Gigue: A JIT Code Binary Generator for Hardware Testing

Quentin Ducasse, Pascal Cotret, Loïc Lagadec

Gigue: A JIT Code Binary Generator for Hardware Testing

Quentin Ducasse
quentin.ducasse@ensta-bretagne.org
ENSTA Bretagne, Lab-STICC
Brest, France

Pascal Cotret
pascal.cotret@ensta-bretagne.fr
ENSTA Bretagne, Lab-STICC
Brest, France

Loïc Lagadec
loic.lagadec@ensta-bretagne.fr
ENSTA Bretagne, Lab-STICC
Brest, France

Abstract

Just-in-time compilers are the main virtual machine components responsible for performance. They recompile frequently used source code to machine code directly, avoiding the slower interpretation path. Hardware acceleration and performant security primitives would benefit the generated JIT code directly and increase the adoption of hardware-enforced primitives in a high-level execution component.

The RISC-V instruction set architecture presents extension capabilities to design and integrate custom instructions. It is available as open-source and several capable open-source cores coexist, usable for prototyping. Testing JIT-compiler-specific instruction extensions would require extending the JIT compiler itself, other VM components, the underlying operating system, and the hardware implementation. As the cost of hardware prototyping is already high, a lightweight representation of the JIT compiler code region in memory would ease prototyping and implementation of new solutions.

In this work, we present Gigue, a binary generator that outputs bare-metal executable code, representing a JIT code region snapshot composed of randomly filled methods. Its main goal is to speed up hardware extension prototyping by defining JIT-centered workloads over the newly defined instructions. It is modular and heavily configurable to qualify different JIT code regions' implementations from VMs and different running applications. We show how the generated binaries can be extended with three custom extensions, whose execution is guaranteed by Gigue's testing framework. We also present different application case generation and execution on top of a fully-featured RISC-V core.

Keywords: JIT, RISC-V, Hardware Development

1 Introduction

Efficient execution of high-level languages, whether statically or dynamically typed involves a set of complex components embedded in a Virtual Machine (VM), the execution engine of the source language. Among them are Java, Ruby, Python, JavaScript, or Pharo. Their VMs often involve a *parser*, an *intermediate representation compiler*, an *interpreter*, and a set of *just-in-time (JIT) compilers*. VMs provide portability for the developer as the distribution of the VM binary is enough to support any application, regardless of the underlying operating system (OS) and architecture. The VM profiles the executing source code, feeds frequently used parts to the JIT compilers that recompile it to machine code, and redirects control flow to the machine code representation when encountered again. JIT compilers output machine code at run time for the current architecture, and they need to be extended to support new architectures. At the same time, they are responsible for the gain in performance over sole interpretation. We believe they are particularly suited for hardware acceleration, either through dedicated hardware units implementing JIT-specific helpers (arithmetic, object offset access, *etc.*) or security mechanisms (*e.g.* trusted execution environments [17]).

RISC-V is an open-source Instruction Set Architecture (ISA) that leads to open-source implementations and allows developers to propose new instruction subsets in a reserved space [19]. This provides instruments to define and prototype application-specific custom instructions.

However, when designing custom instructions for the JIT compiler to emit, a huge technology stack bridges the VM JIT compiler to the actual custom instruction support in hardware. Figure 1 presents the main elements involved.

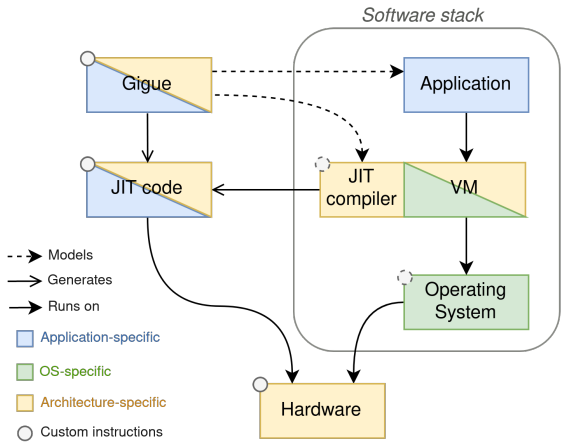

Figure 1. JIT-oriented hardware development.

On top of the hardware lives the operating system, providing base capabilities to the VM, itself distributed for the

OS/architecture combination. Adding custom instructions would require the extension of the VM and OS along with the hardware itself (dotted circles on the figure).

