Porting a JIT Compiler to RISC-V: Challenges and Opportunities

Quentin Ducasse, Guillermo Polito, Pablo Tesone, Pascal Cotret, Loïc Lagadec

Porting a JIT compiler to RISC-V: Challenges and Opportunities

Quentin Ducasse [1], Guillermo Polito [2], Pablo Tesone [2]
Pascal Cotret [1], Loic Lagadec [1]

[1] Laboratoire Lab-STICC - ENSTA Bretagne - 29200 Brest, France
`firstname.lastname@ensta-bretagne.org`
[2] CNRS, INRIA - Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
`firstname.lastname@inria.fr`

Abstract

The RISC-V Instruction Set Architecture (ISA) is an open-source, modular and extensible ISA. The ability to add new instructions into a dedicated core opens up perspectives to accelerate VM components or provide dedicated hardware IPs to applications running on top. However, the RISC-V ISA design is clashing on several aspects with other ISAs and therefore software historically built around them. Among them, the lack of condition codes and instruction expansion through simple instruction combination. In this paper we present the challenges of porting Cogit, the Pharo's JIT compiler tightly linked to the x86 ISA, on RISC-V. We present concrete examples of them and the rationale behind their inclusion in the RISC-V ISA. We show how those mismatches are solved through design choices of the compilation process or through tools helping development: a VM simulation framework to keep the development in a high-level environment for the most part, an ISA-agnostic test harness covering main VM functionalities and a machine code debugger to explore and execute generated machine code. We also present a way to prototype custom instructions and execute them in the Pharo environment.

1 Introduction

Managed programming languages use a *managed runtime environment* called a virtual machine (VM). Several well-known VMs (V8 for JavaScript, JVM for Java, CLR for C♯) first compile the code to an intermediate representation called *bytecodes* then interpret these bytecodes at runtime. Having this layer of abstraction allows an application to be compiled to this architecture-independent intermediate representation and benefit from the portability of the VM. When porting the VM to new architectures, while most of the process is abstracted by the compiler that generates the runtime environment executable, the generation of machine code through the Just-in-Time (JIT) compiler has to be rewritten for each new architecture as it needs to output machine code corresponding to the target architecture.

The RISC-V instruction set architecture defines *open-source* and *modular* specifications to design processors. In the historical mindset of *Restricted Instruction Set Computer*, it is simplified to facilitate processor design: for example, it lacks condition codes or multiple data address modes. This leads to simple processors that can cover a restricted amount of instructions depending on the chosen extensions. The extensibility of the ISA opens it to new possibilities

in terms of dedicated hardware tasks. Acceleration through specific IP cores in fields such as signal processing or machine learning is one of those possibilities. Another would be specific processor implementations of VM tasks such as garbage collection or security through new instructions or hardware mechanisms. Experiments in hardware-software co-design for virtual machines make this ISA an interesting architecture to support.

The two main motivations behind this port are: (1) use a complete VM on the RISC-V architecture and (2) define and experiment with custom instructions. However, the complexity leveraged from the processor has to be handled on the compiler side and might come at a considerable price. While RISC-V defends key design choices, it comes as a clash for compilers inspired from or supporting the x86 architecture. This mismatch has to be addressed when writing a JIT compiler for RISC-V. To leverage some work needed to port a VM to a new ISA, an *instruction set architecture (ISA)-agnostic test suite* has been developed around the Pharo VM and its JIT compiler Cogit [1]. This suite consists of around 1400 configurable tests that range from checks on simple one-bytecode recompilation to correct polymorphic inline cache [2] generation. The whole test suite keeps most of the development within the Pharo [3] environment itself to make the best usage of its powerful native tools such as the debugger or object inspector or the dedicated JIT code debugger. It has helped port the Pharo VM to the ARMv8 ISA without access to hardware supporting the instruction set. Once again, these tools helped through the development of the Pharo JIT compiler port to RISC-V. Quick prototyping of custom instructions ideas comes at a reduced cost thanks to the high-level development environment Pharo and its simulation framework. Apart from the official RISC-V specifications and well-known compiler implementations source code (such as `gcc` or `clang`), few articles have highlighted the cost of porting a compiler to the RISC-V architecture. Others VM have recently finished their port to RISC-V (V8 by the PLCT Lab) or are currently in development (LuaJIT or DartVM by the PLCT Lab) and several have it planned but not public (JikesRVM by Martin Maas or OpenJ9 that does not support JIT compilation yet).

This paper presents (1) the process of porting the Pharo JIT compiler Cogit to RISC-V, (2) the main clashes between the original compiler and intermediate representation with RISC-V, and (3) design choices and recommendations to overcome them. This paper presents the following contributions:

- An open-source implementation of the Cogit RISC-V JIT compiler.

- A presentation of the clashes compiler developers face when porting to the RISC-V ISA.

