Pydrofoil: Accelerating Sail-Based Instruction Set Simulators

Carl Friedrich Bolz-Tereick
Heinrich-Heine-Universität Düsseldorf, Germany

Luke Panayi
Imperial College London, UK

Ferdia McKeogh
University of St Andrews, UK

Tom Spink
University of St Andrews, UK

Martin Berger
University of Sussex, Brighton, UK
Montanarius Ltd, London, UK

Abstract

We present *Pydrofoil*, a multi-stage compiler that generates instruction set simulators (ISSs) from processor instruction set architectures (ISAs) expressed in the high-level, verification-oriented ISA specification language Sail. Pydrofoil achieves a $> 230\times$ speedup over the C-based ISS generated by Sail on our benchmarks, thanks to the following insights. (i) An ISS is effectively an interpreter loop, and tracing just-in-time (JIT) compilers have proven effective at accelerating those, albeit mostly for dynamically typed languages. (ii) ISS workloads are highly atypical, dominated by intensive bit manipulation operations. Conventional compiler optimisations for general-purpose programming languages have limited impact for speeding up such workloads. We develop suitable domain-specific optimisations. (iii) Neither tracing JIT compilers, nor ahead-of-time (AOT) compilation alone, even with domain-specific optimisations, suffice for the generation of performant ISSs. Pydrofoil therefore implements a hybrid approach, pairing an AOT compiler with a tracing JIT built on the meta-tracing PyPy framework. AOT and JIT use domain-specific optimisations. Our benchmarks demonstrate that combining AOT and JIT compilers provides significantly greater performance gains than using either compiler alone.

2012 ACM Subject Classification Software and its engineering → Retargetable compilers; Software and its engineering → Just-in-time compilers; Software and its engineering → Source code generation; Software and its engineering → Runtime environments; Software and its engineering → Domain specific languages; Hardware → Simulation and emulation; Software and its engineering → Interpreters

Keywords and phrases Instruction set architecture, processor, domain-specific language, just-in-time compilation, meta-tracing

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.3

Acknowledgements We thank Alasdair Armstrong, Max Bernstein, Tobias Grosser, Tim Hutt, Alex Jeffery, Yutaka Nagashima, Amir Naseredini, Zdeněk Přikryl, Alastair Reid, Niyas Sait, Laurie Tratt, Mojtaba Valizadeh and the anonymous reviewers for their helpful comments. We thank Nico Rittinghaus for implementing Tristate numbers in the RPython JIT.

1 Introduction

A processor's functionality is defined by its *instruction set architecture* (ISA), which specifies the machine language interpreted by the hardware. Accurate and comprehensive ISA specifications are essential for processor design, compiler development, and verification.

Instruction set simulators (ISSs), also called functional models, provide software-based representations of processor behaviour. The efficiency of ISS execution directly impacts processor development speed and associated costs. Throughout this paper, unless otherwise noted, ISS refers to full system simulation, encompassing the entire hardware and software stack, not user-mode simulation, which models only the application layer and sends system calls to the host's operating system layer [47, 50].

Traditionally, ISSs were manually implemented, a process that becomes increasingly time-consuming and error-prone as ISA complexity grows. To address this scalability challenge, *architecture definition languages* (ADLs) emerged, allowing for automated ISS generation from formal ISA descriptions [41]. These ADLs aim to streamline the processor development process. Performance overhead associated with generated ISSs is often mitigated through just-in-time (JIT) compilation or partial evaluation. While ISSs are crucial for functional verification, computer architects also require cycle-accurate models and hardware synthesis capabilities. Consequently, many widely used ADLs, such as SystemC [32], extend beyond pure ISA specification to incorporate microarchitectural details, including pipelining information. This integration, while providing comprehensive modelling capabilities, typically uses low-level language constructs, such as C++ classes and macros, which introduce complexity into the specification and verification process.

Sail [6] is an alternative to low-level ADLs and eschews all hardware implementation detail in favour of ISA-only modelling. Sail streamlines ISA specification with high-level language features such as arbitrary-precision arithmetic, pattern matching, and type-based bitwidth-generics. Sail supports formal verification by also compiling to logic, targeting theorem provers like Isabelle/HOL, Lean, and Z3. Notably, Sail has been adopted as the official ISA-specification language for RISC-V [33]. However, the Sail-generated ISS suffers from performance limitations. This leads to our research agenda: *how can a high-level ISA-specification language like Sail be compiled into performant ISSs?* The unique combination of Sail's high-level abstraction and focus on verification challenges the applicability of conventional ISS compilation techniques, including existing JIT-based approaches. Concretely, our paper seeks to answer the following research questions:

- **RQ1.** What are the bottlenecks causing the Sail-generated ISSs to be slow?
- **RQ2.** Can tracing JIT compilation be used to speed up Sail-generated ISSs significantly?
- **RQ3.** How close can we get to the performance of an industrial strength ISS, hand-tuned to RISC-V?

Contributions. To address the research questions, we present *Pydrofoil*, an open-source[1] tracing JIT compiler for Sail, implemented using the RPython/PyPy meta-tracing framework[2]. We developed and evaluated a suite of domain-specific optimisations tailored to the unique characteristics of ISS workloads, which differ from the typical Python applications targeted by PyPy. Our evaluation methodology employs established computer architecture benchmarks, primarily Linux-boot and SPEC workloads, alongside detailed ablation studies to quantify the impact of individual optimisations. In evaluations using the RISC-V Sail model, Pydrofoil exhibited a >230× speedup over the Sail-generated ISS, while exhibiting a 26.7× slowdown compared to handwritten and optimised QEMU. This achievement shows that whilst we make significant improvements over the Sail interpreter, there is still performance to be gained. We discuss methods of achieving this in Section 5.

[1] `https://github.com/pydrofoil/pydrofoil/`
[2] `https://pypy.org/`

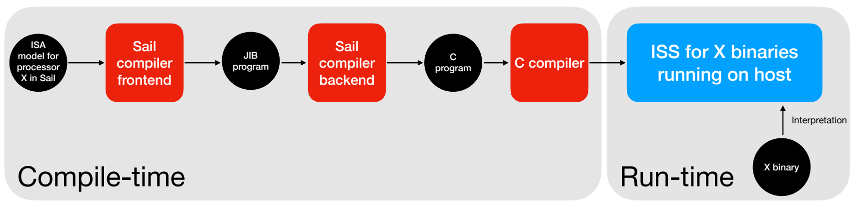


Figure 1 Sail's multi-stage compilation pipeline from Sail to an ISS running as a static binary.

2 Sail and the RISC-V Sail model

2.1 Sail

Sail is a domain-specific language (DSL) within the ML family [27, 40], designed to facilitate processor specification and verification. Sail hybridises first class support for stateful programming with a rich typing system, including algebraic-, singleton-, dependent- and higher-kinded types, as well as domain-specific arbitrary-precision and bitvector types, and a register construct. The compilation pipeline of Sail is depicted in Figure 1. Beyond its primary objective of enabling processor specification and verification through theorem proving, Sail's most noteworthy changes from other ML-family languages are:

- Sail restricts function definitions to first-order, eliminating higher-order functions.
- Sail supports *liquid types* [56, 80].

Liquid types, a form of dependent types, strike a balance between expressive power and decidable type inference. They are useful for addressing the bitwidth parameterisation inherent in modern processor architectures. For instance, the RISC-V ISA supports both 32-bit (RV32) and 64-bit (RV64) variants, where the majority of the ISA specification remains parametric with respect to the bitwidth, represented by a type declaration `xlen`. Traditional ADLs often lack strong typing system support for bitwidth-generic data structures, instead resorting to compile-time meta-programming techniques such as the C preprocessor or C++ templates. Meta-programming simplifies compiler optimisations, but necessitates redundant verification across different bitwidth configurations [12]. We will argue in Section 5 that the current Sail compiler may not fully exploit the optimisation potential afforded by liquid type usage within the RISC-V specification.

This paper primarily focuses on domain-specific compiler design rather than programming language ergonomics. Consequently, a detailed exposition of Sail's language features is beyond the scope of this work. However, it is noteworthy that Sail's capacity to seamlessly support both simulation and logic-based verification contributed to its adoption as the official specification language for RISC-V [33].

2.2 The RISC-V instruction set architecture

RISC-V [8, 75] is an open-source ISA designed to be modular. Unlike proprietary ISAs such as x86, Arm, and MIPS, RISC-V is vendor-neutral. This open nature facilitates academic research and industrial adoption. RISC-V "*is structured as a small base ISA with a variety of optional extensions. The base ISA is simple. [...] The optional extensions form a more powerful ISA for general purpose and high-performance computing.*" [75]. This modularity, combined with the availability of CHERI-RISC-V [78] and its 32-bit and 64-bit variants, results in a significant number of RISC-V ISA configurations, rendering manual ISS

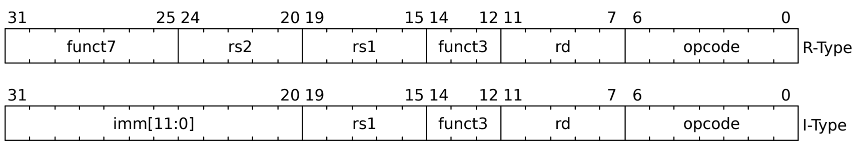


Figure 2 Two of the RISC-V instruction formats [76].

development impractical. Furthermore, the existence of RV32 and RV64 suggests the use of bitwidth-generic data structures, particularly bitvectors. While bitwidth-generics offer flexibility, they introduce compilation complexities and potential performance overhead in ISS implementations, as discussed in subsequent sections.

2.3 The RISC-V model in Sail

We shall now look at a concrete ISA specification in Sail, in order to illustrate how Sail supports the definition of ISA families, using the official RISC-V formal specification [34] as running example.

Abstractly, the toplevel function of the RISC-V Sail model is a big loop, called *fetch-decode-execute loop* in computer architecture. In each round, the loop fetches the next instruction from the external main memory (which an ISS simulates) as a bit pattern, then decodes that pattern into a concrete instruction, and finally calls an `execute` function that executes the decoded instruction. As a simplification, the following pseudo-code serves as a useful mental model:

```
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
```

Let us focus our attention on the addition instruction, which, in RISC-V, has the mnemonic `addi rd, rs1, imm`. This instruction adds the 12-bit integer `imm` to the content of a source register `rs1`. The result is stored in the destination register `rd`. Like most ISAs, RISC-V uses instruction formats, which can be seen as rough classifications of instructions that aid decoding bit patterns into instructions. Figure 2 shows the structure of two RISC-V instruction formats. Our example `addi rd, rs1, imm` is an instance of the `ITYPE` format. Like all `ITYPE` instructions,

- `ITYPE` has opcode `0b0010011`, found in bits 0-6.
- The destination register of the instruction is identified in bits 7-11, while the source register is given by bits 15-19.
- The actual function to be computed is in the `funct3` field (bits 12 - 14): `addi` uses `0b000` here, while, e.g., `0b100` identifies `xori`.
- The 12-bit constant to be added to the source register is in bits 20-31.