To speed up hardware prototyping in the context of JIT compilation, we present Gigue [1] (French for *jitter*), an open-source binary generator that outputs executable snapshots of the JIT code memory region. The region is split at the granularity of methods, whose content is randomized. It implements JIT custom instructions without the need to extend the whole technology stack to operate. We believe, as the JIT compiler is the sole runtime machine code generator, that a focus on its output and how it can be instrumented provides good insight into the interest of one solution over another. Gigue is heavily parametrized to represent different JIT code regions (and therefore different VMs) as well as different application types running on top. Gigue usage is three-fold: (1) generate bare-metal randomized snapshots of the JIT code region that qualify different VMs or applications; (2) verify the execution of generated binaries using a CPU emulator (Unicorn [15]) extended on the software side to support the new instructions; and (3) generate a set of instrumented binaries in ELF format, ready to be executed on a core.

In this paper, we propose the following contributions:

- Gigue, an open-source parametrizable and modular binary generator that outputs a JIT code region executable, filled with randomized methods, for various layouts and application classes,
- An extensible test framework to define the behavior of added instructions and ensure the soundness of their implementation on the software side, here as an extension of the Unicorn CPU emulator,
- An example use case on top of the Rocket open-source processor [2] with the baseline execution of different application classes.

This article is organized as follows. Section 2 presents the basis of the Pharo VM, the initial inspiration for Gigue. It also presents the RISC-V ISA and related works to narrow the range of the technology stack involved between hardware and VM development. Section 3 presents Gigue design principles both in terms of its binary and execution structure as well as the main generation algorithm. Section 4 highlights the important characteristics of Gigue: modularity to better represent VMs, parametrization to better represent applications, and a test framework to guarantee the correct execution of generated binaries. Section 5 shows a use-case to generate binaries for different application classes on top of the Rocket CPU. Finally, Section 6 and 7 a utility in the context of hardware development and lay down the future works.

[1] https://github.com/QDucasse/gigue

2 Background

2.1 Pharo VM

The Pharo language inherits from Smalltalk-80 and is a pure object-oriented language revolving around message sends as its main control-flow construct. Pharo, in the same vein as Python or JavaScript is dynamically-typed and supported by a language *Virtual Machine* (VM). The VM serves as the execution environment the language needs to be executed, it provides portability, memory management of the live program through snapshots, and IOs handling through foreign function interfaces.

To grant and retrieve memory, Pharo and other managed languages use a *garbage collector* that traces and records the usage of live objects and constructs. Once an object is not referenced anymore, it is collected and its memory is released. To compile and execute source code, the VM also contains a *parser*, a *bytecode compiler*, an *interpreter*, and a *JIT compiler*. When the source code is first met, the parser of the VM constructs the corresponding AST, which is compiled to bytecode by the bytecode compiler. This bytecode is then consumed by an interpreter, dispatching each bytecode element to its corresponding effect. The Pharo VM uses a baseline non-optimizing JIT compiler and defines *polymorphic inline caches (PICs) [8]* in its JIT code region. PICs are machine code stubs setting up a combination of type guards and jump tables to cache polymorphic JIT method addresses. Several other VMs hold various JIT compilers with increasing optimization complexity, triggered when the profiling ensures time spent compiling and optimizing is worth it.

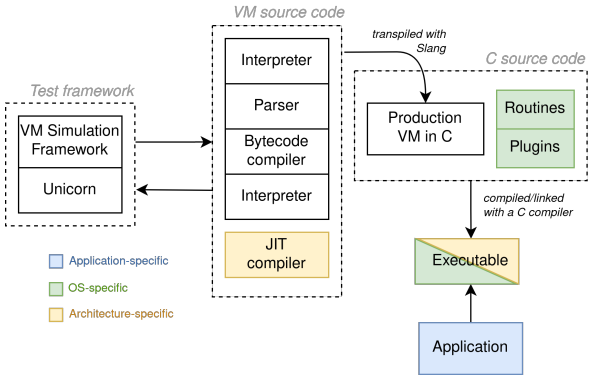

Figure 2. Pharo VM compilation process.

As shown in Figure 2, the Pharo VM is written in Slang [9], a restricted Smalltalk sublanguage that supports a reduced number of Smalltalk features. It is then transpiled to C and compiled for the target architecture along with architecture and OS-specific helpers. This way, the main components responsible for the execution of the program are easily portable as long as the C compiler supports the target architecture. This portability still comes at the cost

of performance since the JIT compiler has to be manually ported to new architectures [6, 13].

2.2 RISC-V

The RISC-V ISA is emerging as a serious competitor for adoption in wide areas from limited resources embedded devices to hyper-computing clusters. Its main benefits are its **modularity** and the simplicity inherited from RISC ISAs; the **open-source** characteristics of the specifications and several fully-featured cores; and its **extensibility** with custom instructions.