- A presentation of design choices and recommendations when dealing with the RISC-V architecture.

Section 2 of this paper presents the Pharo virtual machine components and the RISC-V instruction set. Section 3 highlights the main clashes between the RISC-V ISA and the Pharo JIT compiler and intermediate representation. It also presents design elements of the Pharo VM to patch them. Section 4 presents the main issues with the existing JIT compiler, with the development environment and how they could be extended and reworked. Section 5 hovers over the simulation environment of the Pharo virtual machine and the extension added for custom instructions. Section 6 concludes and outlines next steps for future work.

2 Background

2.1 Pharo VM

Pharo language: The Pharo language is an evolution of Smalltalk according to the Smalltalk-80 specification. It is a pure object-oriented dynamically typed language that revolves around

message passing as its base way to redirect control flow. It defends a simple syntax and extensions over the base Smalltalk language. It has been in development for more than ten years and is available for companies to build their project on either in web development, data analysis and visualization or user interfaces. At their core, these applications are executed on top of the Pharo VM.
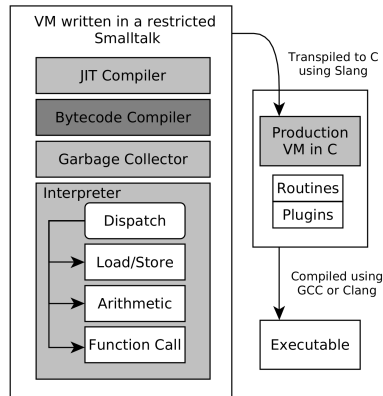

Figure 1: Slang VM transpilation.

Slang: The Pharo VM is a *meta-circular* VM written in Smalltalk and transpiled to C using a VM-specific translator called Slang [4]. Slang translates a group of classes into a single C file, transforming methods into functions. Slang restricts the source language from some features such as polymorphism or exceptions to translate it correctly to C code. The C code is then compiled along mandatory and optional plugins to an executable that runs on the desired architecture. The usage of Slang and translation to C has several advantages. It performs various interpreter optimizations such as the inlining of bytecode cases in the interpreter and implements threaded code [5]. Another key advantage is that it allows the VM developer to stay in the Pharo environment to simulate the Pharo VM, before translating it to C as its final step. This process is presented in Figure 1. This gives the developer full control over the VM source code and simplifies the development as Pharo tools (such as the debugger or object inspector) are available.

Cogit: The VM itself implements three main components: a threaded bytecode interpreter, a linear non-optimizing JIT compiler named Cogit [6] and a generational scavenger garbage collector. Diving deeper in Cogit, it is a method-based that uses a 2-address-code intermediate representation (IR) called CogRTL to compile the succession of bytecodes down to machine code. It does not model a control flow graph and compiles at the granularity of a method linearly, meaning the generated machine code mostly has a one-to-one mapping to the JIT IR. The CogRTL IR uses fixed virtual registers assigned ahead of time to physical registers for each backend to avoid the need of a complex register allocator. The compilation of a method comes in three phases:

- **(1) Bytecode scan phase** to extract metadata from the bytecodes (*e.g.* detection of message sends to inform the need for a frame).

- **(2) Bytecode parsing phase** to translate bytecodes into IR.

- **(3) Machine code generation phase** to translate IR into machine code (also called *concretization*).

This process is presented on Figure 2 and as shown, the first and second phases are *ISA-agnostic* and only the third phase has to be redefined for every new architecture the JIT
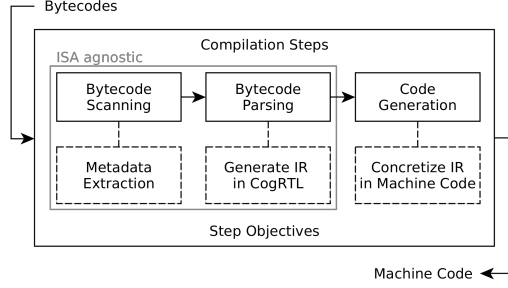
3

Figure 2: Cogit compilation phases.

compiler aims to support. It aims to be as machine-independent as possible as there is a clear distinction between the parsing and concretization phase.

Code patching: As Pharo is a dynamically typed object-oriented programming language, the JIT compiler also implements polymorphic inline caches [2] to improve performance. This means Cogit has to patch machine code without the knowledge obtained from bytecode scanning. Call sites are initially compiled as calls to dedicated trampolines send routines. The call site is then linked to the method itself once method lookup is performed. Cogit uses machine code stubs to rewrite the needed hook points: a call to a trampoline becomes a type-checked entry point, a *monomorphic cache*. If the type check fails, a new case is added making it a *polymorphic cache* up until a threshold where it is upgraded to a *megamorphic cache*. This patching method is also used by the garbage collector to update references to moved objects. Since these patching methods use machine code stubs, they are architecture-dependent and need to be reworked for each new architecture to support.