Let us look at a simplified form of the Sail implementation of `addi`.[3]

```
scattered union ast

val execute : ast -> Retired
scattered function execute

val encdec : ast <-> bits(32)
scattered mapping encdec

union clause ast = ILLEGAL : word
```

First some type declarations: `ast`, short for abstract syntax tree, is the disjoint union type, representing all decoded instructions. In Sail, `ast` is a *scattered* union, which means that the definitions of the variants are allowed to be in different files. This is useful when processor ISAs evolve: we can add new instruction formats in new files *without* needing to modify existing code. The function `execute` (also scattered) specifies the actual semantics of RISC-V instructions, meaning the changes to processor state induced by executing `addi`. The `execute` function takes an `ast` as argument and returns `Retired`, which is an `enum` specifying whether the instruction successfully executed or not (execution can fail, for example due to interrupts). The RISC-V model is imperative: `execute` can and does modify processor state, in particular registers, when invoked. Such processor state is global.

To translate between the encoding of instructions in bits, and their more structured representation as `ast`, the RISC-V model uses the (scattered) mapping `encdec`, short for "encoding-decoding". The keyword `mapping` is for *bidirectional* function definitions [25]. This means `encdec` can either take an `ast` and encode it into a 32-bit bitvector, or take a bitvector and decode it into an `ast`. This reflects, and the typing system enforces, the (almost) one-to-one relationship between ASTs and bitvectors – almost, because not all 32-bit patterns correspond to valid instructions. The model has a fall-through clause `union clause ast = ILLEGAL : word`.

We'll now look at the clauses of `ast`, `encdec` and `execute` for the `ITYPE` format.

```
1  enum iop = {RISCV_ADDI, RISCV_SLTI, RISCV_SLTIU, RISCV_XORI, RISCV_ORI, RISCV_ANDI}
2  ...
3  union clause ast = ITYPE : (bits(12), regidx, regidx, iop)
```

Line 1 enumerates all `ITYPE` instructions and names this type `iop`. Line 3, containing `union clause ast = ITYPE : ...`, defines the variant of the `ast` type that holds `ITYPE` instructions. This clause carries a tuple of information: `(bits(12), regidx, regidx, iop)` The first component holds 12 bits using the type `bits(12)`. This is the constant to be added to the source register. The next two components identify the source and target registers, where `regidx` abbreviates the type `bits(5)`, the type of bitvectors of length 5. (RISC-V has 32 general purpose registers, so 5 bits suffice.) We also need to translate between the 3-bit opcode field in the `ITYPE` instruction format and the enumeration `iop`. This is done by the following bidirectional `mapping`:

```
mapping encdec_iop : iop <-> bits(3) = {
  RISCV_ADDI  <-> 0b000,
  RISCV_SLTI  <-> 0b010,
  RISCV_SLTIU <-> 0b011,
  RISCV_ANDI  <-> 0b111,
  RISCV_ORI   <-> 0b110,
  RISCV_XORI  <-> 0b100
}

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

[3] See `https://github.com/riscv/sail-riscv/blob/2dfc4ff9f2bed3dcd0a3e8748211c99099e70ab7/model/riscv_insts_base.sail#L159` [55] for full code.

The actual semantics of the `ITYPE` instructions is given by the following `execute` clause:

```
 1  function clause execute (ITYPE (imm, rs1, rd, op)) = {
 2    let rs1_val = X(rs1);
 3    let immext : bits(64) = sign_extend(imm);
 4    let result : bits(64) = match op {
 5      RISCV_ADDI  => rs1_val + immext,
 6      RISCV_SLTI  => zero_extend(bool_to_bits(rs1_val <_s immext)),
 7      RISCV_SLTIU => zero_extend(bool_to_bits(rs1_val <_u immext)),
 8      RISCV_ANDI  => rs1_val & immext,
 9      RISCV_ORI   => rs1_val | immext,
10      RISCV_XORI  => rs1_val ^ immext
11    };
12    X(rd) = result;
13    RETIRE_SUCCESS
14  }
```

The code is beautifully simple and clear. However, the simplicity hides the following complexities:

- The function definition clause in Line 1 pattern matches on the `ast` data type, triggers when the instruction is `ITYPE`, and gives convenient, locally-scoped names to the components `imm`, `rs1`, `rd` and `op`, which were chosen to match the corresponding names in [76].
- Line 2 loads the register value to be added from `rs1`, the source register.
- Line 3 sign-extends the 12 bits value `imm` to the target register bitwidth (32 or 64 bits, depending on the version of RISC-V used). Two things are noteworthy about `sign_extend(imm)` and its use here. First, `sign_extend(imm)` is parameterised with source and target bitwidths, albeit subject to the requirement that the latter not be smaller than the former. This bitwidth parameterisation is visible in the function's type
  $$\texttt{forall 'n 'm, 'm} \geq \texttt{'n. (implicit('m), bits('n))} \rightarrow \texttt{bits('m)}$$
  which has type parameters `'n` and `'m` that range over integers (singleton types). The predicate `'m` $\geq$ `'n` is a refinement predicate, evaluated during type checking. We refer to the Sail manual [5] for a detailed explanation of Sail's type syntax. Second, while `sign_extend(imm)` takes two arguments, in Line 3 it is called with only one: the required target bitwidth `'m` is not specified in the call. The keyword `implicit` gives it away: Sail has a restricted form of implicit function parameterisation [37, 43], albeit resolved at compile-time. The resolution of implicit arguments is restricted to the expected output type, here `bits(64)`.
- Line 4 pattern matches on the symbolic name of the exact function to be executed.
- Line 5 triggers if we match `RISCV_ADDI`. In this case, the integer `imm` encoded in the instruction and suitably sign-extended, is added to the content in the source register by `rs1_val + immext`.
- Line 12 writes the result into the destination register `rd`.
- Line 13 returns an indication that the instruction has been successfully executed to the caller.
- Another subtlety is that the register access function $X(\dots)$ is overloaded, and reads from a register when on the right, but writes when on the left of an equality sign.

For this code, Sail offers strong *compile*-time guarantees, for example, we can neither assign bitvectors of different bitwidths, nor access a vector out-of-bounds.

2.4 How does the Sail compiler generate an ISS?

Sail generates a C-based ISS. Unfortunately, this ISS is slow: on a 2024 laptop (AMD Ryzen 7 PRO 7840U, with 32 GB main memory) it executes less than a million RISC-V instructions per second; QEMU simulates a few hundred million instructions in the same time. In this section we identify the root-causes of Sail's poor performance. We start with two possibly unsurprising remarks to set the scene.

```
 1 uint64_t imm = ...;
 2 ...
 3 // let rs1_val = X(rs1);
 4 uint64_t rs1_val;
 5 rs1_val = rX_bits(rs1);
 6 // let immext : xlenbits = sign_extend(imm);
 7 uint64_t immext;
 8 {
 9   i_generic i_generic_64;
10   CREATE(i_generic)(&i_generic_64);
11   CONVERT_OF(i_generic, int64_t)(&i_generic_64, INT64_C(64));
12   bv_generic imm_bv_generic;
13   CREATE(bv_generic)(&imm_bv_generic);
14   CONVERT_OF(bv_generic, fbits)(&imm_bv_generic, imm, UINT64_C(12) , true);
15   bv_generic immext_bv_generic;
16   CREATE(bv_generic)(&immext_bv_generic);
17   sign_extend(&immext_bv_generic, i_generic_64, imm_bv_generic);
18   immext = CONVERT_OF(fbits, bv_generic)(immext_bv_generic, true);
19   KILL(bv_generic)(&immext_bv_generic);
20   KILL(bv_generic)(&imm_bv_generic);
21   KILL(i_generic)(&i_generic_64);
22 }
23 uint64_t result;
24 {
25   __label__ case_8084, finish_match_8083;
26   {
27     if ((RISCV_ADDI != op)) goto case_8084;
28     // rs1_val + immext
29     result = rs1_val + immext;
30     goto finish_match_8083;
31   }
32 case_8084: ;
33   ... // the other cases
34 }
35 {
36   wX_bits(rd, result);
37 }
38 cbz31056 = RETIRE_SUCCESS;
39   goto finish_match_8062;
```

Figure 3 Fragment of the `execute` clause of `ITYPE` instructions after compilation to C by Sail.

- Sail is a high-level language that emphasises precise expression of ISA semantics over concern for simulation speed at every step. High-level languages are harder to generate fast code from than low-level languages.
- Development of optimising compilers is labour intensive and there is a direct correlation between the level of optimisation achievable and developer hours spent. The Sail community chose to spend the scarce developer resources available in a university lab on verification-related parts of the toolchain.

Let us look at concrete code that shows where Sail leaves performance on the table. Figure 3 shows C generated from the `execute` clause of `ITYPE` instructions, discussed above. The code is lightly edited for readability, with Sail code added as comments before the corresponding C code lines. The computation in Line 29 of the result in the `ADDI` case is mapped to an addition on the host machine. Sign extending `imm`, the 12-bit immediate, is done by calling `sign_extend` in Line 17. The arguments of this call are of type `bv_generic`, which is a *heap*-allocated data structure storing bitvectors of arbitrary width. Converting `imm` to `bv_generic`, requires a heap allocation as part of the `CONVERT_OF` macro; `i_generic` is also heap allocated and represents integers of arbitrary size.[4] In Sail, both `bv_generic`

[4] Those types are called `lbits` and `sail_int` in the *actual* C code that Sail generates. We change their names for consistency with Section 3, where we discuss the datatypes in more detail.

and `i_generic` are implemented using the *GNU Multiple Precision Arithmetic library (GMP)* [69] for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers. We summarise the key reasons why the Sail-generated simulator is slow:

- **Bitvectors.** Sail tries to infer the width of the bitvector variables in the program. If they are statically known to fit into an unsigned 64-bit integer, the C backend will map them to the `uint64_t` type. However, many functions in a Sail spec are parametrised with bitwidths, for example `sign_extend`. The bitvectors in these functions are implemented using `bv_generic` and thus ultimately become GMP integers. This has a number of drawbacks: all operations on GMP-based bitvectors require costly heap allocations, because GMP integer types are always heap-allocated.
- **Integers.** Sail's integer type has arbitrary precision. As with bitvectors, Sail tries to infer whether a given integer variables can only have values that fit into a signed 64-bit integer. When that is possible, the C backend will use `int64_t`. However, where it is impossible for Sail to prove that the values fits into `int64_t`, the C backend uses an arbitrary-precision representation from GMP which requires heap allocations on every integer operation.
- **Interpretation overhead.** Finally, Sail generates an interpreter, which has to re-analyse the bits of the program being simulated again and again. Note that actual processors use caches to ameliorate this overhead, e.g., with an instruction cache.

This brings us to our key objective: **how can we improve simulation performance?**

2.5 RPython

Pydrofoil is implemented using the PyPy framework [53]. RPython [3] is PyPy's implementation language. RPython (short for Restricted Python) is a statically typable subset of Python2. The language was developed to aid the implementation of high-performance virtual machines for dynamic languages. To this end, RPython contains a generic meta-tracing just-in-time compilation infrastructure [15] that can be retargeted to different guest languages. Meta-tracing works by tracing through the main interpreter loop (written in the host language – here RPython) of the guest language interpreter [9,23,24,42,64]. The tracing JIT traces many iterations of the main interpreter loop, effectively unrolling it. This means that one trace contains several guest language instructions, which can then be optimised together. The trace is optimised, compiled to machine code, and then available to execute the corresponding guest language program fragment more efficiently. As ISA models, whether written in Sail or not, are essentially interpreters for an ISA, it seems reasonable to expect that the RPython meta-tracing JIT can speed up the Sail interpreter loop, and compile guest machine instructions to host machine instructions. This has been confirmed in the Pydgin project [39], which implements user-mode ISSs for subsets of Arm v6, MIPS and RISC-V by writing interpreters for them manually in RPython.

3 How is Pydrofoil implemented?

Pydrofoil's approach for better ISS performance is based on the following ideas:

- Better static optimisations of Sail code, in particular optimisations for bitvectors and integers, with the goal of minimising heap allocations.
- Leveraging run-time information with the help of the RPython tracing JIT compiler for further optimisations not available statically.

The first one is, at least in principle, also available to Sail's C backend.

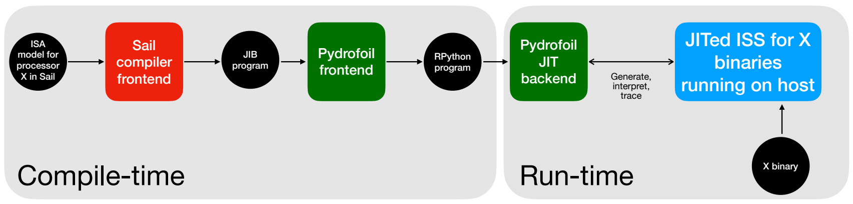


Figure 4 Pydrofoil multi-stage compilation pipeline from Sail to a JITed ISS.

3.1 Pydrofoil architecture

The Sail system emits an ISS in C to be able to use the existing ecosystem of optimising C compilers. In the same way, Pydrofoil reuses the Sail frontend, RPython and the RPython JIT to generate a faster ISS from a Sail model. Pydrofoil's overall architecture is shown in Figure 4.

Pydrofoil's input is JIB, the Sail compiler's textual intermediate representation [7]. When JIB is emitted, all of Sail's type checking has been done, all variables have explicit types and all pattern matching is compiled away into basic blocks and (conditional) gotos. This makes JIB easy for the Pydrofoil ahead-of-time (AOT) compiler to work with. From JIB, Pydrofoil generates its own SSA-based intermediate representation (IR) for Sail functions. These functions are optimised by Pydrofoil's static optimisation passes. Then Pydrofoil generates RPython source code from the optimised IR. The generated code makes use of Pydrofoil's runtime library of support functions that implement Sail datatypes, including problem cases like the aforementioned bitvectors, and arbitrary precision integers. The support library also implements the simulated main memory that the simulated processor interacts with. This resulting RPython code is saved to file, and can be executed with a normal Python2 implementation. This is not performant, but useful for testing. Pydrofoil further translates it into a binary with the existing RPython translation toolchain. During this translation, a tracing JIT compiler can be (optionally) inserted into the resulting binary. The following sections will elaborate on key steps of this process.

3.2 Static Pydrofoil IR optimisations

Once JIB is translated to Pydrofoil's SSA-based IR, we apply a number of standard compiler optimisations: constant folding, dead code elimination, common subexpression elimination, inlining, and scalar replacement of aggregates (using a simple form of partial escape analysis) [14, 62]. All static optimisations are applied until a fixpoint is reached, and further optimisation passes would not alter the IR.

3.2.1 Static Optimisations of bitvector and integer operations

We focus our attention on static optimisations for operations on the bitvector and integer data types, because of their importance for ISSs in general, and Sail in particular. JIB uses two different types for each of these:[5]

[5] In line with the previous section, we renamed two JIB types for consistency: `%bv_generic` is called `%bv` in JIB, whereas `%i_generic` is just called `%i`.

