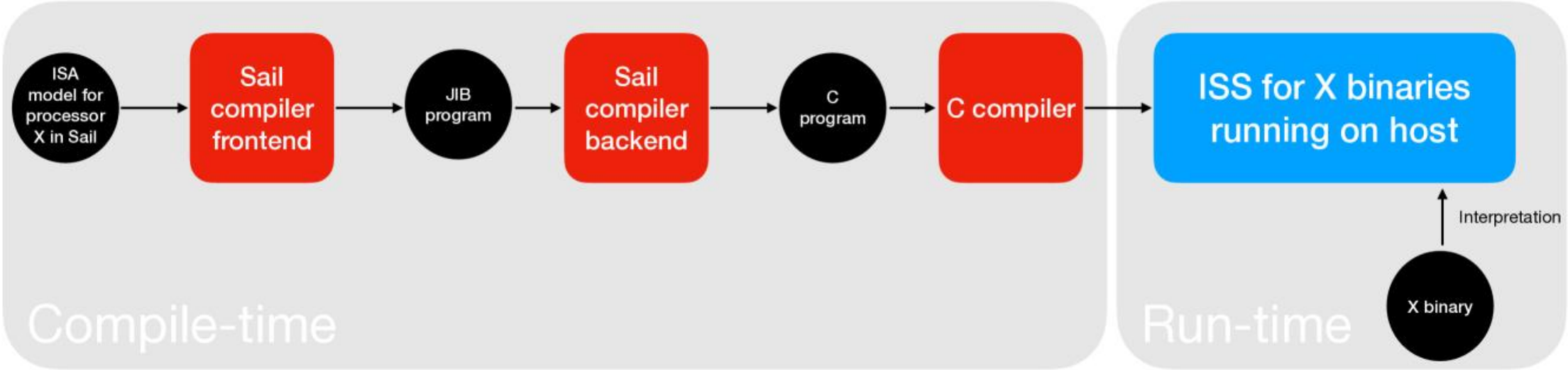
Sail 将对有状态编程的一级支持与丰富的类型系统相结合，包括代数类型、单例类型、依赖类型和高阶类型，以及领域特定的任意精度比特位向量类型，以及寄存器构造。除了通过定理证明实现处理器规范和验证的主要目标外，Sail 与其他 ML 家族语言的最显著变化是：

- Sail 将函数定义限制为一阶，消除了高阶函数。
- Sail 支持 liquid types (作为一种依赖类型，在表达能力和可判定类型推断之间取得了平衡。它们对于解决现代处理器架构中固有的位宽参数化问题非常有用)

RISC-V ISA 支持 32 位 (RV32) 和 64 位 (RV64) 变体，其中大多数 ISA 规范在位宽方面保持参数化，由类型声明 `xlen` 表示。传统的 ADL 通常缺乏对位宽通用数据结构的强类型系统支持，而是诉诸于编译时元编程技术，如 C 预处理器或 C++ 模板

Sail 的多阶段编译管道

从 Sail 到作为静态二进制运行的 ISS



基于 Sail 的 RISC-V 模型

抽象地说，SAIL MODEL 的顶层函数是一个大循环
在计算机架构中称为取指-解码-执行循环。

```
initialise_processor()
pc = 0
while true:
 cmd = memory[pc]
 pc += 4
 match decode(cmd):
 case addi(rd, rs1, imm):
                execute_addi(rd, rs1, imm)
 case xori(rd, rs1, imm):
                execute_xori(rd, rs1, imm)
                ...
 case _:
                ... # Illegal instruction
```

Sail implementation of `addi`

```
function clause execute (ITYPE (imm, rs1, rd, op)) = {
    let rs1_val = X(rs1);
    let immext : bits(64) = sign_extend(imm);
    let result : bits(64) = match op {
        RISCV_ADDI => rs1_val + immext,
        RISCV_SLTI => zero_extend(bool_to_bits(rs1_val <_s immext)),
        RISCV_SLTIU => zero_extend(bool_to_bits(rs1_val <_u immext)),
        RISCV_ANDI => rs1_val & immext,
        RISCV_ORI => rs1_val | immext,
        RISCV_XORI => rs1_val ^ immext
    };
    X(rd) = result;
    RETIRE_SUCCESS
}
```

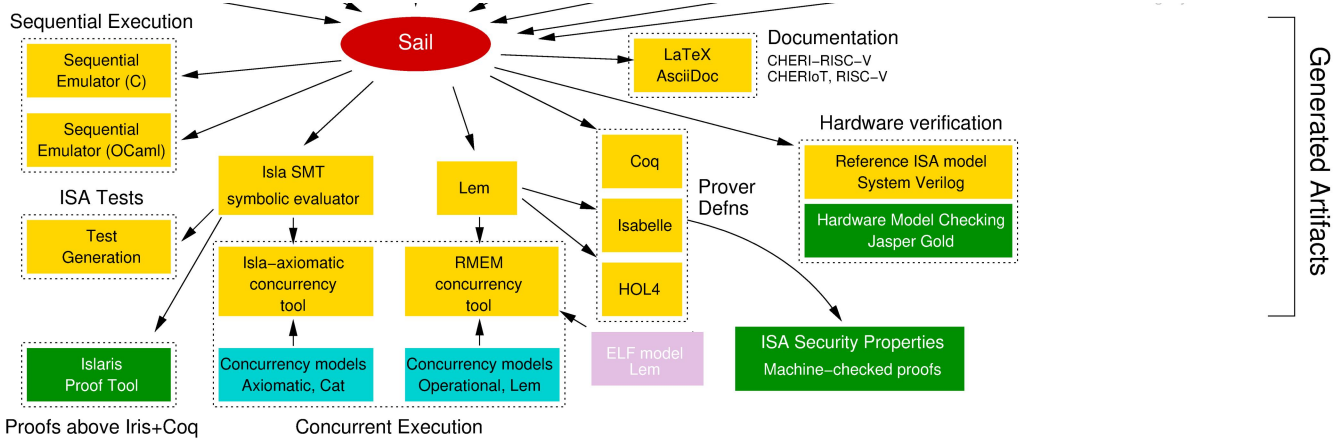
1 2 3 4 4 5

代码优雅地简单明了。然而，这种简单性掩盖了复杂性

```
forall 'n 'm, 'm ≥ 'n. (implicit('m), bits('n)) → bits('m)
```