2.2 RISC-V

RISC-V is an *open-source* and *extensible* instruction set. This instruction set architecture (ISA) is gaining increasing attention from academics and industry as it focuses on simplicity and modularity. It has been designed with the objective and history of a *Restricted Instruction Set Computer (RISCV)* where the complexity is shifted away from the core and brought into the compiler. Examples are the lack of rotate instructions or overflow detection that have to be resolved in few other arithmetic operations. It also simplifies out-of-order processor execution by removing condition codes or instructions that depend on implicit state.

The RISC-V architecture consists of 32 general purpose registers and a program counter. Instructions are split in several extensions available in either 32 or 64 bits from which the main ones are: I for integer operations, M for integer multiplication and division, A for atomic operations, F for floating-point operations, C for compressed instructions, D for double precision operations and G as an equivalent for RV64IMAFD. The combination of RV64GC allows a RISC-V core to support a Linux distribution (*e.g.* Debian or Fedora support RISC-V).

While the above extensions have their specifications frozen, several others are still open to propositions. This is the case for extensions such as the J extension related to Dynamically Translated Languages [7]: it aims to provide specific instructions for Garbage Collectors or JIT engines. This particular extension is an opportunity to determine what would be a coherent choice of instruction(s) for a VM. The room for extensibility in the ISA allows adding dedicated machine code instructions in the generated RISC-V instructions. Extensions open up the space for hardware-related implementation ideas for the VM (*i.e.* garbage-collection or security) or acceleration with coprocessors (*i.e.* signal processing or machine learning). This space can be invested through quick prototyping of custom instructions.

4

3   Porting an x86-inspired compiler to RISC-V

The RISC-V Instruction Set Architecture (ISA) makes bold design decisions to leverage complexity from the processor. However, some of those decisions clash with expectations from existing ISAs such as x86 or ARM. RISC-V provides one data addressing mode and no complex call/return or stack instructions. Some of those design choices clash with Cogit historical architecture that revolved around x86 at first. In this section, we present the main design clashes between the RISC-V ISA and an x86-inspired JIT compiler.

3.1   Intermediate Representation Mismatch

The RISC-V instruction set, as MIPS, does not have flag registers and condition codes for instructions. The reason behind this choice is to simplify out-of-order execution as condition operands require the processor to use register renaming, which maps the register names in the program onto a larger number of internal physical registers. As out-of-order processors execute instructions opportunistically, it will map a physical register whether the condition holds. This extra operand increases the cost of the register file, register renamer and out-of-order execution hardware. RISC-V also gets rid of the well-known delayed branch of MIPS-32 or Oracle SPARC. They add extra state that is implicitly set by most instructions and complicates the dependence calculation of out-of-order execution.

In our case, as the CogRTL IR has been developed around the x86 ISA at first then extended to ARM and other ISAs, it supposed conditional registers were mandatory and revolved around it. This means it defines IR instructions that expect and implicit flag code from previous execution. A notable case is the succession of `CmpCqR`/`Jump<Condition>`. This succession compares the value of a register to a given quick constant (value that will be embedded in the immediate field of an instruction if possible). It then jumps accordingly with the corresponding condition and offset. The translation of these IR instructions to ARMv8 is presented in Listing 1. A one-to-one mapping between the IR and the generated machine code is possible.

Listing 1: ARMv8 machine code generation of a conditional jump.

```
# CogRTL instructions
cogit CmpR: ClassReg R: TempReg
cogit JumpNonZero: (Label 2).

# ARMv8 output
cmp r1, r22 # x1 and x22 are ARMv8 class/temp registers
b.ne 48     # label 2 has offset 48
```

However, RISC-V, following the same path as the MIPS ISA by only defining branches that compare the values of two registers. This way, `beq`, `bge`, `blt`, *etc.* need information from the IR instruction setting implicit state and the next one expecting it to make a decision. Compiling the IR instructions in Listing 1 to RISC-V should result in `bne x<class>, x<temp>, 48`, therefore extracting information from two IR instructions at compile time.

- *Rationale:* Conditional codes have a large impact on out-of-order execution performance on the hardware side as implicit state is used by instructions and should be kept through the pipeline.

- *Impact:* A mismatch between the JIT IR and corresponding machine code leads to a refactoring of either the IR itself, the redefinition of flag registers in the context of JIT compilation only or another processing step on top of the machine code generation phase to rewrite problematic instruction sequences.

3.2 Condition Codes Handling

RISC-V does not define condition codes and therefore flag registers. On the other hand, CogRTL IR is historically mapped on x86 instructions and expects implicit setting of those flags to perform, for example, the `JumpZero offset` intermediate representation instruction that expects a previous comparison that sets the zero register. To solve this issue, one way would be to redefine the whole CogRTL IR instruction set. While this would benefit the RISC-V or MIPS implementations, it would require a full rewriting of the existing x86, IA-32, ARMv5 and ARMv8 JIT implementations and therefore considerable development time.