```
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 ...
```

Figure 5 JIB code for the `execute` clause of `ITYPE` instructions.

- **Bitvectors:** if JIB can infer precise bitwidth, then bitvector types like `%bv64` or `%bv16` of fixed width are used. Otherwise `%bv_generic` is used, a generic bitvector type where the width is not known at runtime (which turns into the `bv_generic`).
- **Integers:** if JIB can infer that an integer can always fit into a 64-bit signed machine integer then `%i64` is used. Otherwise `%i_generic`, a type of arbitrary precision integer, is used.

JIB code frequently casts between the generic and fixed-width variants of the respective representations. Casts do not change the underlying value, but cost a heap allocation when casting to a generic datatype, and heap memory reads when casting in the other direction. Both are expensive. Another reason why the generic forms `%bv_generic` and `%i_generic` are much less efficient at runtime is that the operations (e.g., addition for `%i_generic`) cannot be mapped to directly to host CPU operations, since we do not know their bitwidth. Here is a concrete example. The JIB code in Figure 5, lightly edited for clarity, corresponds to the beginning of `ITYPE`'s `execute` fragment that we already saw in Sail and C in Section 2.3, and Figure 3. Line 8 in Figure 5 shows the `sign_extend` from Sail in comments. JIB Lines 9–16, execute this `sign_extend`: first the immediate value of type `%bv12` is cast to a generic bitvector (Line 13), then sign-extended to 64 bits (Line 15), and finally cast back to `%bv64` (Line 16). This sign extension alone by default requires three heap allocations. The reason why these casts are necessary is that the `sign_extend` function takes a `bv_generic` and an `i_generic` and returns a `bv_generic`.[6]

For all built-in Sail functions and operations that deal with bitvectors and integers, Pydrofoil implements a bundle of variants. They all perform the same computation, but take more specific types as arguments. This enables Pydrofoil to avoid some heap allocations, as we discuss next. To optimise such operations on generic bitvectors and integers, Pydrofoil's static optimiser uses peephole rewrite rules that inspect the IR of a function and rewrite the operations on these types into more efficient forms. In the best case rewrites remove casts between generic and specific bitvector types (as well as machine integers and arbitrary integers) and use specialised operations on the types (when they are available in Pydrofoil's runtime library).

[6] Note that the JIB translation of `sign_extend` takes a bitvector and then an integer as input. We saw in Section 2.3 that the order is reversed in the eponymous Sail function. This is an artefact of Sail's resolution of implicit arguments, and irrelevant for compiler optimisations. It's implemented in the Sail prelude of the RISC-V model by having a small one-operation Sail helper function that has the implicit argument, which calls the JIB `sign_extend` operation with arguments reversed.

Here are Pydrofoil's variants of the `sign_extend` function, together with the rewrite rules that the static optimiser uses:

```
# 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)
```

After applying the rewrite rules to the `sign_extend` call in the example code on Line 15 in Figure 5, it can be replaced by a call to `sign_extend_bv_c⟨12, 64⟩`. The latter takes a `%bv12` as argument, and returns a `%bv64`. This means we get rid of all three heap allocations. The resulting code is equivalent to the following JIB code and does not contain casts:

```
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
  immext = sign_extend_bv_c<12, 64>(imm)
...
```

To be able to narrow the generic integer type `%i_generic` to `%i64`, Pydrofoil also performs a range analysis. If the analysis finds an integer variable where the range fits into a signed 64-bit int, Pydrofoil narrows the type to `%i64`. An example is the RISC-V `mulw` instruction, which multiplies two signed 32-bit bitvectors by converting them to `%i_generic` first. The result of the multiplication of these two integers must fit into an `%i64`, so Pydrofoil narrows the integer types and rewrites the multiplication operation accordingly.

3.2.2 Interaction with inlining

Inlining helps to make bitvector and integer operation rewrites more effective. Many Sail functions are small helper functions that operate on generic bitvectors and integers, passed as arguments. After inlining such small functions into their use contexts, the concrete bitvector widths of the argument can often be inferred from the calling context. Then the bitvector operations in the inlined function are specialised to smaller bitwidths.

Consider another `ITYPE` operation as an example: the `slti` case uses the `operator <_s`, to perform the signed comparison. It is defined as follows in Sail:

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

Here `signed` is a Sail function that takes a bitvector of width n, and returns the corresponding signed integer. This code is compiled to the JIB code in the top part of Figure 6. It implements a signed comparison of bitvectors and operates on generic bitvectors. In the concrete context where the function is called, the bitvector widths are mostly known. For example, its use in the `slti` instruction can be seen in the bottom half of Figure 6.

```
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   ...
```

Figure 6 JIB code for `operator <_s` (top half) and its use in the JIB code of the `execute` clause of the `slti` instruction (bottom half).

The arguments to `operator <_s` are cast from `%bv64` to `%bv_generic` (Lines 6 and 8). After inlining the call to `operator <_s`, the calls to `signed` in its body can therefore be optimised to a variant that takes a `%bv64` and returns an `%i64`, which then allows the `lt_int` operation to be replaced by a version that operates on `%i64`. This removes *all* heap allocations from the inlined body of `operator <_s` that replaces the call-site.

3.2.3 Function specialisation

In cases where a function is too large[7] to be inlined, Pydrofoil tries to specialise it to the specific bitvector widths and `%i64` types of its arguments at the function's call-sites. The specialiser will go over all the non-inlinable function calls in the program and check whether any of the arguments are of type `%bv_generic` or `%i_generic` and either constant, or the result of casts from more specific types. For such calls, the target function will be copied into a more specific version. The copied function takes the more specific argument types. The body of the copy is then optimised to take the newly available bitvector widths of its arguments into account. Sometimes, as the result of specialisation, the return type of a function can become more specific, enabling the caller to be optimised further.

3.2.4 When do the static bitvector and integer optimisations fail?

Peephole rewrites for bitvector and integer operations work only within single functions. They can only be applied when the arguments of such operations are cast from more specific types in the same function where the operation happens. Inlining and function specialisation help with peephole rewrites being restricted to single functions. But this is sometimes still insufficient for giving peephole rewrites enough context to do their work. In more complicated cases, particularly if the arguments of bitvector operations are read out of a `union` or a

[7] We use an inlining limit of 25 JIB operations in up to 4 basic blocks at the moment.

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

enum word_width = {BYTE, HALF, WORD, DOUBLE}

union clause ast = LOAD : (bits(12), regidx, regidx, bool, word_width, bool, bool)

val extend_value : forall 'n, 0 < 'n <= xlen. (bool, bits('n)) -> xlenbits
function extend_value(is_unsigned, value) = if is_unsigned then zero_extend(value)
     else sign_extend(value)

mapping size_bytes : word_width <-> {1, 2, 4, 8} = {
  BYTE   <-> 1,
  HALF   <-> 2,
  WORD   <-> 4,
  DOUBLE <-> 8,
}

function clause execute (LOAD(imm, rs1, rd, is_unsigned, width, aq, rl)) = {
  let offset : xlenbits = sign_extend(imm);
  let width_bytes = size_bytes(width);
  let addr = X(rs1) + offset;
  // simplified, the real code deals with virtual memory and checks for alignment
  match mem_read(Read(Data), addr, width_bytes, aq, rl, false) {
    Ok(result) => { X(rd) = extend_value(is_unsigned, result); RETIRE_SUCCESS },
    Err(e)     => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
  }
}
```

Figure 7 Sail code for the RISC-V instructions that load from main memory into a register.