Sail 优缺点

- Sail 提供了强大的编译时保证
- Sail 支持丰富的编译目标



Sail 性能较差，在 AMD Ryzen 7 PRO 7840U，32 GB 内存的测试环境下

- Sail 每秒执行不到一百万条 RISC-V 指令
- 而 QEMU 在同样的时间内模拟了几亿条指令

生成 ISS 代码

```
{
  uint64_t imm = ...;
  ...
  // let rs1_val = X(rs1);
  uint64_t rs1_val;
  rs1_val = rX_bits(rs1);
  // let immext : xlenbits = sign_extend(imm);
  uint64_t immext;
  {
    i_generic i_generic_64;
    CREATE(i_generic)(&i_generic_64);
    CONVERT_OF(i_generic, int64_t)(&i_generic_64, INT64_C(64));
    bv_generic imm_bv_generic;
    CREATE(bv_generic)(&imm_bv_generic);
    CONVERT_OF(bv_generic, fbits)(&imm_bv_generic, imm, UINT64_C(12), true);
    bv_generic immext_bv_generic;
    CREATE(bv_generic)(&immext_bv_generic);
    sign_extend(&immext_bv_generic, i_generic_64, imm_bv_generic);
    immext = CONVERT_OF(fbits, bv_generic)(immext_bv_generic, true);
    KILL(bv_generic)(&immext_bv_generic);
    KILL(bv_generic)(&imm_bv_generic);
    KILL(i_generic)(&i_generic_64);
  }
  uint64_t result;
  {
  __label__ case_8084, finish_match_8083;
    {
      if ((RISCV_ADDI != op))
        goto case_8084;
      // rs1_val + immext
      result = rs1_val + immext;
      goto finish_match_8083;
    }
  case_8084:;
     ... // the other cases
  }
  {
    wX_bits(rd, result);
  }
  cbz31056 = RETIRE_SUCCESS;
  goto finish_match_8062;
}
```

```
mapping clause encdec = ITYPE(imm, rs1, rd, op)
  <-> imm @ encdec_reg(rs1) @ encdec_iop(op) @ encdec_reg(rd) @ 0b0010011

function clause execute (ITYPE (imm, rs1, rd, op)) = {
  let immext : xlenbits = sign_extend(imm);
  X(rd) = match op {
    RISCV_ADDI  => X(rs1) + immext,
    RISCV_SLTI  => zero_extend(bool_to_bits(X(rs1) <_s immext)),
    RISCV_SLTIU => zero_extend(bool_to_bits(X(rs1) <_u immext)),
    RISCV_ANDI  => X(rs1) & immext,
    RISCV_ORI   => X(rs1) | immext,
    RISCV_XORI  => X(rs1) ^ immext
  };
  RETIRE_SUCCESS
}
```

生成 ISS 代码

```
{
  uint64_t imm = ... ;
  ...
  // let rs1_val = X(rs1);
  uint64_t rs1_val;
  rs1_val = rX_bits(rs1);
  // let immext : xlenbits = sign_extend(imm);
  uint64_t immext;
  {
    i_generic i_generic_64;
    CREATE(i_generic)(&i_generic_64);
    CONVERT_OF(i_generic, int64_t)(&i_generic_64, INT64_C(64));
    bv_generic imm_bv_generic;
    CREATE(bv_generic)(&imm_bv_generic);
    CONVERT_OF(bv_generic, fbits)(&imm_bv_generic, imm, UINT64_C(12), true);
    bv_generic immext_bv_generic;
    CREATE(bv_generic)(&immext_bv_generic);
    sign_extend(&immext_bv_generic, i_generic_64, imm_bv_generic);
    immext = CONVERT_OF(fbits, bv_generic)(immext_bv_generic, true);
    KILL(bv_generic)(&immext_bv_generic);
    KILL(bv_generic)(&imm_bv_generic);
    KILL(i_generic)(&i_generic_64);
  }
  uint64_t result;
  {
  __label__ case_8084, finish_match_8083;
    {
      if ((RISCV_ADDI != op))
        goto case_8084;
      // rs1_val + immext
      result = rs1_val + immext;
      goto finish_match_8083;
    }
  case_8084:;
     ... // the other cases
  }
  {
    wX_bits(rd, result);
  }
  cbz31056 = RETIRE_SUCCESS;
  goto finish_match_8062;
}
```

```
mapping clause encdec = ITYPE(imm, rs1, rd, op)
  <-> imm @ encdec_reg(rs1) @ encdec_iop(op) @ encdec_reg(rd) @ 0b0010011

function clause execute (ITYPE (imm, rs1, rd, op)) = {
  let immext : xlenbits = sign_extend(imm);
  X(rd) = match op {
    RISCV_ADDI  => X(rs1) + immext,
    RISCV_SLTI  => zero_extend(bool_to_bits(X(rs1) <_s immext)),
    RISCV_SLTIU => zero_extend(bool_to_bits(X(rs1) <_u immext)),
    RISCV_ANDI  => X(rs1) & immext,
    RISCV_ORI   => X(rs1) | immext,
    RISCV_XORI  => X(rs1) ^ immext
  };
  RETIRE_SUCCESS
}
```

生成 ISS 代码

```
628  void sign_extend(lbits *rop, const lbits op, const sail_int len)
629  {
630    assert(op.len <= (uint64_t) len);
631    rop->len = (uint64_t) len;
632    if(mpz_tstbit(*op.bits, op.len - 1)) {
633      mpz_set(*rop->bits, *op.bits);
634      for(mp_bitcnt_t i = rop->len - 1; i >= op.len; i--) {
635        mpz_setbit(*rop->bits, i);
636      }
637    } else {
638      mpz_set(*rop->bits, *op.bits);
639    }
640  }
```

```
1  fn execute(mergez3var) {
2  ...
3   imm : %bv12
4   imm = ...
5   // let rs1_val = X(rs1);
6   rs1_val : %bv64
7   rs1_val = rX_bits(rs1)
8   // let immext : xlenbits = sign_extend(imm);
9   immext : %bv64
10  sail_int_64 : %i_generic
11  sail_int_64 = i64_to_i_generic(64) // cast
12  imm_bv_generic : %bv_generic
13  imm_bv_generic = imm // cast
14  immext_bv_generic : %bv_generic
15  immext_bv_generic = sign_extend(sail_int_64, imm_bv_generic)
16  immext = immext_bv_generic // cast
17 ...
```

int(64), bits(12) -> bits(64)

int(64)-> i_generic

bits(12)-> bv_generic

int_generic, bv_generic -> bv_generic

bv_generic -> bits(64)

- bv_generic, i_generic 都是堆分配的数据结构
- bv_generic 和 i_generic 都是使用 GNU 多精度算术库（GMP）实现，用于任意精度算术

Sail 开销

- **Bitvectors**：`uint64_t` 在函数调用中可能退化为 `i_generic` 和 `bv_generic` 例如 `sign_extend` 。所有基于 GMP 的位向量的操作都需要昂贵的堆分配，因为 GMP 整数类型总是堆分配的。
- **Integers**：当 Sail 无法证明有符号 64 位整数适合 `int64_t` 时，可能退化为 `i_generic`，导致每个整数操作中都需要堆分配。
- **Interpretation overhead**：Sail 生成一个解释器，它必须不断重复分析正在模拟的程序的各个部分。而实际处理器使用缓存来减轻这种开销，例如指令缓存。