Listing 2: Rewriting of conditional branching with no conditional codes.

```
CogCompiler >> noteFollowingConditionalBranch: nextInstruction

| newBranchLeft newBranchOpcode newBranchRight |

"Opcode extraction from the next instruction"
newBranchOpcode := nextInstruction opcode caseOf: {
  [JumpZero]      -> [BrEqualRR].
  [JumpNonZero]   -> [BrNotEqualRR].
  ...
} otherwise: [self unreachable. 0].

"Operands extraction from the current instruction"
opcode caseOf: {
  ...
  [CmpRR]   ->  [newBranchLeft := operands at: 1.
  newBranchRight := operands at: 0.
  opcode := Label].
  [CmpCqR]  ->  [newBranchLeft := operands at: 1.
  newBranchRight := TempReg.
  opcode := MoveCqR.
  operands at: 1 put: TempReg].
  ...
} otherwise: [self unreachable].

nextInstruction rewriteOpcode: newBranchOpcode with: newBranchLeft
   with: newBranchRight.
^ nextInstruction
```

Therefore, this mismatch is resolved either by redefining flag registers in the context of JIT compilation (*i.e.* assigning scratch registers to play the role of flag registers) or by refining the concretization process for architectures that do not have conditional codes. The first solution implies no additional change to the underlying compilation process but changes the mapping of one-to-one for IR to machine code to one-to-n as every arithmetic and logic instruction now comes with flag register settings. The other solution refines compilation process by notifying the concretization of a given IR instruction that there is a conditional instruction following. The mapping changes from one-to-one to two-to-one (or two-to-two in cases where a constant is involved). An example of this refinement and rewriting is presented in Listing 2.

Part of the method presented shows that when reaching a following conditional branch in the IR, the compiler gets notified and rewrites the instruction to new ones, passing the corresponding operands to the next instruction concretization. The first part extracts the new opcode from the following IR instruction. For example, a following `JumpZero` results in a `BrEqualRR`. The second part extracts operands from the current IR instruction. In the case of `CmpRR`, both registers are extracted and the `CmpRR` instruction is converted to a `Label` (therefore removed). In the case of `CmpCqR`, it is converted to a `MoveCqR` of the constant to a temporary

register, and this register along with the original ones are passed to the branch instruction.

3.3   Instruction Expansion

RISC-V main objective is to come as a modular ISA. All instructions are not available by default in the RV32I minimal set as it only contains 32-bits integer operations. The multiplication and division instructions have to be added explicitly through the M extension, floating-point operations in the F and 64-bits instructions in RV64. However, even when using RV64GC (RV64IMAFDC), several instructions are not available as they are in x86 or ARM. RISC-V rather relies on the combination of simple instructions to make up for a more complex one. This is the case for example of rotate instructions that have to be composed of three to four instructions. This is also the case for overflow checking in arithmetic operations.

Listing 3: rotate and software overflow checking.

```
# Rotate left with shift amount in register
rol:
  sll rd, rs1, rshamt      # x[rs1] << rshamt
  sub temp, zero, rshamt   # get the negative count
  srl temp, rs1, temp      # x[rs1] >> (xlen - rshamt)
  or rd, rd, temp          # or between (1) and (2)

# Software overflow check
addoverflow:
  add t0, t1, t2           # actual addition
  slti t3, t2, 0           # t3 = t2 sign
  slt t4, t0, t1           # t4 = sum smaller than t1
  bne t3, t4, overflow     # if t3 != t4, overflow
```

Listing 3 presents the instructions needed for a rotate instruction or the software checking of overflow on a general signed addition as proposed by Patterson et al [8]. While the rotate instruction `rol` will be present in the bit manipulation B extension, it is not, as the time of writing, ratified and integrated in processors.

- *Rationale:* Most but not all programs ignore integer overflow or rotations. Relying on software checking of overflows or rotations leaves room for new instructions on the hardware side.

- *Impact:* Increase in the number of instructions for the specific instructions that are not available and have to be derived in multiple instructions. This adds complexity in the generated machine code.

3.4   Sign-Extension and Its Implications

The main size unit in common operations is 12 bits for a given offset. This is the case for immediate arithmetical and logical instructions (`addi`, `andi`, `slti`, *etc.*), memory accesses (`lb`, `lwu`, *etc.*) as well as for offsets in branches (`beq`, `bge`, *etc.*) or the jump and link register. Exceptions to this size unit are the load upper immediate `lui` instruction that loads the 20 upper bits of an immediate and helps get 32-bits values in coordination with `addi`. In the same spirit, add upper immediate to program counter (`auipc`) is the key to use PC-relative addressing through those 20 upper bits. Finally, jump and link `jal` accepts a 21-bits offset and jump and link register `jalr` a 12-bits offset. All of these bricks combined allow for powerful operations and is the main resource for calls or jumps. However, all of the above instructions sign-extend the given offset. While this is useful for single instructions loading negative values,

it has to be taken in consideration in an instruction succession handling a value bigger than the offset size and with sign bits activated.