`struct`, more powerful static analysis would be needed to be able to optimise the operations to more specific variants. An example is the `LOAD` constructor in the RISC-V Sail model, which implements loading data from simulated main memory into a register. The Sail code can be seen in Figure 7.

The result of calling `mem_read` is a `union` where one of the variants stores a `%bv_generic` at the JIB level. The width of the result of the read from simulated main memory is data-dependent on some bits of the encoded instruction. Therefore it cannot be specialised at compile-time to the `width_bytes` parameter. This also means that the bitwidth of the result is unknown to the static optimisations.

Similarly, it is much harder to track concrete bitvector widths when the values pass through a `union` or are stored into a `struct`. It is conceivable to handle such cases with more powerful intra-procedural static analysis, and taking the information the liquid types into account. We leave this as future work, see Section 5.

3.3 RPython code generation

After Sail functions have been optimised, we generate RPython source code from each. This is straightforward, the only complexity being that the IR control flow graphs use basic blocks with (conditional) jumps between them, and RPython only supports structured control flow. Pydrofoil compiles that to the standard construction of a program counter variable and an infinite loop with one condition for every basic block.[8]

[8] This is an old approach [18], see Harel [30] for a thorough historical discussion. There are better approaches to generating block-structured code out of arbitrary control flow graphs [49, 81] and we plan to switch to one of them.

3.4 Adding a JIT compiler with RPython

To gain efficiency beyond what an interpreter-based simulator can offer, we use RPython's meta-tracing JIT compiler. It traces the execution of the ISS through one loop of the simulated guest program and turns the resulting traces into host machine code. In this way, RPython's tracing JIT acts as a trace-based dynamic binary translator for the simulated instruction set architecture.

RPython's meta-tracing JIT needs a few annotations in the outermost execution loop of the Sail model in question. In particular, the tracing JIT needs to know what the core execution loop is, and which register of the simulated machine stores the program counter. The meta-tracing JIT uses the program counter to detect backwards jumps in the guest program in order to identify loops at the guest program level.

The meta-tracing JIT can further optimise the generic bitvector and integer operations that the static optimisations of Section 3.2 did not improve. Since it traces a concrete execution path of guest machine code, all used bitvector widths are observable by the JIT. The JIT can thus specialise the generic bitvector operations at runtime to the observed widths. This yields performance improvements because the JIT can remove some of the remaining heap allocations of bitvectors and integers using escape analysis [14]. However, it is still preferable to statically remove as many generic bitvector operations as possible, as otherwise the JIT has to continually remove them, every time the ISS is executed.

Consider the following JIT trace, generated for a single RISC-V `addi` instruction (the details of this trace are not important):

```
1   ...
2   i106 = getfield_gc_i(p1, field=s4)
3   ...
4   # c.addi s4. 0x8
5   i144 = getarrayitem_gc_i(ConstPtr(
        ptr142), 26)
6   i146 = int_and(i144, 65535)
7   i148 = int_and(i144, 3)
8   i150 = int_eq(i148, 3)
9   guard_false(i150)
10  i152 = int_eq(i146, 1)
11  guard_false(i152)
12  i154 = uint_rshift(i146, 5)
13  i156 = int_and(i154, 1)
14  i158 = uint_rshift(i146, 6)
15  i160 = int_and(i158, 1)
16  i162 = int_lshift(i156, 1)
17  i163 = int_or(i162, i160)
18  i165 = uint_rshift(i146, 7)
19  i167 = int_and(i165, 15)
20  i169 = uint_rshift(i146, 11)
21  i171 = int_and(i169, 3)
22  i173 = int_lshift(i171, 2)
23  i174 = int_or(i173, i163)
24  i176 = int_lshift(i167, 4)
25  i177 = int_or(i176, i174)
26  i178 = int_is_zero(i177)
27  guard_false(i178)
28  i180 = uint_rshift(i146, 13)
29  i181 = int_is_zero(i180)
30  guard_true(i181)
31  i182 = int_is_zero(i148)
32  guard_false(i182)
33  i184 = int_and(i165, 31)
34  i186 = uint_rshift(i146, 12)
35  i188 = uint_rshift(i146, 2)
36  i190 = int_and(i188, 31)
37  i192 = int_lshift(i186, 5)
38  i193 = int_or(i192, i190)
39  i194 = int_is_zero(i193)
40  guard_false(i194)
41  i195 = int_is_zero(i184)
42  guard_false(i195)
43  i197 = int_eq(i148, 1)
44  guard_true(i197)
45  i199 = int_xor(i193, 32)
46  i201 = int_sub(i199, 32)
47  i203 = int_and(i201, 4095)
48  guard_value(i184, 20)
49  i206 = int_xor(i203, 2048)
50  i208 = int_sub(i206, 2048)
51  i209 = int_add(i106, i208)
52  i211 = int_add(i2, 2)
53  i213 = int_add(i0, 2)
54  setfield_gc(p1, i209, field=s4)
55  i215 = int_eq(i213, 10000)
56  guard_false(i215)
57  # next guest instruction
```

The trace does not contain heap allocations, which is good. However, the trace is long, given that it performs only a single guest addition. The problem is the `getarrayitem_gc_i` instruction at the start, which is the instruction fetch from simulated main memory. A large part of the trace (Lines 6-50) is manipulating the result of that memory read to perform the instruction decoding. This happens despite the program counter register being constant for that point of the trace. In the following section we will look at an optimisation for removing those *instruction-decoding* trace instructions.

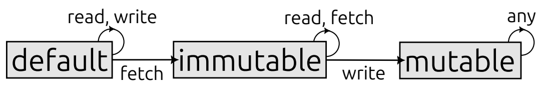


Figure 8 Diagram of the states a 64-bit word of simulated main memory can be in.

3.5 Making the main memory simulation JIT-friendly

One big problem for the JIT compiler is that while it knows the concrete value of the guest program counter, that is not enough for optimising away the decoding of the instruction stored at that program counter.

The reason is the possibility of *self-modifying code*. In practice true self-modifying code is rare, and so we want the JIT to exploit this fact. To do this, we track the status of every part of the simulated main memory. Currently, this tracking happens at the granularity of 8 bytes.[9] Every 64-bit word in simulated main memory can be in one of three states: `status_default`, `status_immutable`, `status_mutable`. All the 64-bit words start out in `status_default` state. If a `status_default` 64-bit word is read or written, its status is unchanged, and the memory read proceeds normally. However, if a `status_default` is read from during instruction fetch, it transitions to `status_immutable`. The JIT will constant-fold reads from `status_immutable` 64-bit words and return the concrete value of the bits stored in the 64-bit word. To behave correctly in the face of self-modifying code, the simulated main memory checks on every write whether a `status_immutable` 64-bit word is being written to.[10] When a `status_immutable` 64-bit word is changed, the generated host machine code is potentially no longer valid, because it was generated under the assumption that `status_immutable` 64-bit words are immutable. Therefore all relevant host machine code is invalidated in such a situation, and the modified 64-bit word is marked as `status_mutable`. `status_mutable` 64-bit words are assumed to potentially contain mutable code, so the JIT will never constant-fold reads from them during host machine code generation. Figure 8 shows a state diagram of the various states every 64-bit word of the simulated main memory can be in, and the transitions it can go through.

After adding this status tracking logic to the simulated main memory, the trace for the `addi` instruction in Section 3.4 gets much simpler. Since the memory read of the instruction bits happens from a `status_immutable` 64-bit word, the bits of the guest instruction are constant. Therefore the trace operations that decode those bits all get constant-folded away. This kind of trace length reduction due to memory status tracking is typical: on average the trace for one guest instruction gets 3–4 times shorter with memory status tracking. The remaining trace looks like this:

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

[9] It is possible that other granularities work better. Pydrofoil could e.g., do this kind of tracking on the granularity of guest pages and take the permission bits from the page tables into account.
[10] This extra check makes all writes to simulated main memory slower, but the trade-off is worth it because instruction fetch is common: there are more instruction fetches than memory writes.

Now the trace looks good! The `addi s4, 0x8` instruction is compiled to a `int_add(..., 8)` instruction, which is its closest equivalent in the RPython trace IR. There are a few extra instructions around to read and write the value of the register field in the `p1` `struct` which contains the simulated machine's state. In addition, there are three instructions that check whether 10,000 instructions have been executed, which would make it time to tick the simulated clock (we discuss the issue of clocks in more details in Section 5).

3.6 Improving the RPython JIT integer optimisations

While developing Pydrofoil, we identified a number of missing trace optimisations for bit manipulation operations in the RPython JIT. This may not be surprising: the RPython JIT was developed for dynamic language implementations where such operations are rare. In contrast, bit manipulation is pervasive in processors, and, as a consequence, in the Pydrofoil traces the JIT needs to optimise. In order to adapt the JIT's optimising passes to such workloads, we have added a number of new integer optimisations. Let us give an example. In processors, individual bits are often used to hold permissions, e.g., whether a specific address range allows non-aligned access. Typically such a permission bit rarely changes, if ever. Nevertheless the permission bit is checked on every access. In hardware such a check is cheap. In software simulation of hardware, checking the bit again and again is expensive. In principle, a JIT can determine during trace-optimisation if a permission bit changes from one instruction to the next. If not, all but the first check can be elided. However JITs targeting high-level languages typically do not track value-constancy *at the bit-level*.

We adapt *Tristate numbers*, used in the Linux kernel by the eBPF verifier to reason about which bits in a variable have constant values [71], to the RPython JIT. An equivalent approach is also used in other compilers such as LLVM, where it is called *Known Bits* [68]. The Tristate number abstraction lets the JIT track values of constant bits of interest in integer and bitvector variables (which map to the same JIT trace IR type) when optimising traces. This enables the removal of permission checks where the JIT can prove that the relevant bits cannot change.

A concrete example for why this is useful are alignment checks. If two load instructions are executed that use the same base register, e.g. like this:

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

Then both loads have to check whether the pointer the value comes from is word-aligned, which means that the lowest bits are zero. The Tristate number analysis lets the JIT conclude that if the first load is aligned, the second one is aligned too, and so the alignment check can be removed from the second load. Here is a sketch[11] of what this looks like in a JIT trace:

```
# ld t0, 0x8(a0) # first load
i1 = getfield_gc_i(p1, field=a0)
i2 = int_add(i1, 8) # add offset to get the actual pointer
i3 = int_and(i2, 7) # alignment check: extract last three bits
i4 = int_is_zero(i3) # check that they are zero
guard_true(i4) # now the JIT knows that i1 and i2 are of the form ???...???000
... # do the actual load from memory

# ld t1, 0x10(a0) # second load
i12 = int_add(i1, 16) # must be of the form ???...???000, because both i1 and 16 are
i13 = int_and(i12, 7) # alignment check for second load: i13 is 0
i14 = int_is_zero(i13) # always 1
guard_true(i14) # can be removed by the JIT's opimizer because i14 is constant 1
... # do the actual load from memory
```

[11] The actual trace is significantly more complicated, due to bounds checks and address translation.