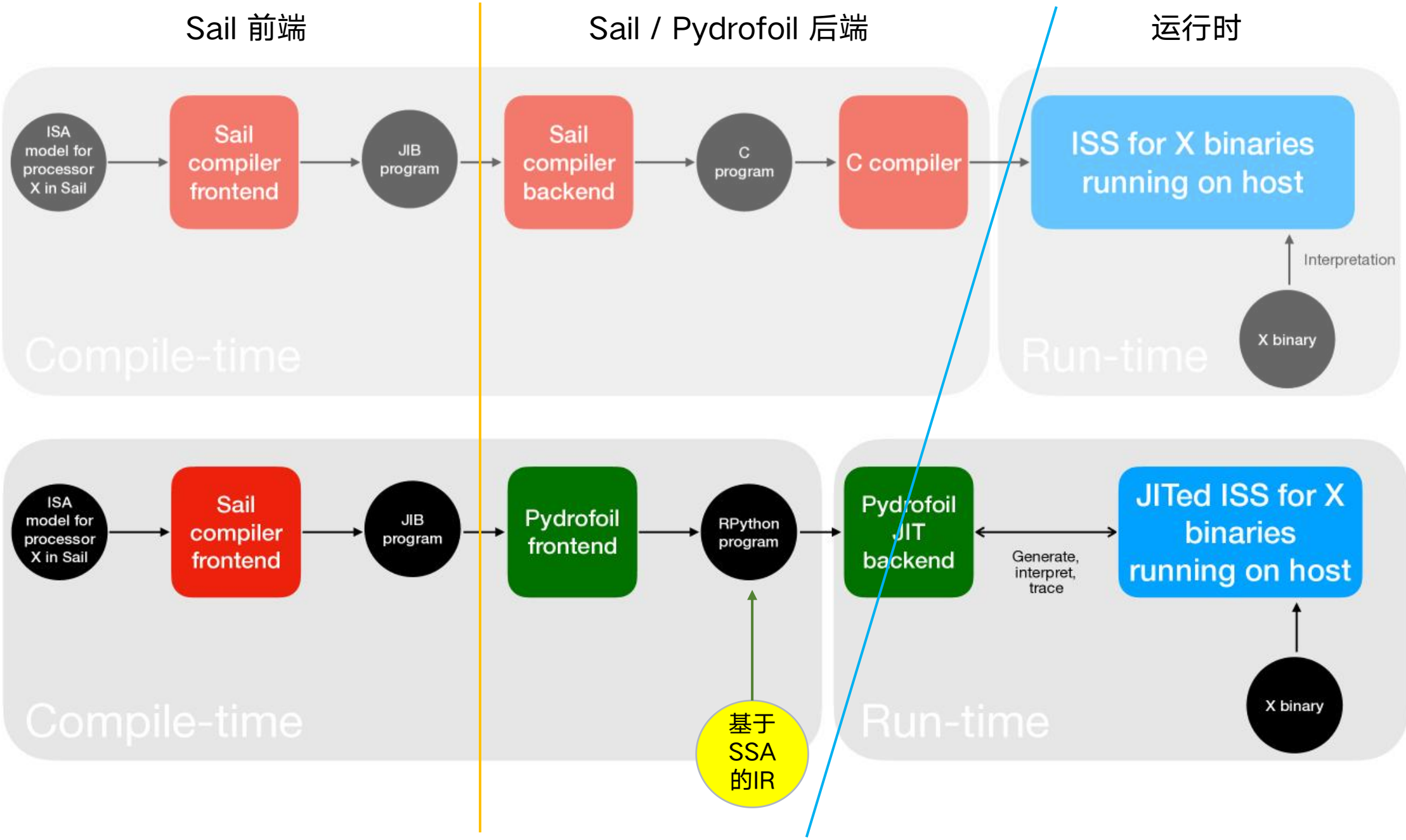
3. How is Pydrofoil implemented?

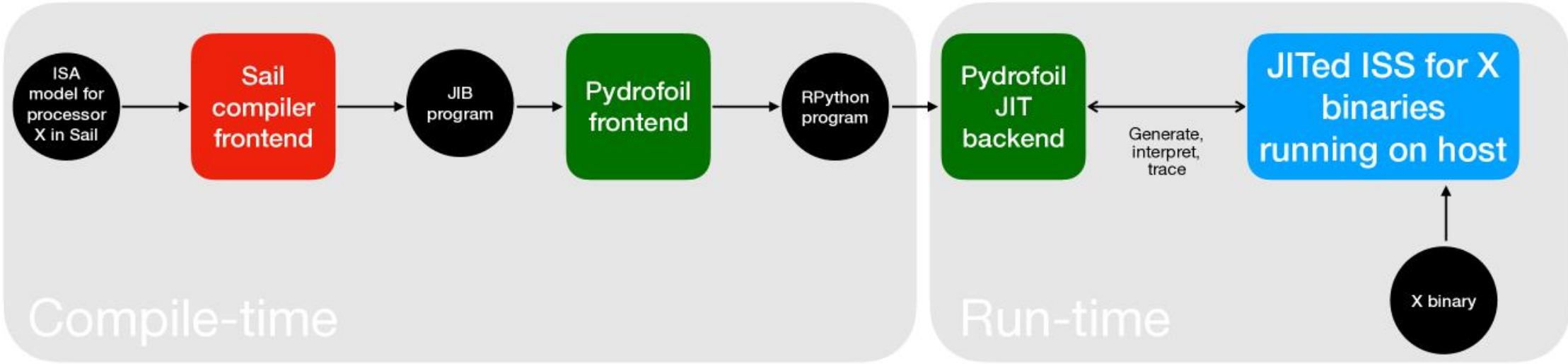
3.1 Pydrofoil 架构

Pydrofoil 是使用 PyPy 框架实现的。PyPy（/ˈpaɪpaɪ/）是一种Python编程语言实现

RPython 是 PyPy 的实现语言。RPython（Restricted Python）是 Python2 的一个静态类型子集。

该语言的开发旨在帮助实现动态语言的高性能虚拟机。
作为 ISA 模型，无论是用 Sail 编写还是其他方式，本质上都是 ISA 的解释器，因此合理地预期 RPython 元追踪 JIT 可以加速 Sail 解释器循环，并将来宾机器指令编译为宿主机器指令。