Listing 4: `call` pseudo-instruction and sign-extension.

```
CogRISCV64Compiler >> concretizeCall

"Compute offsets"
"12 lowest bits"
offsetLow := self
  computeSignedValue64Bits: (offset  bitAnd: 16rFFF).

"20 upper bits"
"11th bit correction for sign extension"
offsetHigh := (((self
  computeSignedValue64Bits: offset) + 16r800) >> 12)
  bitAnd: 16rFFFFF.

"Emit instructions"
"auipc"
self machineCodeAt: 0 put:
  (self addUpperImmediateToPC: offsetHigh toRegister: ConcreteIPReg).
"jalr"
self machineCodeAt: 4 put:
  (self jumpTo: ConcreteIPReg withOffset: offsetLow
     andStorePreviousPCPlus4in: LR).
^ machineCodeSize := 8
```

The concretization (*i.e.* machine code generation) of a `call` IR instruction to RISC-V machine code is presented in Listing 4. The 11th bit of the call target has to be verified manually to ensure it will come out as correct when combined with the next instruction, hence the addition to `0x800` that will cover the case where the 11th bit is set to one and the `jalr` instruction would sign-extend it. This impact on calls is also important in the code patching step as most code patching consists of updating call targets in JIT methods or Polymorphic Inline Caches.

- *Rationale:* Having sign-extension only enabled helps guide the specification towards a single way to encode immediate values.

- *Impact:* Instruction combination on large numbers needs a check and correction at the smallest size unit (12-bits here). This logic has to be added to the patching stage as well to correctly handle call sites updates.

3.5 Immediate Loading Expansion

As shown earlier, handling immediate values is not a simple task in RISC-V. To further complete this statement, we will look at the load immediate `li` pseudo-instruction available in RISC-V assembly. It is available in both 32 and 64 bits through the RV32I or RV64I extensions. Quoting the pseudo-instruction description, *it loads a constant in the destination register using as few instructions as possible. For RV32I, it expands to `lui` and/or `addi`; for RV64I, it can grow as long as `lui`, `addi`, `slli`, `addi`, `slli`, `addi`, `slli` and `addi`.* Therefore, a 64-bits wide immediate will need to go through a succession of adding 12-bits then shifting them until it can load the 20 upper bits. Note that, as presented earlier, offsets are sign-extended in `addi` and need to be corrected if needed. An example of a worst case scenario immediate loading is presented in Listing 5. The result is obtained after using `clang` and `llvm-objdump` on the

resulting object file. It handles the shifts, sign extension corrections and optimizations for sparse immediates.

Listing 5: `li` pseudo-instruction examples.

```
# Load long immediate to t0
li t0, 0x7FFF800800800800
# Expands to
lui     t0, 32768     # 0x8000
addiw   t0, t0, -2047 # 0x801
slli    t0, t0, 12
addi    t0, t0, -2047 # 0x801
slli    t0, t0, 12
addi    t0, t0, -2047 # 0x801
slli    t0, t0, 12
addi    t0, t0, -2048 # 0x800

# Load sparse immediate to t0
li t0, 0x70000000000007FF
# Expands to
addi    t0, zero, 7
slli    t0, t0, 60
addi    t0, t0, 2047 # 0x7FF
```

LLVM developers have defined a complex recursive function to handle all immediate values in the fewest instructions possible [9]. To quote the comment on their function: *In the following, constants are processed from LSB to MSB, but instruction emission is performed from MSB to LSB by recursively calling [the function]. In each recursion, first the lowest 12 bits are removed from the constant and the optimal shift amount, which can be greater than 12 bits if the constant is sparse, is determined. Then, the shifted remaining constant is processed recursively and gets emitted as soon as it fits into 32 bits. The emission of the shifts and additions is subsequently performed when the recursion returns.* GCC also defines several ways to encode a large immediate value and attribute a cost to each method before returning the best fitting choice [10].

- *Rationale:* Loading 64-bits immediate is rare and does not need a dedicated instruction. No official specifications are added to allow any developer to come with their own optimization.

- *Impact:* A single load can expand to up to 8 instructions and requires a complex logic behind its instruction generation. Compilers that store their immediate values embedded in instructions are severely impacted. Code patching of such values means that space for up to 8 instructions has to be allocated and the generation logic has to be launched each time a new immediate value has to be loaded. The logic to output the best sequence of instructions depending on the value is complex and needs to be executed at each regeneration or patch.