4 Benchmarks

To evaluate the performance of Pydrofoil and quantify the impact of our domain-specific optimisations and JIT compilation, we conducted a comprehensive benchmarking study. Our evaluation, constrained by available resources, aims to be a representative set of industrial ISS workloads, and has four main parts.

- Comparison of Pydrofoil with the Sail-generated ISS, using RISC-V 64.
- Ablations to understand the contribution of each individual domain-specific Pydrofoil optimisation, using RISC-V 64.
- Comparison of Pydrofoil with QEMU and Spike, two handwritten industry-standard ISSs.
- Preliminary benchmarks using Arm v9.4a and CHERIoT ISAs.

We use two classes of benchmark: SPEC CPU and booting Linux (excluding CHERIoT, described below). Both are widely used in computer architecture, and motivated next.

- SPEC, the Standard Performance Evaluation Corporation benchmark [63], is a suite of standardised tests evaluating the performance of computer systems. We focus on the SPEC CPU benchmarks [17], which measure the performance of a system's processor and memory subsystem. We restrict our attention to SPECint, and avoid the floating point arithmetic in SPEC CPU because currently the RISC-V model lacks a Sail implementation of the floating point extensions. Instead, the RISC-V model relies on the C-based Berkeley SoftFloat library [31], and benchmarking floating points would *not* exercise Pydrofoil[12].
- Booting Linux is a critical benchmark due to its prevalence in ISS applications and its challenge for JIT compilers, as the boot process lacks the long-running loops that JITs excel at optimising. Typically, the boot process executes a lot of code that is never seen again, and so this benchmark helps to establish JIT compilation latency.

Each SPEC benchmark was integrated into a complete Linux 6.7.0 boot image, using the initial ramdisk [79] (`initrd`). This approach is necessary because the Sail models do not contain emulated disk devices that could have been used used for storing programs. Unfortunately, due to the limitation of a few hundred MB on `initrd` sizes, we were not able to run `657.xz_s.1`. For the `625.x264` variants we had the same problem but moved input file generation into the Linux image. We mark those SPEC benchmarks with an asterisk (*) when we discuss them in Section 4.3.

Due to the deterministic nature of Sail models, and their lack of access to the host clock, all timing was performed externally on the host system, measuring the simulator's total wall-clock runtime. Therefore, our SPEC benchmark times also include the Linux 6.7.0 boot process until user-mode (about 138 million instructions).

Hardware and software used for benchmarking. All Sail compilation uses the following parameters affecting optimisation: `–dno_cast –O –Oconstant_fold –memo_z3`. Generated C was compiled with `gcc`. For pragmatic reasons, we use three different compute setups (indicating for each benchmark which is used).

- TaiShan server with Kunpeng 920 Arm v8.4 cores, 500 GB main memory, running Ubuntu 18.04.6.
- AMD Ryzen 7 PRO 7840U with 32 GB main memory, running Ubuntu 24.04.6.
- AMD Ryzen 9 7950x3d, with 128GB main memory, running Ubuntu 24.04.1.

[12] A pure Sail implementation of the floating point extensions is in preparation. When it becomes part of the official RISC-V model, it makes sense to revisit our benchmarks.

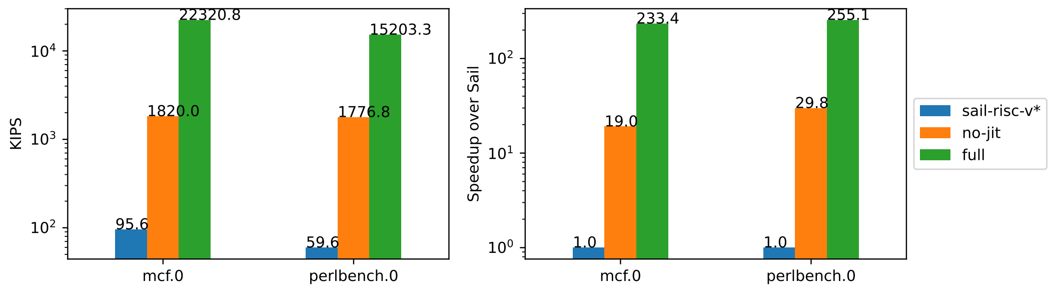


Figure 9 Instructions/second comparison between Sail and Pydrofoil. The left plot shows KIPS (= kilo instructions per second), the right the speedup over Sail. Higher is better.

4.1 Comparing Pydrofoil and Sail performance for the RISC-V model

Pydrofoil's most important goal was generating faster ISSs than Sail itself. This section compares Pydrofoil with Sail. We are resource constrained: the Sail-generated RISC-V ISS is too slow! For example `605.mcf_s.0`, which executes 1.2 trillion instructions, takes roughly six months on the TaiShan server. Therefore, we decided to run the comparison on only two arbitrarily chosen SPEC benchmarks, `605.mcf.0` and `600.perlbench.0`. We stopped the benchmarks for the Sail-generated ISS after 20 billion instructions. Pydrofoil is run to conclusion on the same benchmarks. We compare the instructions per second executed by both systems, extrapolated from the first 20 billion for Sail.

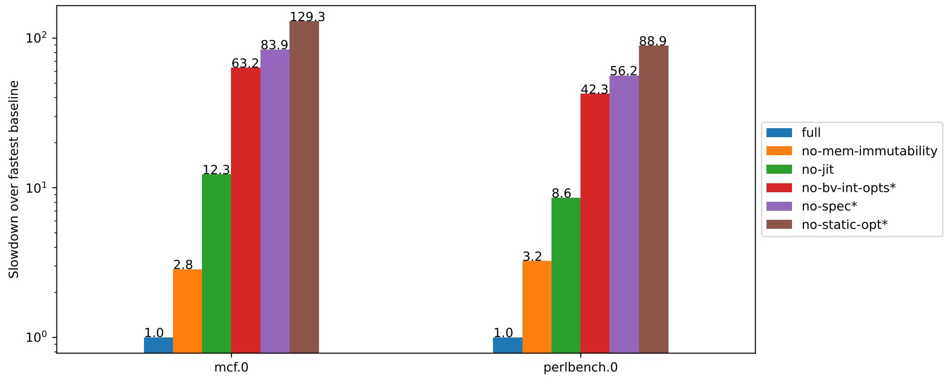
Results. This benchmark uses the TaiShan server. The results of this can be seen in Figure 9. We see that Pydrofoil is 233× and 255× faster than the Sail RISC-V ISS on both benchmarks. Even without the JIT, Pydrofoil is at least 19× faster than Sail.

4.2 Ablations: benchmarking Pydrofoil variants

To quantify the effect of our optimisations, we conduct ablation studies: we systematically disable successively more optimisations and measure the resulting slowdowns. This gives us a quantitative approximation of the relative impact of each optimisation, providing insights into their individual and collective value. We use the following variants.

- `full`: The full Pydrofoil simulator, running the JIT and the AOT with all optimisation enabled.
- `no−mem−immutability`: Disables memory immutability tracking (Section 3.5).
- `no−jit`: Additionally disables the JIT (Section 3.4) and works only as an interpreter at runtime.
- `no−bv−int−opts`: Additionally disables the static optimisations that replace generic bitvector and generic integer operations with more specialise variants operating on 64-bit integers (Section 3.2.1).
- `no−spec`: Additionally disables inlining (Section 3.2.2) and function specialisation (Section 3.2.3).
- `no−static−opt`: Additionally disables all remaining static optimisations (Section 3.2).

Unfortunately, we were unable to run all ablations against the SPEC CPU benchmarks as planned. The problem is that the ablations with many of the optimisations disabled run much too slow to finish in a realistic time frame. For example the aforementioned



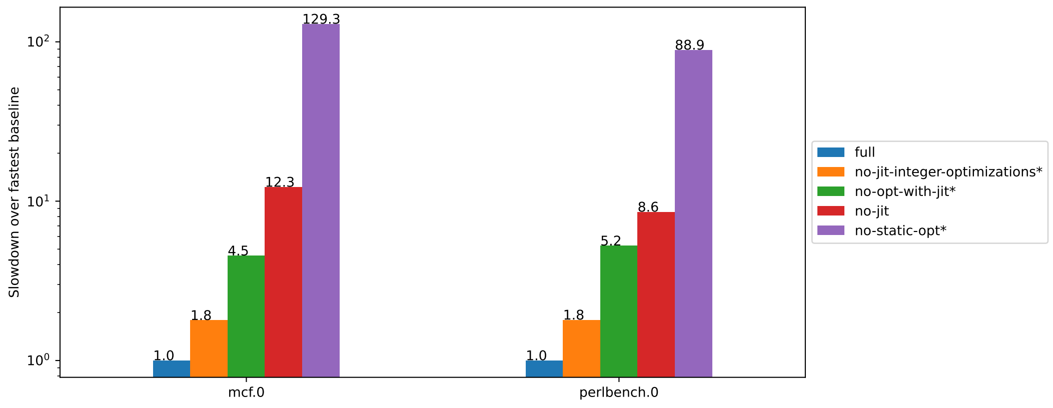
	KIPS		slowdown	
Benchmark	mcf.0	perlbench.0	mcf.0	perlbench.0
Implementation				
full	22320.79	15203.28	1.00	1.00
no-mem-immutability	7836.32	4701.09	2.85	3.23
no-jit	1820.04	1776.76	12.26	8.56
no-bv-int-opts*	353.13	359.32	63.21	42.31
no-spec*	266.06	270.47	83.89	56.21
no-static-opt*	172.60	170.92	129.32	88.95

Figure 10 Benchmark results for ablations, as graph and table. The y-axis of the plot is logarithmic. Benchmark results with * are extrapolated from the first 20 billion instructions.

`605.mcf_s.0` takes several months with `no-static-opt`. We deal with this as we did in the previous comparison and run the ablations on the same arbitrarily chosen SPEC benchmarks, `605.mcf.0` and `600.perlbench.0`. We stopped the benchmarks for the slowest ablations after 20 billion instructions. The remaining ablations are run to conclusion on the same benchmarks. We compare the instructions per second executed by both systems, extrapolated from the first 20 billion for the slowest ablations.

Results. This benchmark uses the TaiShan server. The results are shown in Figure 10. We see clearly that the JIT is fundamental for performance, it provides (together with memory immutability tracking) a 12×/8× performance boost. The static bitvector and integer optimisations are the most useful static optimisations. Disabling them slows Pydrofoil down by about a further 5×. Inlining, function specialisation and all other static optimisations are less important. Turning the runtime-switchable integer representation off leads to a further slowdown of more than 2×. We also investigated what is more important: static optimisations or JIT compilation. To that end, we add another ablation, `no-opt-with-jit` It is outside of the sequence of ablations in the previous subsection, because it disables all static optimisations but leaves the JIT-compiler active. The results are in Figure 11. We see that this variant is 4.54×/5.25× slower than `full`. This slowdown is less than `no-jit`, which removes the JIT but keeps the static optimisations. We conclude that the JIT can partly compensate for missing static optimisations (but not vice versa). The JIT does this by performing some, but not all, of the missing static optimisations at runtime.