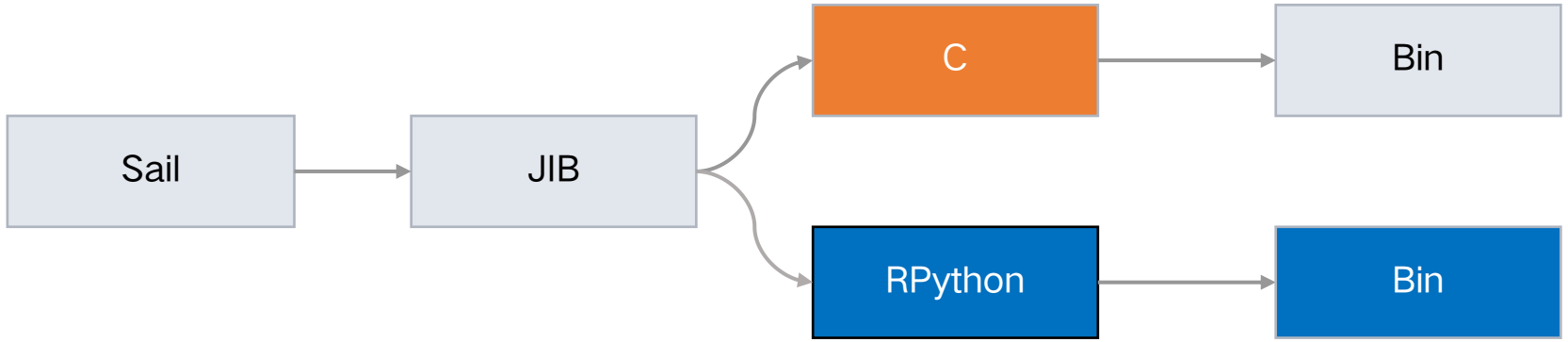

Sail 前端
Sail / Pydrofoil 后端
运行时
ISA model for processor X in Sail
Sail compiler frontend
JIB program
Sail compiler backend
C program
C compiler
ISS for X binaries running on host
Interpretation
X binary
Compile-time
Run-time
ISA model for processor X in Sail
Sail compiler frontend
JIB program
Pydrofoil frontend
RPython program
Pydrofoil JIT backend
Generate, interpret, trace
JITed ISS for X binaries running on host
X binary
基于SSA的IR
Compile-time
Run-time




```
fn execute(mergez3var) {
...
 imm : %bv12
 imm = ...
 // let rs1_val = X(rs1);
 rs1_val : %bv64
 rs1_val = rX_bits(rs1)
 // let immext : xlenbits = sign_extend(imm);
 immext : %bv64
 sail_int_64 : %i_generic
 sail_int_64 = i64_to_i_generic(64) // cast
 imm_bv_generic : %bv_generic
 imm_bv_generic = imm // cast
 immext_bv_generic : %bv_generic
 immext_bv_generic = sign_extend(sail_int_64, imm_bv_generic)
 immext = immext_bv_generic // cast
...
```

- 当 JIB 被生成时，所有 Sail 的类型检查都已完成所有变量都有明确的类型
  - 所有模式匹配都被编译为基本块和（条件）跳转
  - 对 Pydrofoil 的提前编译（AOT）编译器容易处理

性能优化思路



- Sail 代码静态优化（特别是针对位向量和整数的优化，旨在最小化堆分配）
  - （理论上，也可以用于 Sail 的 C 后端。）

- 利用运行时信息，通过 RPython 跟踪 JIT 编译器进行进一步的优化，这些优化在静态情况下不可用。

3.2 Static Pydrofoil IR optimisations

IR 应用了一系列标准编译器优化：常量折叠、死代码消除、公共子表达式消除、内联和聚合的标量替换（使用一种简单的部分逃逸分析形式）

Static Optimisations of bitvector and integer operations

- **Bitvectors**: `uint64_t` 在函数调用中可能退化为 `i_generic` 和 `bv_generic`
  - 如果 JIB 可以推断出精确的位宽，则使用固定宽度的位向量类型，如 `bv64` 或 `bv16`
- **Integers**：当 Sail 无法证明有符号 64 位整数适合 `int64_t` 时，可能退化为 `i_generic`
  - 如果 JIB 可以推断出一个整数始终可以适应 64 位有符号机器整数，则使用 `%i64` 。否则，使用 `i _ generic`

1. 静态化 / 函数调用重写

```
# functions

# nothing is known statically
sign_extend: (%bv_generic, %i_generic) -> %bv_generic
# the target width is known to fit into an i64
sign_extend_g_i: (%bv_generic, %i64) -> %bv_generic
# the target width is a constant
sign_extend_g_c<n>: (%bv_generic) -> %bv<n>
# the argument bitvector has a constant width
sign_extend_bv_c<n, m>: (%bv<n>) -> %bv<m>
```

```
# rewrite rules
sign_extend(b, i64_to_i_generic(i)) -> sign_extend_g_i(b, i)
sign_extend_g_i(b, c) -> sign_extend_g_c<c>(b) [if c <= 64]
sign_extend_g_c<c>(cast(b:%bv<n>, %bv_generic)) -> sign_extend_bv_c<n, c>(b, n, c)
```

优化效果

```
1  fn execute(mergez3var) {
2  ...
3   imm : %bv12
4   imm = ...
5   // let rs1_val = X(rs1);
6   rs1_val : %bv64
7   rs1_val = rX_bits(rs1)
8   // let immext : xlenbits = sign_extend(imm);
9   immext : %bv64
10  sail_int_64 : %i_generic
11  sail_int_64 = i64_to_i_generic(64) // cast
12  imm_bv_generic : %bv_generic
13  imm_bv_generic = imm // cast
14  immext_bv_generic : %bv_generic
15  immext_bv_generic = sign_extend(sail_int_64, imm_bv_generic)
16  immext = immext_bv_generic // cast
17 ...
```

```
1  fn execute(mergez3var) {
2  ...
3   imm : %bv12
4   imm = ...
5   // let rs1_val = X(rs1);
6   rs1_val : %bv64
7   rs1_val = rX_bits(rs1)
8   // let immext : xlenbits = sign_extend(imm);
9   immext : %bv64
10  immext = sign_extend_bv_c<12, 64>(imm)
11 ...
```

2. Interaction with inlining

```
1  val operator <_s : forall 'n, 'n > 0. (bits('n), bits('n)) -> bool
2
3  function operator <_s (x, y) = signed(x) < signed(y)
```

函数内联后，参数的具体位向量宽度通常可以从调用上下文中推断出来

```
1  val (operator <_s): (%bv_generic, %bv_generic) -> %bool
2
3  fn (operator <_s)(x, y) {
4   // signed(x) < signed(y)
5   z40 : %i_generic
6   z40 = signed(x)
7   z41 : %i_generic
8   z41 = signed(y)
9   return = lt_int(z40, z41)
10  end;
11 }
```

```
1  jump @neq(RISCV_SLTI, z0) goto 280
2  // RISCV_SLTI => zero_extend(bool_to_bits(rs1_val <_s immext)),
3  z2 : %bv1
4  z6 : %bool
5  z7 : %bv_generic
6  z7 = rs1_val // cast
7  z8 : %bv_generic
8  z8 = immext // cast
9  z6 = (operator <_s)(z7, z8)
10 z2 = bool_to_bits(z6)
11 ...
```

3. Function specialisation

在函数过大而无法内联时，Pydrofoil 会遍历所有无法内敛的函数，包括

- 参数中包含 `%bv_generic` 或 `%i_generic` 类型
- 并且是常量
- 或者是更具体的类型转化而来

对于这样的调用，目标函数将被复制为更具体的版本，接受更具体的参数类型

- 复制后的函数也可以继续被优化，以考虑其参数的新可用位向量宽度。
- 有时，作为特化的结果，函数的返回类型也可以变得更加具体。

什么时候对 bitvector 和 int 的优化失效?