3.6 Out-of-Line Literals

Immediate values are literals that can be embedded in instructions. Pharo defines two types of JIT compilers: *inline literals* and *out-of-line literals* compilers. The first one will use all given literals as immediate values and therefore may need to split bigger literals through multiple instructions. While using inline literals is simple in the case of variable length instructions (CISC), it becomes trickier with fixed-length instructions (RISC). As presented earlier, RISC-V compilers use complex methods to determine the best way to encode large immediate values in

instructions and can inflate to up to eight instructions in the worst case. Out-of-line literals on the other hand mimic the `.data` section of assembly by putting the literals in nearby memory and accessible through a fixed-size succession of instructions. The two different designs are presented on Figure 3.
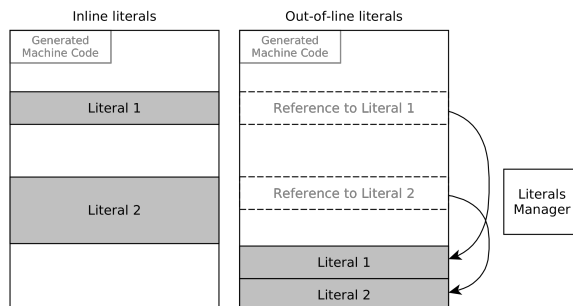

Figure 3: Inline and out-of-line literals.

From the Pharo side, Cogit uses a `LiteralsManager` to keep track of the literals that are needed in the nearby code region. This compiler is called `OutOfLineLiteralsCompiler` in opposition to the `InlineLiteralsCompiler` that embed all values in instructions. We define in a corresponding function `usesOutOfLineLiteral` the need or not for a literal during the concretization of a given IR instruction. The threshold is set at 12 bits after which an immediate value that could be embedded in instructions is promoted as a literal and handled by the `LiteralsManager`. Rather than having to handle up to 8 instructions, the compiler outputs a succession of `auipc/ld` to manage large values. This succession is always composed of two instructions and is patched accordingly if needed.

4 Current Issues and Planned Corrections

4.1 Issues with the Current Compiler Design

The Cogit design is tightly linked to its intermediate representation CogRTL that was defined to *optimize for the common case* and to design a JIT compiler mostly tailored to the Intel x86/x64 ISA. This enabled several short cuts: the internal code representation is not very abstract and resembles the x86/x64 ISA so much that the architecture-specific code generator directly transcribes the internal representation to machine code. As a result, compiling for x86 was simple. It also happened to work for ARMv8 which, despite becoming much closer to RISC/MIPS than ARMv7 ever was, retained some particularities of ARMv7 that keep it close enough to Intel (module number of registers), specifically branching on flags, many addressing modes (including PC-relative), and rare bit manipulation operations (rotations). While this worked for x86 and suffices for ARM, patching the JIT compiler for MIPS/RISC-V becomes a tedious task because it needs to overcome the clashes presented earlier and questions the CogRTL IR itself. While it has been simple by the past, it now drifts from its original form to be accepted by new ISAs that focus on their own rationale of simplicity, common case optimization and iterations from over 25 years of RISC architecture development and refinement. The patching iterations on an IR that are now too different questions its definition and while it implies a rewriting of all backends, it should be the correct choice.

4.2 IR Design Rework

The world of JIT IRs is complex and few articles cover the choice of their JIT IR. V8 uses a sea of nodes [11] with three levels of nodes from high-level Javascript operators that express semantics

of Javascript's overloaded operators down to machine operators that correspond closely to single machine instructions without effect on the overall graph. Between them, intermediate operators express VM-level operations such as allocation or bound checks. LuaJIT implements a linear, pointer-free 2-operand-normalized form SSA IR [12]. It presents higher-level instructions for branching through guarded assertions: they provide an assertion on their operands and are emitted by the backend as branching comparisons. The ability to quickly abstract a succession of machine code to cover a mapping that would be different between backends (branching in our case) is mandatory. Those two approaches are abstract enough to generate machine code for architectures from RISC-V to x86 without meeting the clashes we encountered with CogRTL. The design of a correct IR highlights the need for flexibility in the mapping to machine code as this will also impact optimization passes. In our case, the mapping can be reversed by defining a higher level IR instruction for branches with its operands (registers or immediates). It is then on the backend side to define either one instruction, in the case of a comparison between two registers and branch with an offset (*e.g.* `bne`) for RISC-V and MIPS or two if there is a need for a `cmp` instruction before a branch relying on conditional flags for x86 or ARM. Having this higher-level abstraction prevents us from the backtracking presented in MIPS and RISC-V architectures in Listing 2.