Unfortunately we were unable to benchmark the additional integer optimisations from Section 3.6: Tristate numbers are now too embedded into the RPython JIT to be easily disabled. To get an upper bound of the effect those optimisations have, we ran an ablation



Benchmark	KIPS mcf.0	KIPS perlbench.0	slowdown mcf.0	slowdown perlbench.0
Implementation				
full	22320.79	15203.28	1.00	1.00
no-jit-integer-optimisations*	12488.84	8454.55	1.79	1.80
no-opt-with-jit*	4912.80	2898.55	4.54	5.25
no-jit	1820.04	1776.76	12.26	8.56
no-static-opt*	172.60	170.92	129.32	88.95

Figure 11 JIT-relevant ablation results. The y-axis of the plot is logarithmic.

`no-jit-integer-optimisations` that disables *all* integer optimisations of the RPython JIT (also in Figure 11). Doing that gives a slowdown of 1.79× and 1.80×. Since the RPython JIT also had integer optimisations before the changes motivated by Pydrofoil (including a range analysis done by the JIT at runtime), we expect the effect of Tristate numbers to be below that number. Therefore it is a less important optimisation compared to any of the Pydrofoil static optimisation and compared to having a JIT in the first place.

4.3 Comparison with QEMU and Spike

To assess the practical performance of Pydrofoil, we benchmark against two established, industrial-strength ISSs. Due to the limited availability of ISS generators capable of compiling Sail specifications, this comparison uses simulators not derived from Sail. We selected QEMU [11] and Spike[13]. QEMU is a widely adopted ISS, implemented in C and incorporating significant manual optimisation. It supports multiple prominent ISAs, including RISC-V, and functions as a core component within various processor simulation frameworks. QEMU utilises a JIT compilation engine that translates guest instructions into host instruction sequences with manually developed, ISA-specific translators. Over two decades of development effort have contributed to QEMU's high simulation performance. While not designed for formal verification, its widespread adoption positions QEMU as a practical performance ceiling for Pydrofoil. Conversely, Spike is manually implemented in C++ specifically for the RISC-V ISA and is widely used within the RISC-V ecosystem. It operates as an interpreter without a JIT compiler, enhancing execution speed through the caching of decoded instructions.

We use SPEC and booting Linux as benchmarks, and RISC-V as ISA.

[13] `https://github.com/riscv-software-src/riscv-isa-sim`

Benchmark	time pydrofoil	time qemu	time spike	speedup qemu	speedup spike
600.perlbench_s.0	25462s	1121s	14075s	22.7×	1.8×
600.perlbench_s.1	49089s	760s	13505s	64.6×	3.6×
600.perlbench_s.2	14026s	692s	14232s	20.3×	1.0×
602.gcc_s.0	31757s	993s	16619s	32.0×	1.9×
602.gcc_s.1	35479s	651s	11263s	54.5×	3.2×
602.gcc_s.2	30758s	627s	10455s	49.1×	2.9×
605.mcf_s.0	13248s	6095s	30219s	2.2×	0.4×
620.omnetpp_s.0	58203s	1267s	33816s	45.9×	1.7×
623.xalancbmk_s.0	14044s	3456s	32109s	4.1×	0.4×
625.x264_s.0*	15664s	278s	7281s	56.3×	2.2×
625.x264_s.1*	92698s	947s	1331s	97.9×	69.7×
625.x264_s.2*	69464s	768s	1953s	90.5×	35.6×
631.deepsjeng_s.0	39604s	1749s	34459s	22.6×	1.1×
641.leela_s.0	34928s	1738s	21686s	20.1×	1.6×
648.exchange2_s.0	17023s	1159s	12661s	14.7×	1.3×
657.xz_s.0	49566s	2603s	47487s	19.0×	1.0×
Geometric mean				26.7×	2.3×

Figure 12 Performance comparison of QEMU, Spike and Pydrofoil on SPEC benchmarks, in seconds. Speedup is the factor by which QEMU, resp. Spike, is faster than Pydrofoil. All benchmark times include booting Linux. Benchmarks marked with * also include the time to generate their input files.

Results. This benchmark uses the AMD Ryzen 9 setup. The results are in Figure 12. We see that QEMU's performance is significantly ahead of Pydrofoil's: QEMU is between 2× and 100× faster than Pydrofoil, with a geometric mean of 26.7× . This is clearly disappointing, it shows that for many cases Pydrofoil is not good enough at removing the overhead of Sail. Section 5 gives concrete suggestions on how to reduce the performance gap with QEMU.

Spike also generally outperforms Pydrofoil. The relative speedup of Spike compared to Pydrofoil ranges from 0.4× (so Pydrofoil is faster) to 96.7×. The geometric mean of the speedup across all benchmarks is 2.3×. Pydrofoil exhibits significantly lower performance relative to Spike on the `625.x264_s.1/2` benchmarks, in particular. The underlying causes of these performance differences, especially the slowdowns observed for Pydrofoil, warrant future investigation.

4.4 Preliminary benchmarks for Pydrofoil-Arm and Pydrofoil-CHERIoT

All benchmarks above use RISC-V. This raises the question how dependent on RISC-V the performance numbers are. This benchmark presents preliminary performance results for two other ISAs, Arm v9.4-a [52] and CHERIoT [1, 2, 19]. For both ISAs, we compared the performance of the Pydrofoil-generated ISS with that of the Sail C backend.

The Arm v9.4-a specification in Sail is obtained by transpilation from Arm's ASL [51] specification, and is *substantially* bigger and more complex than RISC-V. For example the JIB obtained by translation of the Arm model is 20× larger than the corresponding RISC-V. As Arm v9.4-A benchmark, we boot Linux 6.0.7, but only up to the initialisation of the `init` process. This executes 24.6 million instructions.

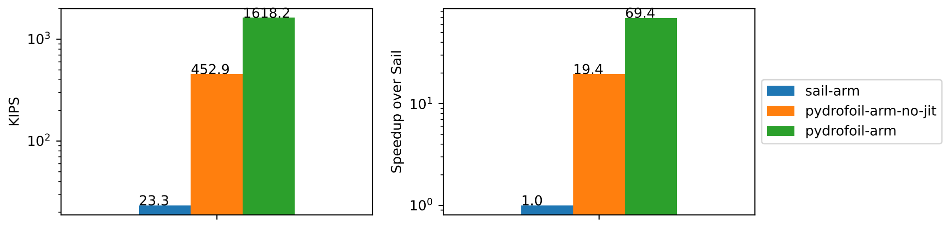


Figure 13 Benchmark results for the Arm ISS, booting Linux. The left plot shows KIPS, the right the speedup over Sail. Higher is better.

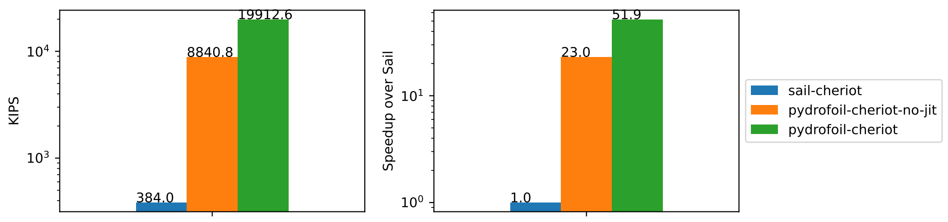


Figure 14 Benchmark results for the CHERIoT ISS, running the allocation-benchmark. The left plot shows KIPS, the right the speedup over Sail. Higher is better.

In contrast to the complex Arm ISA, CHERIoT is a simple 32-bit RISC-V micro-controller without virtual memory and caches, co-designed with a real-time operating system (RTOS), all with a strict focus on security. CHERIoT is a capability machine, based on (a small variant of) CHERI [78]. Capabilities can be seen as a hardware implementation of "fat pointers": conventional integer pointers are replaced by capabilities, which are memory addresses together with permissions constraining capability use. This enables the processor to check every memory access to ensure that it is not out-of-bounds (spatial memory safety). Capabilities can be used to mitigate a wide range of memory safety bugs and vulnerabilities. CHERIoT comes with a Sail specification of its ISA semantics [46], and the ISS that Sail generates is currently the only available ISS for CHERIoT. For CHERIoT we boot its RTOS [45] and then run CHERIoT's "allocation-benchmark"[14]. Booting the RTOS takes 490,000 instructions, while the "allocation-benchmark" takes 274 million.

Results. This benchmark uses the AMD Ryzen 7 setup. The results for Arm are in Figure 13. While Pydrofoil still manages to produce a much faster ISS for Arm than the Sail C backend (by a factor of almost 70×), in terms of instructions per second the performance is significantly worse than that of the RISC-V variant.

The CHERIoT results are in Figure 14. The speedup of Pydrofoil-CHERIoT is about 50×, even lower than that of Arm. This seems to be at least partly caused by Pydrofoil's poor handling of the fact that the CHERIoT-Sail-model uses a Sail `struct` type to store the CHERIoT capability registers (in RISC-V and Arm most registers are bitvectors).

[14] The "allocator-benchmark" can be found in [44]. For each power of 2 from 32 to 131072, the benchmark tries to allocate 1 MB with `malloc`, and then deallocates it using `free`. It also prints (to the emulated character device) the time it took to allocate and deallocate each 1 MB.

4.5 Threats to validity

This study, while providing insights into Pydrofoil's performance, is subject to several threats to validity. The primary limitation stems from the computational intensity of benchmarking ISSs, which operate at significantly lower speeds than the hardware they model. A comprehensive evaluation of Pydrofoil across a broad range of benchmarks requires computational resources and time that exceed our current capacity. Consequently, our findings are based on a subset of representative benchmarks, potentially limiting the scope of our conclusions. Second, the selected benchmarks, while intended to be representative of typical computer architecture workloads, were not specifically designed for ISS evaluation. This misalignment may not fully exercise Pydrofoil's capabilities, potentially under-representing its strengths and weaknesses. We acknowledge the need for benchmarks specifically tailored to evaluate ISS generators. Third, to mitigate limitations in existing tools, we employed certain experimental shortcuts. Notably, we did not perform multiple benchmark runs to assess statistical variation[15]. Finally, our primary focus on the RISC-V ISA introduces a potential limitation to the generalisability of our findings. While we included preliminary evaluations for Arm and CHERIoT, these are subject to methodological constraints. The Arm Sail model, derived from ASL through transpilation, may introduce runtime checks due to ASL's weaker typing, potentially confounding the attribution of performance characteristics. This highlights the challenges of comparing ISSs derived from different specification languages. The CHERIoT evaluation is based on a single, small-scale benchmark, whose results may not generalise well. Nevertheless, we believe that our work demonstrates that ISA specifications in Sail can be compiled into performant ISSs.