- 位向量和整数操作的重写仅在单个函数内有效。
  - 只有当此类操作的参数在操作发生的同一函数中从更具体的类型转换时，才能应用它们。
- 内联和函数特化有助于将重写限制在单个函数内。
  - 但有时这仍然不足以为窥视孔重写提供足够的上下文以完成它们的工作。
- 在更复杂的情况下，特别是如果位向量操作的参数是从 `union` 或 `struct` 中读取的，则需要更强大的静态分析才能将操作优化为更具体的变体。

```
1 val mem_read : forall 'n, 0 < 'n <= max_mem_access . (AccessType(ext_access_type), physaddr, int('n), bool, bool,
bool) -> MemoryOpResult(bits(8 * 'n))
```

3.3 RPython code generation

Jlb 翻译到 IR

```
32   # concat(concat(x, const1), const2) → concat(x, const1+const2)
33   # example:
34       # i198 = block110.emit(Operation, 'zget_16_random_bits', [UnitConstant.UNIT], SmallFixedBitVector(16), '`12 827:
35       # i199 = block110.emit(Operation, '@bitvector_concat_bv_bv', [SmallBitVectorConstant(0×0, SmallFixedBitVector(8)
36       # i200 = block110.emit(Operation, '@bitvector_concat_bv_bv', [SmallBitVectorConstant(0, SmallFixedBitVector(6)),
37       # i201 = block110.emit(Operation, '@bitvector_concat_bv_bv', [SmallBitVectorConstant(0b10, SmallFixedBitVector(2
```

Jib 翻译到 RPython 源代码

```
class __extend__(pairtype(types.SmallFixedBitVector, types.GenericBitVector)):
    def convert((from_, to), expr, codegen):
        assert from_.width <= 64
        return "bitvector.from_ruint(%s, %s)" % (from_.width, expr)
```

IR 基本块跳转 -> 基于 PC 的无限循环

```
    # Assignment(result='zgsz34381_lz310', value=Var(name='false'))
    zgsz34381_lz310 = False
    pc = 16
if pc == 16:
    # Assignment(result='zgaz32649_lz32', value=Var(name='zgsz34381_lz310'))
    zgaz32649_lz32 = zgsz34381_lz310
    if zgaz32649_lz32:
        # inline pc=36
        # LocalVarDeclaration(name='zgsz34386_lz33', typ=NamedType('%i'), value=Var(name='zwidth'))
        # zgsz34386_lz33: NamedType('%i')
        zgsz34386_lz33 = Integer.fromint(zwidth)
        # Operation(args=[Var(name='zt'), Var(name='zpaddr'), Var(name='zgsz34386_lz33')], name='zhtif_load', result='return')
        return_ = func_zhtif_load(zt, zpaddr, zgsz34386_lz33)
        pc = 38
        continue
```

3.3 RPython code generation

```
CF Bolz-Tereick, 6个月前 | 2 authors (CF Bolz-Tereick
 84 > class BitVector(W_Root):···
176
177
CF Bolz-Tereick, 上个月 | 3 authors (naseweisssss and
178 > class SmallBitVector(BitVector):··
```

```
...
708 > class GenericBitVector(BitVector):···
1177
1178
...
1179 > class Integer(object):···
1242
...
1243 > class SmallInteger(Integer):···
1490
...
1491 > class BigInteger(Integer):···
1907
```

3.4 Adding a JIT compiler with RPython

- meta-tracing JIT 标记哪里是核心循环，以及 PC 程序计数器等
- JIT 在运行时会进一步优化 bitvector 和 interger 操作
- 通过逃逸分析移除部分堆分配

```
708  class GenericBitVector(BitVector):
782      @jit.unroll_safe
783      def _add_ruint(self, othervalue):
784          resdata = [r_uint(0)] * len(self.data)
785          for i, value in enumerate(self.data):
786              res = value + othervalue
787              resdata[i] = res
788              othervalue = r_uint(res < value)
789          return self.make(resdata, True)
```

```
54  @always_inline
55  @unwrap("o o o i")
56  def platform_read_mem(machine, read_kind, addr_size, addr, n):
57      assert n <= 8
58      addr = addr.touint()
59      res = jit.promote(machine.g).mem.read(addr, n)
60      return bitvector.SmallBitVector(n*8, res) # breaking abstracting a bit, but much more efficient
61
62  def platform_read_mem_o_i_bv_i(machine, read_kind, addr_size, addr, n):
63      return jit.promote(machine.g).mem.read(addr, n)
64
65  @always_inline
66  def platform_write_mem(machine, write_kind, addr_size, addr, n, data):
67      n = n.toint()
68      assert n <= 8
69      assert addr_size == 64
70      assert data.size() == n * 8
71      jit.promote(machine.g).mem.write(addr.touint(), n, data.touint())
72      return True
```

指令解码

```
170  mapping clause encdec = ITYPE(imm, rs1, rd, op)
171    ⟷ imm @ encdec_reg(rs1) @ encdec_iop(op) @ encdec_reg(rd) @ 0b0010011
172
```

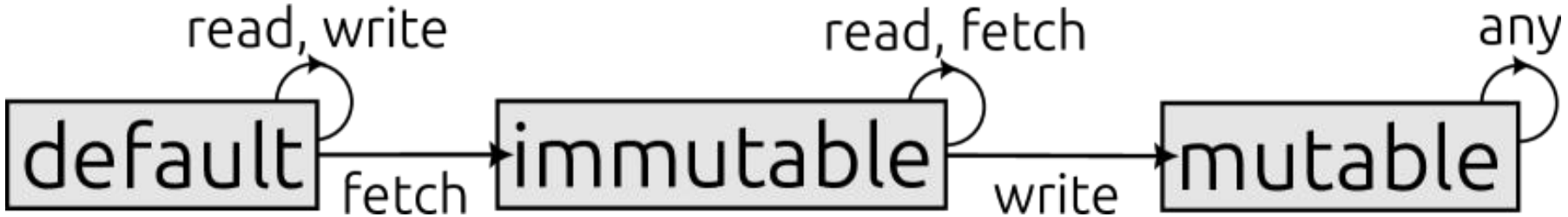
mapping -> match_backwords -> if + goto

Simulator
Elf

取指(IF)
译码(DE)
执行(EXE)
访存(MEM)
回写(WB)

```
...
i106      =      getfield_gc_i(p1, field=s4)
...
# c.addi s4, 0x8
i144                                  = getarrayitem_gc_i(ConstPtr(ptr142), 26)
i146 = int_and(i144, 65535)
i148 = int_and(i144, 3)
i150 = int_eq(i148, 3)
guard_false(i150)
i152 = int_eq(i146, 1)
guard_false(i152)
i154 = uint_rshift(i146, 5)
i156 = int_and(i154, 1)
i158 = uint_rshift(i146, 6)
i160 = int_and(i158, 1)
i162 = int_lshift(i156, 1)
i163 = int_or(i162, i160)
i165 = uint_rshift(i146, 7)
i167 = int_and(i165, 15)
i169 = uint_rshift(i146, 11)
i171 = int_and(i169, 3)
i173 = int_lshift(i171, 2)
i174 = int_or(i173, i163)
i176 = int_lshift(i167, 4)
i177 = int_or(i176, i174)
i178 = int_is_zero(i177)
guard_false(i178)
i180 = uint_rshift(i146, 13)
i181 = int_is_zero(i180)
guard_true(i181)
i182 = int_is_zero(i148)
guard_false(i182)
i184 = int_and(i165, 31)
i186 = uint_rshift(i146, 12)
i188 = uint_rshift(i146, 2)
i190 = int_and(i188, 31)
i192 = int_lshift(i186, 5)
i193 = int_or(i192, i190)
i194 = int_is_zero(i193)
guard_false(i194)
i195 = int_is_zero(i184)
guard_false(i195)
i197 = int_eq(i148, 1)
guard_true(i197)
i199 = int_xor(i193, 32)
i201 = int_sub(i199, 32)
i203 = int_and(i201, 4095)
guard_value(i184, 20)
i206 = int_xor(i203, 2048)
i208 = int_sub(i206, 2048)
i209 = int_add(i106, i208)
i211 = int_add(i2, 2)
i213 = int_add(i0, 2)
setfield_gc(p1, i209, field=s4)
i215 = int_eq(i213, 10000)
guard_false(i215)
# next guest instruction
```