5   Extending the simulation environment

5.1   Existing VM Simulation Setup

Smalltalk and Pharo by extension is an object system. The entire system is stored in a snapshot file (called an image). A complete Pharo snapshot mostly consists of a memory dump of the entire heap, containing the entire system, including its development tools and application code. It includes objects, compiled methods as bytecodes, and running processes. Overall, this step consists of a scaling of the interactions tested separately in unit tests. The VM simulator (used to perform *full-system simulation* presented earlier) uses a particular memory mapping of the heap that contains the machine code zone, the different spaces (old and young) of the generational garbage collector as well as the C and Smalltalk stacks in a contiguous byte array. The Smalltalk objects of the VM are used as is in the Pharo environment instead of the `.text` and `.data` sections where their translated versions would be stored. The JITted code is executed through the processor emulator (Unicorn in our case).

The Pharo test harness uses the processor emulator to setup a simulated environment. It interacts with it by simulating some core features the VM will have to interact with such as *jumps*, *calls* or *register smashes*. It also redirects calls from the machine code to the interpreter by using fake addresses for trampolines or primitives the machine code might call, catching invalid memory accesses that those fake addresses trigger then redirecting to the corresponding simulator method in the Pharo environment before giving control back to the machine code.

5.2   Simulating a Custom Instruction

As RISC-V main interest lies in its modularity and extensibility, it is important to be able to run custom instructions. Custom instructions are defined in the Pharo environment by adding their corresponding behavior as a method on the processor simulator bindings, the same way calls or jumps are simulated. Unicorn provides hooks to attach behavior on the detection of given errors. The `UC MEM UNMAPPED` family is used to simulate trampolines or primitive sends as presented earlier. In our case, we use the `UC INSN INVALID` error and add a hook to redirect control flow.

The custom instruction simulation process proceeds as follows: (1) the interpretation loop begins in the VM simulator; (2) when a JIT-compiled method is encountered, control flow is handed over to the processor simulator to execute JIT code; (3) when reaching an undefined custom instruction, the simulator errors and triggers the associated hook; (4) this hook extracts the opcode and operands from the undefined instruction, checks if a corresponding simulation method exists and calls it; (5) Control flow is handed back over the custom instruction to the processor simulator. Finally, once the JIT code is executed, the control flow is handed to the interpreter in the Pharo environment.

Overall, this process requires the developer to: *define* the new machine code instruction; add it to the *concretization phase* when compiling a given IR instruction; add its opcode to the list of *simulated instructions* handled by Cogit; and define the *instruction behavior* in the architecture simulator bindings. ISA simulator generators such as Pydgin create the simulator along with custom instructions whose behavior is specified before generation. Since Unicorn is used as the main simulator for unit-tests-related simulations due to its capacity to hook invalid memory accesses, the instrumentation needed to run custom instructions comes at a relative low cost.

6 Conclusion

RISC-V is an open-source extensible ISA. Its modularity makes it interesting to experiment new features to accelerate application-specific code or parts of the execution engine. Virtual machines could benefit from changes in the underlying hardware implementation itself to accelerate some of its components such as the garbage collector or enforce strong security properties. It however comes with clashing design choices for compilers historically developed around the x86 ISA.

The mismatch between Pharo JIT IR and RISC-V is presented through four major aspects of RISC-V and the solutions added to Pharo either in terms of design choices or tools. A way to close the gap of mapping between the IR and machine code around conditional codes is presented and out-of-line literals allow us to avoid immediate loading through instructions. The test harness and simulation framework present around the Pharo VM leverage tests writing to check for architecture-specific tricks. The machine code debugger helps visualizing trampolines, IR and machine code mappings to ease the development process.

We presented a way to prototype custom instructions using hooks from the underlying processor simulator and redirecting the control to a Pharo method simulating the instruction's behavior. We want to use this method to experiment with RISC-V custom instructions both in VM-related topics such as security or garbage collection and applications such as media processing or machine learning. We would like to experiment some features from RISC-V custom instructions and test them before modifying the actual processor and surrounding components. Among these features, we want to implement a prototype of the RIMI [13] security model to protect JIT code through RISC-V custom instructions.

References

[1] G. Polito, P. Tesone, S. Ducasse, L. Fabresse, T. Rogliano, P. Misse-Chanabier, and C. Hernandez Phillips, "Cross-ISA testing of the Pharo VM: lessons learned while porting to ARMv8," in *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, 2021, pp. 16–25.

[2] U. Hölzle, C. Chambers, and D. Ungar, "Optimizing dynamically-typed object-oriented languages with polymorphic inline caches," in *ECOOP'91 European Conference on Object-*

Oriented Programming, P. America, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 21–38.

[3] A. P. Black, O. Nierstrasz, S. Ducasse, and D. Pollet, *Pharo by example*. Lulu. com, 2010.