4.6 Summary of benchmarking

To address research questions RQ2 and RQ3, we conducted extensive benchmarking, aiming to quantify Pydrofoil's performance. Our ablation studies demonstrate that Pydrofoil's hybrid architecture, combining AOT and JIT compilation with domain-specific optimisations, is a viable approach for accelerating ISSs generated from Sail. Our smaller-scale benchmarks across Arm and CHERIoT ISAs corroborate the trends observed with RISC-V, giving us additional confidence in the robustness of our approach. However, comparisons with QEMU highlight a significant remaining performance gap, indicating substantial room for improvement to match the speed of hand-crafted full-system emulators.

Notably, the considerable resource demands of these benchmarks suggest a need for more efficient and feasible benchmarking strategies.

5 Suggestions for future research

While the present work has demonstrated significant performance gains, achieving a >250× speedup over the Sail-generated ISS, Pydrofoil remains 26.7× slower than QEMU. Spink et al. (2020) [60] report a 2× speedup over QEMU by employing a combination of AOT and partial evaluation techniques, similar in principle to Pydrofoil, albeit compiling from a low-level ADL, which is easier to optimise for than Sail. These discrepancies highlight the potential for further optimisation, and motivate the roadmap presented below, which is intended to bring Pydrofoil to performance parity with QEMU. It might also provide insights for the development of high-performance ISS generators for other high-level ADLs.

[15] Due to the required computational intensity. We hope to fix this in later versions of this work.

Clocks. Simulation of the processor clock is a known inefficiency for ISS acceleration [58]. The RISC-V Sail model ticks its clock every 100 instructions. This number was chosen arbitrarily, and is *not* part of the RISC-V instruction set architecture [76, 77]. Checking whether the next clock tick is reached in every guest instruction is expensive. Moreover, the JIT must generate control flow edges from *all* instruction to the clock logic. QEMU optimises this by only checking for clock ticks at the start of generated basic blocks, and deoptimises whenever a timer event happens elsewhere [48]. It is interesting to investigate if QEMU's approach or the mitigations discussed in [58] improve Pydrofoil performance.

Address translation. Real processors cache the results of address translation in the translation lookaside buffer (TLB), a dedicated hardware cache. Simulation has no TLB, so every memory access must simulate a full page table walk. This is a known problem in ISS research [11, 59]. We believe that a JIT with suitable domain-specific optimisations can lower the cost of simulated page table walks, for example by exploiting the predictability arising from contiguous addresses.

JIT warmup. The time it takes the JIT to generate optimised machine code is called warmup [10], and is the result of a fine balance between the cost and benefits of optimisations. PyPy's tracing mechanism was optimised for general purposed workloads: in PyPy, one Python bytecode produces 6-7 trace operations on average, which the optimiser reduces to about 2. Contrast this with Pydrofoil: tracing one RISC-V guest instruction produces more than 600 trace operations, which are optimised down to about 65 on average. The trade-offs driving the JIT optimisation design in PyPy are likely to be suboptimal for ISS workloads, hence have significant potential for improvement.

Type-erasure. Sail's liquid types can express strong constraints on bitvector widths. Sail's current compilation chain mostly erases them. Consider `size_bytes` from Section 3.2.4 with signature $\texttt{size\_bytes} : \texttt{word\_width} \leftrightarrow \{1, 2, 4, 8\}$. In other words, any value obtained by calling `size_bytes` is constrained to values $1, 2, 4$ or 8 in the Sail source. This strong constraint is lost in JIB: all the Pydrofoil AOT sees is that `size_bytes` returns `%i64`. If we want to optimise further, the fact that `%i64` stores one of $1, 2, 4, 8$ needs to be re-discovered by static analysis, either at compile-time, or by the JIT at run-time. Such strong constraints are prevalent in ISAs. Given the increasing interest in liquid types in other languages [36, 70], we believe that a better exploitation of the information from liquid types in the compilation toolchain might deliver better optimisations even beyond Sail.

6 Related work

6.1 DSLs used in computer architecture

There are a variety of DSLs that have been created for processor descriptions. Most relevant here are the aforementioned *architecture description languages*. Examples include nML [22], ISDL [28], EXPRESSION [29], SystemC [32], ArchC [54], and GenC [72]. They are often described as being high-level, and are actively used to derive ISSs, especially in commercial tools. Compared to Sail, and with the exception of GenC, they all betray their origins in layers of the processor design stack lower than ISA: they invariably also contain facilities to describe low-level detail, whether pipeline layout, timing constraints, or data-path conflicts. In their context of origin, computer architecture, they are considered higher-level, but only by contrast with RTL languages, such as Verilog and VHDL. Such DSLs are often based on C/C++, a notable exception being SSL [20], which specifies the ISA semantics of machine instructions in Object-Z (an extension of Z specification language). However, this approach has not gotten much traction. It is only with Arm's ASL [51], Sail's most important predecessor, that a mature, ISA-only DSL emerged.

6.2 ISS acceleration

Evaluating Pydrofoil's performance against existing ISS solutions presents challenges due to the lack of directly comparable systems that utilise Sail. Proprietary tools such as Arm's FastModel [38] and Architecture Envelope Models [4], offer high-performance simulation capabilities. However, their internal architectures, including the extent to which they are automatically generated from Arm's ASL, remain largely undisclosed. Similarly, commercial tools from Synopsys (e.g., ASIP Designer [65], Platform Architect [67], ImperasDV [57, 66]), MachineWare [16, 26], and Codasip [21] rely on handwritten components or generation from lower-level ADLs like SystemC, nML, or Codal [21]. Many academic ISSs, while valuable, have not seen recent publications or updates to their codebase, making comparison difficult.

Pydgin. Pydgin [39], another meta-tracing and PyPy-based ISS, simulates only user-mode execution. This simplification omits features such as address translation, self-modifying code, and interrupt handling. Furthermore, Pydgin utilises its own Architecture Description Language, which offers a lower level of abstraction compared to Sail. These differences render direct performance comparisons difficult. In addition, due to the lack of recent updates to the Pydgin codebase, Pydgin no longer builds "out of the box". Even after resolving these build issues, we found that the resulting simulator could not correctly load and execute our SPEC benchmarks. For these reasons, we excluded Pydgin from our performance evaluation.

GenSim from the University of Edinburgh. GenSim [60, 72–74] provides a retargetable and JIT-accelerated ISS framework. The GenSim toolchain, using the GenC specification language, and Captive dynamic binary translator (DBT) achieve a performance speedup of 2× over QEMU [61]. The performance gains are attributed to: (i) hardware-accelerated guest memory address translation [59]; (ii) partial evaluation [73], a technique related to meta-tracing [13, Chapter 12]; (iii) DBT optimised for ISS workloads [61]; (iv) parallelisation [35]; (v) use of a DSL with performance-focused constructs. We believe most of these techniques are general over the JIT/DBT system and thus applicable to Pydrofoil as an avenue for future work. However, while the hardware-acceleration of guest memory address translation provides substantial performance improvement, the implementation used by Captive is not possible in Pydrofoil. The technique requires arranging host page tables so guest address translations can be performed by the host's memory management unit. This use of hardware translation is faster than the simulation of multiple levels of page table walks in software. Unfortunately, it is only suitable for systems where the address space can be fully managed (Captive hosts the DBT in a bare-metal virtual machine, to gain full and unrestricted access to the host machine's hardware); as a user-mode application Pydrofoil is incapable of that. Additionally, this would harm one of Pydrofoil's features by no longer fully simulating the precise model provided, and instead requiring ahead-of-time knowledge of the guest's memory translation regime and a separate implementation for each host-guest architecture combination. The performance implications of compiling from Sail, compared to lower-level languages, remain a subject of ongoing investigation. A tool was made to translate Sail to GenC, forming a Sail frontend for GenSim, but was unsuccessful. The internal architecture of the GenSim toolchain scales poorly with input model size; the handwritten models are 20,000 lines of

GenC and compile in minutes, but attempts to use Sail translated to GenC resulted in tens of millions of lines and failed to compile even after days. The same issue occurred when developing a tool to translate Sail into Rust; the large volume of code being emitted caused the Rust compiler to use an unsustainable amount of memory. This appears to be a consistent challenge faced when working with high-level, machine-generated Sail specifications.

7 Conclusion

This paper introduces Pydrofoil, a tool designed to enhance the performance of ISSs based on Sail. We identify the primary bottlenecks of the current Sail system in Sections 2 and 3, addressing our first research question (RQ1). We propose a staged compiler architecture that utilises meta-tracing, as detailed in Section 3. Our benchmarks, presented in Section 4, demonstrate that Pydrofoil significantly improves simulation speed compared to Sail. However, in its current iteration, it does not yet reach the performance levels of a hand-tuned system like QEMU, answering RQ2 and RQ3. Future work aimed at closing this gap is outlined in Section 5, serving as a roadmap for achieving performance parity with QEMU.

References

1 Saar Amar, Tony Chen, David Chisnall, Felix Domke, Nathaniel Wesley Filardo, Kunyan Liu, Robert M. Norton, Yucong Tao, Murali Vijayaraghavan, Robert N. M. Watson, and Hongyan Xia. *CHERIoT Architecture specification*. CHERIoT Platform, version 0.6 (draft) edition, January 2025. URL: `https://cheriot.org/cheriot-sail/cheriot-architecture.pdf`.

2 Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert Norton, Simon W. Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. CHERIoT: Complete Memory Safety for Embedded Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, pages 641–653, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3613424.3614266`.

3 Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, pages 53–64, New York, NY, USA, 2007. Association for Computing Machinery. `doi:10.1145/1297081.1297091`.

4 Arm. Arm Architecture FVPs. `https://developer.arm.com/Tools%20and%20Software/Fixed%20Virtual%20Platforms/Arm%20Architecture%20FVPs`. Accessed February 2025.

5 Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, and Shaked Flur. The Sail instruction-set semantics specification language. `https://alasdair.github.io/`. Accessed February 2025.

6 Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2019. Proc. ACM Program. Lang. 3, POPL, Article 71. `doi:10.1145/3290384`.

7 Alasdair Armstrong, Brian Campbell, Ben Simner, Thibaut Perami, and Peter Sewell. Isla. `https://github.com/rems-project/isla`, Accessed Jan. 2025.

8 Krste Asanović and David A. Patterson. Instruction Sets Should Be Free: The Case For RISC-V. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley, August 2014. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html`.

9 Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A Transparent Dynamic Optimization System. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 46, February 2003. `doi:10.1145/349299.349303`.

10 Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual machine warmup blows hot and cold. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), October 2017. doi:10.1145/3133876.

11 Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, USA, 2005. USENIX Association.

12 Martin Berger and Laurence Tratt. Program Logics for Homogeneous Generative Run-Time Meta-Programming. *Logical Methods in Computer Science (LMCS)*, 11(1:5), March 2015. doi:10.2168/LMCS-11(1:5)2015.

13 Carl Friedrich Bolz. *Meta-Tracing Just-in-Time Compilation for RPython.* PhD thesis, Heinrich-Heine-Universität Düsseldorf, 2014.