Making the main memory simulation JIT-friendly




```
1 i35 = getfield_gc_i(p1, field=s4)
2 ...
3 # c.addi s4. 0x8
4 i44 = int_add(i35, 8)
5 i64 = int_add(i0, 2)
6 setfield_gc(p1, i44, field=s4)
7 i66 = int_eq(i64, 10000)
8 guard_false(i66)
```

Improving the RPython JIT integer optimisations

```
1 ld t0, 0x8(a0)
2 ld t1, 0x10(a0)
```

- 在处理器中，单个比特通常用于保存权限 (特定地址范围是否允许非对齐访问)
- 在硬件中，这种检查是便宜的。在软件模拟硬件中，一次又一次地检查位是昂贵的。原则上，JIT 可以在跟踪优化期间确定权限位是否在指令之间发生变化。
- 如果没有，除了第一次检查之外的所有检查都可以省略

Improving the RPython JIT integer optimisations

```
1  # ld t0, 0x8(a0) # first load
2  i1 = getfield_gc_i(p1, field=a0)
3  i2 = int_add(i1, 8) # add offset to get the actual pointer
4  i3 = int_and(i2, 7) # alignment check: extract last three bits
5  i4 = int_is_zero(i3) # check that they are zero
6  guard_true(i4) # after this instruction the JIT knows that i1 and i2 are of the form ???...???000
7  ... # do the actual load from memory
8
9  # ld t1, 0x10(a0) # second load
10 i12 = int_add(i1, 16) # must be of the form ???...???000, because both i1 and 16 are
11 i13 = int_and(i12, 7) # alignment check for second load: i13 is 0
12 i14 = int_is_zero(i13) # always 1
13 guard_true(i14) # can be removed by the JIT's opimizer because i14 is constant 1
14 ... # do the actual load from memory
```

Benchmark

评估 Pydrofoil 性能面临挑战

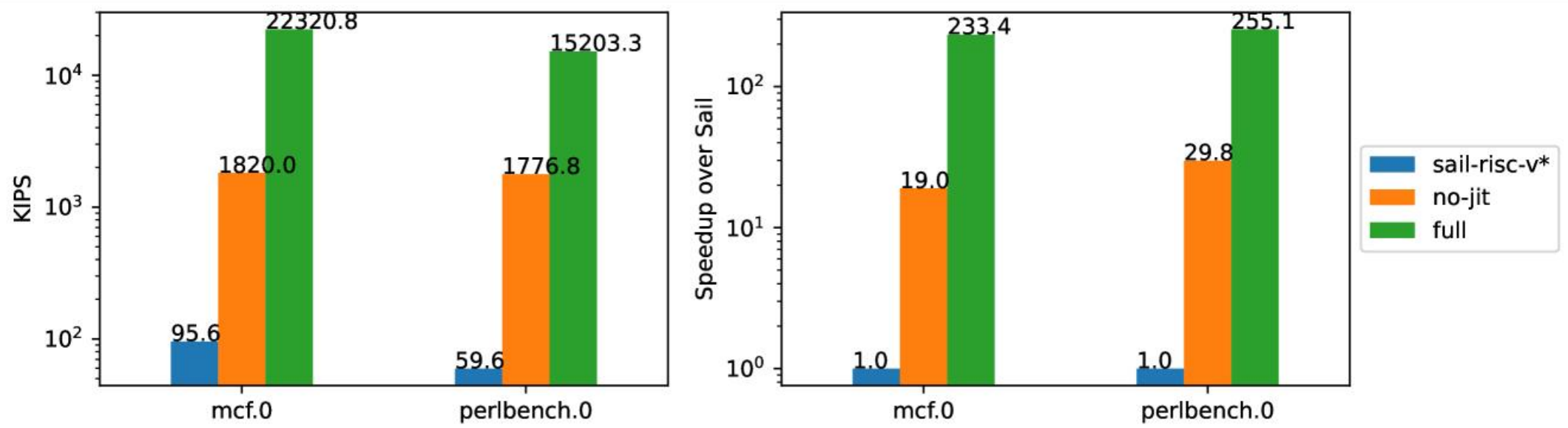
像 Arm 的 FastModel 和 Architecture Envelope Models 这样的专有工具提供了高性能的仿真能力。但是它们的内部架构，包括从 Arm 的 ASL 自动生成的程度，仍然在很大程度上未公开

- 来自 Synopsys （例如，ASIP Designer，Platform Architect，ImperasDV），MachineWare 和 Codasip 的商业工具依赖于手写组件或从较低级别的 ADL（如 SystemC，nML 或 Codal）生成。
- 许多学术 ISS 虽然有价值，但是缺乏更新维护

- 尝试使用 Sail 转换为 GenC 的结果是数千万行，甚至在几天后仍然无法编译
- 尝试转换为 Rust 代码的结果是大量生成的代码导致 Rust 编译器使用了不可持续的内存量
- 目前 Sail 生成的 RV64 C 语言代码量为 30万行