[4] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay, "Back to the future: The story of Squeak, a practical Smalltalk written in itself," vol. 32, no. 10. New York, NY, USA: Association for Computing Machinery, Oct. 1997, p. 318–326. [Online]. Available: https://doi.org/10.1145/263700.263754

[5] M. A. Ertl and D. Gregg, "The structure and performance of efficient interpreters," *Journal of Instruction-Level Parallelism*, vol. 5, pp. 1–25, 2003.

[6] E. Miranda, "The Cog Smalltalk virtual machine," in *VMIL'11: Proceedings of the 5th workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*, 2011.

[7] A. Zabrocki, M. Maas, and S. Cetola, "Working draft of the RISC-V J extension specification," 2021. [Online]. Available: https://github.com/riscv/riscv-j-extension

[8] D. Patterson and A. Waterman, "The RISC-V Reader: An Open Architecture Atlas," Strawberry Canyon, Tech. Rep., 2015.

[9] LLVM, "LLVM li pseudoinstruction recursive function implementation," 2022. [Online]. Available: https://github.com/llvm/llvm-project/blob/4c3d916c4bd2a392101c74dd270bd1e6a4fec15b/llvm/lib/Target/RISCV/MCTargetDesc/RISCVMatInt.cpp

[10] GCC, "GCC li pseudoinstruction cost functions implementation," 2022. [Online]. Available: https://gcc.gnu.org/git/?p=gcc.git;a=blob;f=gcc/config/riscv/riscv.c;h=a545dbf66f734855090568ce1a253b373345eec0;hb=HEAD#l395

[11] B. L. Titzer, "Turbofan jit design," 2015. [Online]. Available: https://docs.google.com/presentation/d/1sOEF4MlF7LeO7uq-uThJSulJlTh--wgLeaVibsbb3tc

[12] G. Gillespie, "Luajit ssa ir 2.0," 2018. [Online]. Available: http://wiki.luajit.org/SSA-IR-2.0#introduction_example-ir-dump

[13] H. Kim, J. Lee, D. Pratama, A. M. Awaludin, H. Kim, and D. Kwon, "RIMI: instruction-level memory isolation for embedded systems on RISC-V," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.

[14] C. Heinz, Y. Lavan, J. Hofmann, and A. Koch, "A catalog and in-hardware evaluation of open-source drop-in compatible RISC-V softcore processors," in *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2019, pp. 1–8.

[15] F. Bellard, "QEMU, a fast and portable dynamic translator," in *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. Anaheim, CA: USENIX Association, Apr. 2005. [Online]. Available: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator

[16] A. Rigo and S. Pedroni, "PyPy's approach to virtual machine construction," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006, pp. 944–953.

[17] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis, "RPython: a step towards reconciling dynamically and statically typed oo languages," in *Proceedings of the 2007 symposium on Dynamic languages*, 2007, pp. 53–64.

[18] O. Nierstrasz, S. Ducasse, and D. Pollet, *Squeak by example.* Lulu. com, 2009.

[19] G. Bracha, P. v. d. Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda, "Modules as objects in Newspeak," in *European Conference on Object-Oriented Programming.* Springer, 2010, pp. 405–428.

[20] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon, "Maxine: An approachable virtual machine for, and in, Java," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–24, 2013.

[21] N. A. Quynh and D. H. Vu, "Unicorn: Next generation CPU emulator framework," *Black-Hat USA*, vol. 476, 2015.

[22] K. Asanovic, D. A. Patterson, and C. Celio, "The Berkeley out-of-order machine (BOOM): An industry-competitive, synthesizable, parameterized RISC-V processor," University of California at Berkeley Berkeley United States, Tech. Rep., 2015.

[23] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The Rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, 2016.

[24] C. Kotselidis, A. Nisbet, F. S. Zakkak, and N. Foutris, "Cross-ISA debugging in meta-circular VMs," in *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, 2017, pp. 1–9.

[25] E. Miranda, C. Béra, E. G. Boix, and D. Ingalls, "Two decades of Smalltalk VM development: Live VM development through simulation tools," in *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, ser. VMIL 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 57–66. [Online]. Available: https://doi.org/10.1145/3281287.3281295

[26] F. Kermarrec, S. Bourdeauducq, H. Badier, and J.-C. Le Lann, "LiteX: an open-source SoC builder and library based on Migen Python DSL," in *OSDA 2019, colocated with DATE 2019 Design Automation and Test in Europe*, 2019.

[27] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Nov 2019.

[28] A. Waterman and Y. Lee, "RISC-V instruction set manual," 2021. [Online]. Available: https://github.com/riscv/riscv-isa-manual

[29] R.-V. Foundation, "RISC-V Spike simulator," 2020. [Online]. Available: https://github.com/riscv-software-src/riscv-isa-sim

[30] SiFive, "HiFive1 Rev B," 2021. [Online]. Available: https://www.sifive.com/boards/hifive1-rev-b