14 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Allocation removal by partial evaluation in a tracing JIT. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '11, pages 43–52, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1929501.1929508.

15 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1565824.1565827.

16 Nils Bosbach, Niko Zurstraßen, Rebecca Pelke, Lukas Jünger, Jan Henrik Weinstock, and Rainer Leupers. Towards High-Performance Virtual Platforms: A Parallelization Strategy for SystemC TLM-2.0 CPU Models. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, DAC '24, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3649329.3658257.

17 James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 41–42, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3185768.3185771.

18 Corrado Böhm and Giuseppe Jacopini. Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules. *Commun. ACM*, 9(5):366–371, May 1966. doi:10.1145/355592.365646.

19 CHERIoT Platform. https://cheriot.org/. Accessed February 2025.

20 C. Cifuentes and S. Sendall. Specifying the Semantics of Machine Instructions. In *Proceedings of the 6th International Workshop on Program Comprehension*, IWPC '98, page 126, USA, 1998. IEEE Computer Society.

21 Codasip. What is CodAL? https://codasip.com/2021/02/26/what-is-codal/. Accessed February 2025.

22 A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *Proceedings the European Design and Test Conference. ED&TC 1995*, pages 503–507, 1995. doi:10.1109/EDTC.1995.470354.

23 Joseph Allen Fisher. *The optimization of horizontal microcode within and beyond basic blocks: an application of processor scheduling with resources.* PhD thesis, New York University, USA, 1979. AAI8010348.

24 Joseph Allen Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. Comput.*, 30(7):478–490, July 1981. doi:10.1109/TC.1981.1675827.

25 John Nathan Foster. *Bidirectional Programming Languages.* PhD thesis, University of Pennsylvania, 2009. URL: https://repository.upenn.edu/handle/20.500.14332/32277.

26 MachineWare GmbH. Virtual Components Modeling Library (vcml). https://github.com/machineware-gmbh/vcml, Accessed February 2025.

27 Michael Gordon, Robin Milner, Lockwood Morris, Malcom Newey, and Christopher Wadsworth. A Metalanguage for Interactive Proof in LCF. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 119–130. ACM, 1978. doi:10.1145/512760.512773.

28 George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: an instruction set description language for retargetability. In *Proceedings of the 34th Annual Design Automation Conference*, DAC '97, pages 299–302, New York, NY, USA, 1997. Association for Computing Machinery. doi:10.1145/266021.266108.

29 A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: a language for architecture exploration through compiler/simulator retargetability. In *Design, Automation and Test in Europe Conference and Exhibition, 1999. Proceedings (Cat. No. PR00078)*, pages 485–490, 1999. doi:10.1109/DATE.1999.761170.

30 David Harel. On folk theorems. *Commun. ACM*, 23(7):379–389, July 1980. doi:10.1145/358886.358892.

31 John Hauser. Berkeley SoftFloat. https://www.jhauser.us/arithmetic/SoftFloat.html. Accessed March 2025.

32 IEEE Standard for Standard SystemC Language Reference Manual, 2023. doi:10.1109/IEEESTD.2023.10246125.

33 RISC-V International. ISA Formal Spec Public Review Public. https://github.com/riscvarchive/ISA_Formal_Spec_Public_Review. Accessed February 2025.

34 RISC-V International. RISC-V Sail Model. https://github.com/riscv/sail-riscv. Accessed February 2025.

35 Stephen Kyle, Igor Böhm, Björn Franke, Hugh Leather, and Nigel Topham. Efficiently parallelizing instruction set simulation of embedded multi-core processors using region-based just-in-time dynamic binary translation. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 21–30, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2248418.2248422.

36 Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. Flux: Liquid Types for Rust. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi:10.1145/3591283.

37 Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. Implicit parameters: dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 108–118, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/325694.325708.

38 Arm Limited. Fast Models: Develop Software without Hardware. https://www.arm.com/products/development-tools/simulation/fast-models. Accessed February 2025.

39 Derek Lockhart, Berkin Ilbeyi, and Christopher Batten. Pydgin: generating fast instruction set simulators from simple architecture descriptions with meta-tracing JIT compilers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267, 2015. doi:10.1109/ISPASS.2015.7095811.

40 Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised Edition)*. MIT Press, 1997.

41 Prabhat Mishra and Nikil Dutt. *Processor Description Languages*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

42 James George Mitchell. *The design and construction of flexible and efficient interactive programming systems*. PhD thesis, Carnegie Mellon University, USA, 1970. AAI7104538.

43 Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. Simplicitly: foundations and applications of implicit function types. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158130.

44 CHERIoT Platform. alloc.cc. https://github.com/CHERIoT-Platform/cheriot-rtos/blob/44c1bdeba1278354ec379e36d87fa33c6dc10b0c/benchmarks/allocation/alloc.cc. Accessed February 2025.

45 CHERIoT Platform. CHERIoT RTOS. `https://github.com/CHERIoT-Platform/cheriot-rtos/`. Accessed February 2025.

46 CHERIoT Platform. Cheriot specification and sail model. `https://github.com/CHERIoT-Platform/cheriot-sail`. Accessed February 2025.

47 QEMU. System Emulation. `https://www.qemu.org/docs/master/system/index.html`. Accessed March 2025.

48 QEMU.org. TCG Instruction Counting. `https://www.qemu.org/docs/master/devel/tcg-icount.html`. accessed February 2025.

49 Norman Ramsey. Beyond Relooper: recursive translation of unstructured control flow to structured control flow (functional pearl). *Proceedings of the ACM on Programming Languages*, 6(ICFP):1–22, August 2022. `doi:10.1145/3547621`.

50 Allison Randal. The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers. *ACM Comput. Surv.*, 53(1), February 2020. `doi:10.1145/3365199`.

51 Alastair Reid. Trustworthy specifications of ARM v8-A and v8-M system level architecture. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pages 161–168, 2016. `doi:10.1109/FMCAD.2016.7886675`.

52 Cambridge University REMS Team. Github repo sail-arm. `https://github.com/rems-project/sail-arm`. Accessed February 2025.

53 Armin Rigo and Samuele Pedroni. PyPy's approach to virtual machine construction. In *DLS*, Portland, Oregon, USA, 2006. ACM. `doi:10.1145/1176617.1176753`.

54 S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. ArchC: a systemC-based architecture description language. In *16th Symposium on Computer Architecture and High Performance Computing*, pages 66–73, 2004. `doi:10.1109/SBAC-PAD.2004.8`.

55 RISC-V model: `riscv_insts_base`. `https://github.com/riscv/sail-riscv/blob/2dfc4ff9f2bed3dcd0a3e8748211c99099e70ab7/model/riscv_insts_base.sail#L159`. Accessed February 2025.

56 Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. *SIGPLAN Not.*, 43(6):159–169, June 2008. `doi:10.1145/1379022.1375602`.

57 Imperas Software. riscvOVPsim: A Complete, Fully Functional, Configurable RISC-V Simulator. `https://github.com/riscv-ovpsim/imperas-riscv-tests/tree/v20230724/riscv-ovpsim`. Accessed February 2025.

58 Tom Spink, Harry Wagstaff, and Björn Franke. Efficient asynchronous interrupt handling in a full-system instruction set simulator. In *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*, LCTES 2016, pages 1–10, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2907950.2907953`.

59 Tom Spink, Harry Wagstaff, and Björn Franke. Hardware-Accelerated Cross-Architecture Full-System Virtualization. *ACM Trans. Archit. Code Optim.*, 13(4), October 2016. `doi:10.1145/2996798`.

60 Tom Spink, Harry Wagstaff, and Björn Franke. A Retargetable System-level DBT Hypervisor. *ACM Trans. Comput. Syst.*, 36(4), May 2020. `doi:10.1145/3386161`.

61 Tom Spink, Harry Wagstaff, Björn Franke, and Nigel Topham. Efficient code generation in a region-based dynamic binary translator. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '14, pages 3–12, New York, NY, USA, 2014. Association for Computing Machinery. `doi:10.1145/2597809.2597810`.

62 Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 165–174, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/2544137.2544157`.

63 Standard performance evaluation corporation (SPEC) homepage. `https://www.spec.org/spec/`. Accessed February 2025.

64 Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. Dynamic native optimization of interpreters. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, IVME '03, pages 50–57, New York, NY, USA, 2003. Association for Computing Machinery. doi:10.1145/858570.858576.

65 Synopsys. ASIP Designer. https://www.synopsys.com/dw/ipdir.php?ds=asip-designer. Accessed February 2025.

66 Synopsys. ImperasDV: RISC-V Processor Verification Made Easy. https://www.synopsys.com/verification/imperasdv.html. Accessed February 2025.

67 Synopsys. Platform Architect: SoC architecture analysis and optimization for performance and power. https://www.synopsys.com/verification/virtual-prototyping/platform-architect.html. Accessed February 2025.

68 Jubi Taneja, Zhengyang Liu, and John Regehr. Testing static analyses for precision and soundness. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '20, pages 81–93, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3368826.3377927.

69 GMP Development Team. GMP: The GNU Multiple Precision Arithmetic Library. https://gmplib.org/. Accessed February 2025.

70 Niki Vazou. *Liquid Haskell: Haskell as a Theorem Prover*. PhD thesis, UC San Diego, 2016. URL: https://escholarship.org/uc/item/8dm057ws.

71 Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Sound, precise, and fast abstract interpretation with tristate numbers. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '22, pages 254–265. IEEE Press, 2022. doi:10.1109/CGO53902.2022.9741267.

72 Harry Wagstaff. *From High Level Architecture Descriptions to Fast Instruction Set Simulators*. PhD thesis, School of Informatics, University of Edinburgh, 2015.

73 Harry Wagstaff, Miles Gould, Björn Franke, and Nigel Topham. Early partial evaluation in a JIT-compiled, retargetable instruction set simulator generated from a high-level architecture description. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2013. doi:10.1145/2463209.2488760.

74 Harry Wagstaff, Tom Spink, and Björn Franke. Automated ISA branch coverage analysis and test case generation for retargetable instruction set simulators. In *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '14, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2656106.2656113.

75 Andrew Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, EECS Department, University of California, Berkeley, January 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html.

76 Andrew Waterman and Krste Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V International, December 2019. Document Version 20191214-draft.

77 Andrew Waterman and Krste Asanović, editors. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. RISC-V International, December 2021. Document Version 20211203.

78 Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9). Technical Report UCAM-CL-TR-987, University of Cambridge, Computer Laboratory, September 2023. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-987.pdf.

79 Wikipedia contributors. Initial ramdisk — Wikipedia, the free encyclopedia, 2025. [Online; accessed 2-March-2025]. URL: `https://en.wikipedia.org/w/index.php?title=Initial_ramdisk&oldid=1277729657`.

80 Hongwei Xi. Dependent ML: An approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, March 2007. `doi:10.1017/S0956796806006216`.

81 Alon Zakai. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, OOPSLA '11, pages 301–312, New York, NY, USA, October 2011. Association for Computing Machinery. `doi:10.1145/2048147.2048224`.