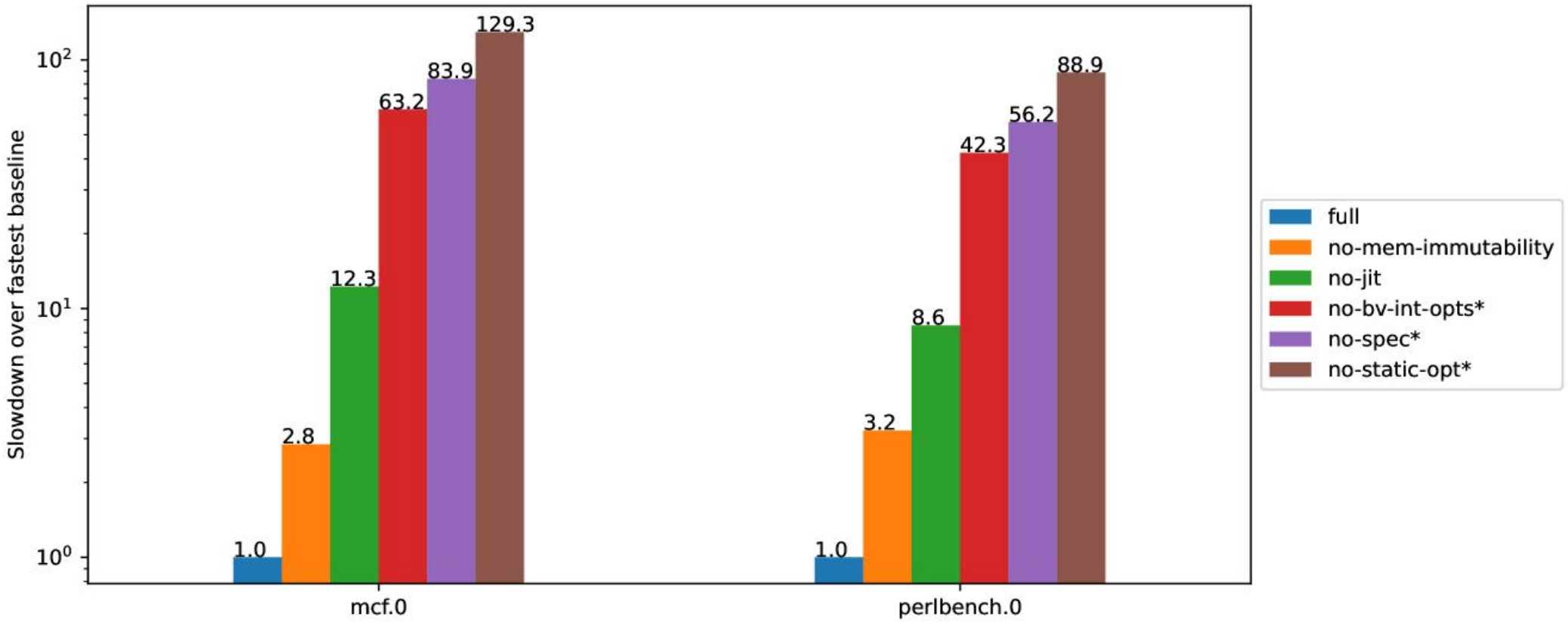
Pydrofoil vs Sail

SPEC, Booting Linux



> This benchmark uses the TaiShan server.

消融实验


Slowdown over fastest baseline
10^2
10^1
10^0
129.3
83.9
63.2
12.3
2.8
1.0
88.9
56.2
42.3
8.6
3.2
1.0
mcf.0
perlbench.0
full
no-mem-immutability
no-jit
no-bv-int-opts*
no-spec*
no-static-opt*

Pydrofoil vs QEMU

QEMU 的核心（JIT）编译引擎 使用手工制作的、特定于指令集架构的翻译器 将来宾指令翻译为主机指令序列，。超过二十年的广泛工程经验为 QEMU 卓越的仿真性能做出了贡献。

QEMU 的性能明显优于 Pydrofoil，比 Pydrofoil 快 2~100 倍，几何平均值为 26.7

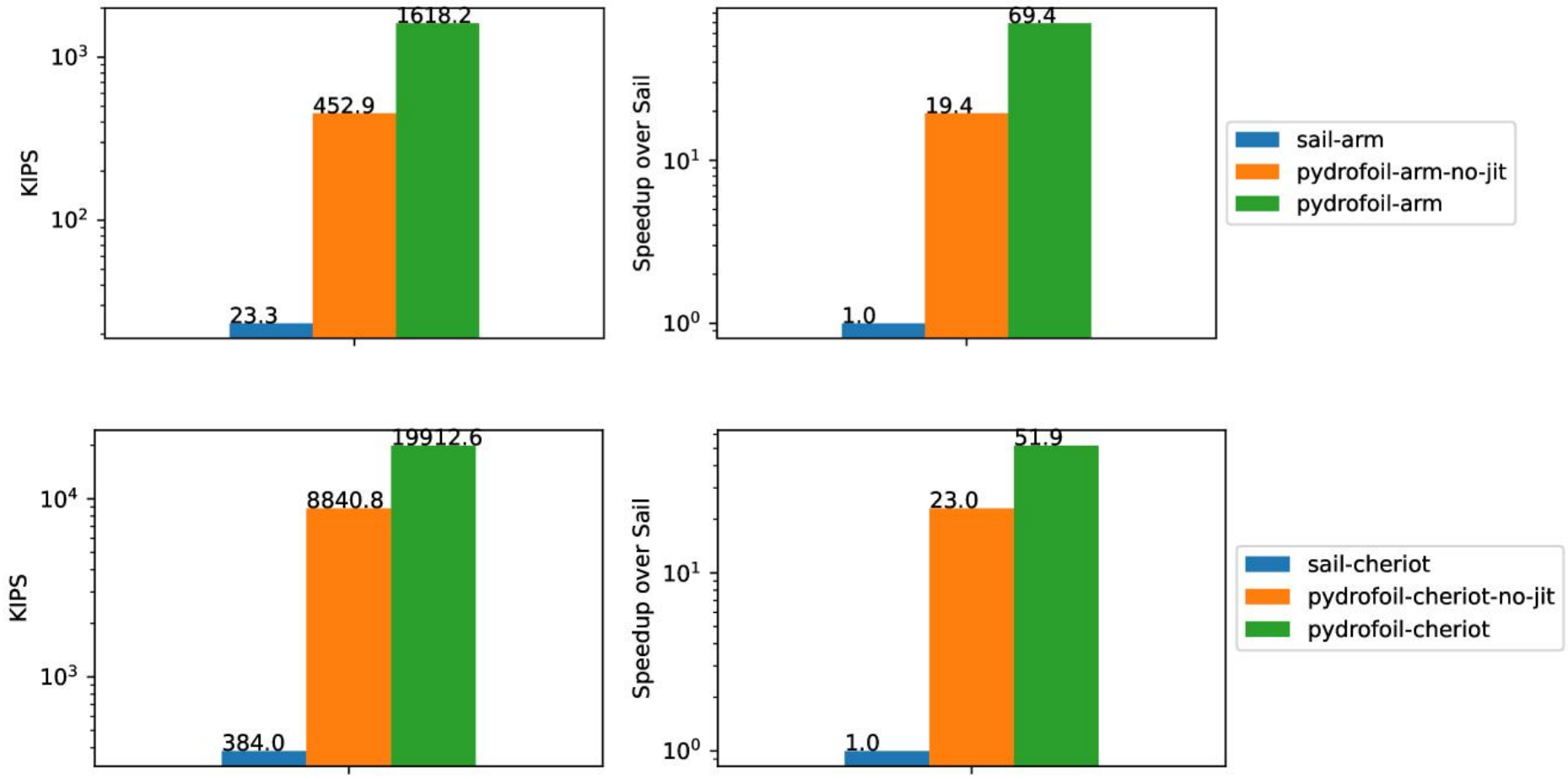
Benchmark	time pydrofoil	time qemu	speedup
600.perlbench_s.0	25462s	1121s	22.7×
600.perlbench_s.1	49089s	760s	64.6×
600.perlbench_s.2	14026s	692s	20.3×
602.gcc_s.0	31757s	993s	32.0×
602.gcc_s.1	35479s	651s	54.5×
602.gcc_s.2	30758s	627s	49.1×
605.mcf_s.0	13248s	6095s	2.2×
620.omnetpp_s.0	58203s	1267s	45.9×
623.xalancbmk_s.0	14044s	3456s	4.1×
625.x264_s.0*	15664s	278s	56.3×
625.x264_s.1*	92698s	947s	97.9×
625.x264_s.2*	69464s	768s	90.5×
631.deepsjeng_s.0	39604s	1749s	22.6×
641.leela_s.0	34928s	1738s	20.1×
648.exchange2_s.0	17023s	1159s	14.7×
657.xz_s.0	49566s	2603s	19.0×
Geometric mean			26.7×

Pydrofoil-Arm vs Pydrofoil-CHERIoT

测试性能数据对 RISC-V 的依赖程度如何

- Arm v9.4-a 规范在 Sail 中是通过从 Arm 的 ASL 规范转译而获得的
  - 通过翻译 Arm 模型获得的 JIB 比相应的 RISC-V 大 20 倍
  - 启动 Linux 6.0.7，但仅到 `init` 进程的初始化。共执行了 2460 万条指令。
- CHERIoT 是一个简单的 32 位 RISC-V 微控制器，没有虚拟内存和缓存，但严格关注安全性
  - 启动 RTOS 需要 490,000 条指令

KIPS 和 加速比

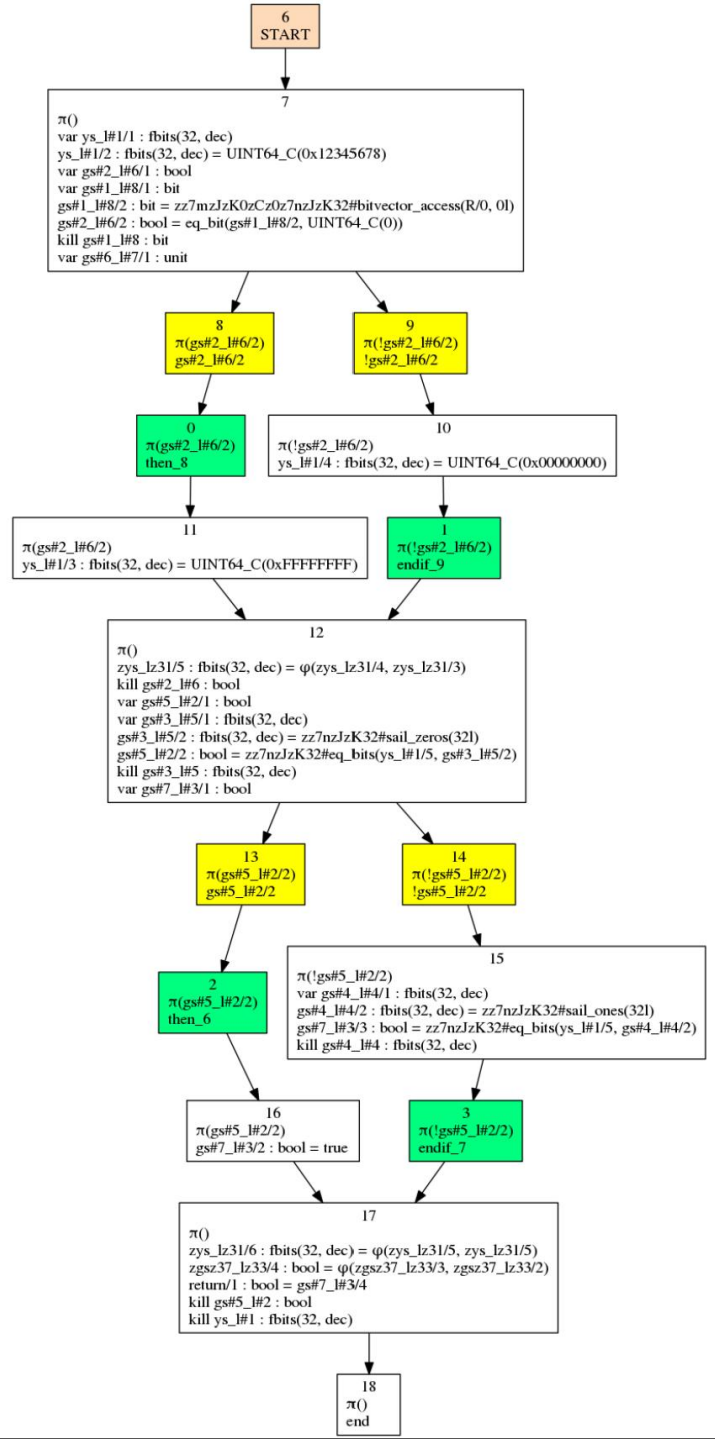

KIPS
10^3
10^2
23.3
452.9
1618.2
Speedup over Sail
10^1
10^0
1.0
19.4
69.4
sail-arm
pydrofoil-arm-no-jit
pydrofoil-arm
KIPS
10^4
10^3
384.0
8840.8
19912.6
Speedup over Sail
10^1
10^0
1.0
23.0
51.9
sail-cheriot
pydrofoil-cheriot-no-jit
pydrofoil-cheriot

Pydrofoil 其他未来方向

- Clocks：处理器时钟的仿真是 ISS 加速中的一种已知低效
- Address translation：真实处理器在翻译后备缓冲区（TLB）中缓存地址翻译的结果
- JIT warmup：推动 PyPy 中 JIT 对 ISS 工作负载得优化
- Type-erasure：Sail 当前会擦出一些类型的强约束

Sail 优化方向

```
var ys#u12_l#9 : fbits(32, dec)
ys#u12_l#9 : fbits(32, dec) = UINT64_C(0x12345678)
var gs#2#u12_l#15 : bool
var gs#1#u12_l#17 : bit
gs#1#u12_l#17 : bit = zz7mzJzK0zCz0z7nzJzK32#bitvector_access(R, 0l)
gs#2#u12_l#15 : bool = eq_bit(gs#1#u12_l#17, UINT64_C(0))
kill gs#1#u12_l#17 : bit
var gs#6#u12_l#16 : unit
jump gs#2#u12_l#15 then_13
ys#u12_l#9 : fbits(32, dec) = UINT64_C(0x00000000)
gs#6#u12_l#16 : unit = UNIT
goto endif_14
then_13:
ys#u12_l#9 : fbits(32, dec) = UINT64_C(0xFFFFFFFF)
gs#6#u12_l#16 : unit = UNIT
endif_14:
kill gs#2#u12_l#15 : bool
var gs#5#u12_l#10 : bool
var gs#3#u12_l#14 : fbits(32, dec)
gs#3#u12_l#14 : fbits(32, dec) = zz7nzJzK32#sail_zeros(32l)
gs#5#u12_l#10 : bool = zz7nzJzK32#eq_bits(ys#u12_l#9, gs#3#u12_l#14)
kill gs#3#u12_l#14 : fbits(32, dec)
var gs#7#u12_l#11 : bool
jump gs#5#u12_l#10 then_11
var gs#4#u12_l#12 : fbits(32, dec)
var gs#0#u9_l#13 : fbits(32, dec)
```



Thanks