RISC-V defines a set of drafted and ratified extensions starting from the 32-bit integer (`RV32I`) and increasingly adding 64-bit support, multiplications and divisions, atomic operations, floating-point operations, and compressed instructions to define `RV64IMAFDC` (or `RV64GC`) [19], a set of instructions capable of supporting a fully-featured OS. In addition to instructions themselves, a privileged specification defines three privilege modes: machine, supervisor, and user mode [20]. The system can only be configured in machine mode, the OS runs on the supervisor mode and provides an interface through which user programs are executed in user mode. This combination of extensions allows a core to support a fully-featured operating system such as Linux. Additional extensions dedicated to specific use cases are ratified or still in development, waiting to be frozen in the specifications.

Now, in the context of VMs, the technology stack goes from high-level managed and dynamically-typed source code down to generated machine code. Since the JIT compiler generates machine code at run time, we believe it could use JIT-specific dedicated instructions either for common security measures (stack isolation [4], pointer integrity [11], or domain-based isolation [10]), acceleration measures (vector operations as defined in the `V` extension [19], neural-network operations [7]), or VM-specific measures (type checking, object offset verification). However, adding a new instruction to the JIT compiler requires support from both the OS and the underlying processor. This technology stack covering all levels of execution makes it complex to prototype, and test new instructions and their impact at the different levels of the stack. Alongside software complexity, the multiplicity of core implementations, either through their design or the extensions they support makes it very complex to maintain or test ideas.

2.3 Custom Instruction Examples

To provide a better understanding of what Gigue aims to support, we present examples of custom instructions of increasing complexity that could be added to a processor. The first one consists of a hardware-accelerated primitive while the two others define hardware security primitives:

- **E1:** Dedicated instructions to handle bits rotation (included in the RISC-V `B` extension not ratified yet).
- **E2:** A shadow-stack implementation with two instructions to push and pop a return address to a dedicated call stack, taken from the FIXER co-processor implementation [4].
- **E3:** An instruction-level domain isolation mechanism derived from RIMI [10]. It duplicates all memory access instructions, assigns them to separate memory regions, and guarantees only intra-domain instructions access to their corresponding data.

Implementation of **(E1)** adds four instructions, two rotations between registers (`R`-type `ror` and `rol`) and the corresponding rotation using an immediate value (`I`-type `rori`).

Implementation of **(E2)** adds two new instructions dedicated to the handling of return addresses during *calls* and *returns* that either push or pop it to the duplicated stack, using `spush` and `spop`. *Calls* should use the dedicated instructions for return address push, and *returns* should use the dedicated return address pop. The return is either located in the methods' epilogue or the dedicated trampoline, if in use.

Implementation of **(E3)** duplicates all loads and stores (`lb1`, `lh1`, `lw1`, ..., `sb1`, `sh1`, `sw1`, `sd1`), and adds two domain-changing instructions for calls and returns changing domains (`chdom` and `retdom`). To separate the interpretation loop in one domain and the JIT code in another, all generated loads and stores should be the ones duplicated, and the routine to change domains if needed should be added to the trampolines responsible for the control-flow transfer.

2.4 Existing Development Tools

On the software side, the Pharo VM relies on an in-depth simulation framework [12] to keep most of its development high-level and in its feature-rich environment. The JIT compiler is tested in addition to this framework by feeding recompiled machine code into the Unicorn CPU emulator [15]. This technique helped add robust unit tests working in a black-box fashion to ensure the soundness of the JIT compilation. Unicorn is a lightweight wrapper on top of QEMU [3] that provides numerous hooks to catch and handle CPU exceptions. This framework helped port Cogit, the Pharo JIT compiler to ARMv8 and RISC-V [6, 13]. It also helped prototype the use of custom instructions in the JIT compilation flow by: generating them, launching the execution on Unicorn, catching an "unknown instruction" exception, running its desired behavior on the software side and modifying the CPU state in Unicorn directly, then finally resuming execution.

On the hardware side, testing the correct implementation of the ISA in a CPU is two-fold: (1) ensuring the correct behavior of the ISA as defined in the standards, an assembly test suite (`riscv-tests` [16]) is defined to ensure the soundness

of implementation; (2) ensuring the correct implementation of the components of the CPU, the different stages of the pipeline, memory hierarchy and peripherals. Softcore implementations are written in Hardware Description Languages (HDL) that can be generated into bitstreams to be deployed on a reconfigurable architecture for prototyping. Tools help simulate the design through waveform generation, among them closed-source commercial simulators such as Synopsys, and open-source alternatives such as Verilator [18].

3 Gigue: JIT Code Benchmark Generator

To prototype new instruction ideas for the JIT compiler, we have to bridge the gap in the testing framework between the correct implementation in the JIT compilation pipeline, and guarantees enforced by the hardware itself. As stated in the introduction, the technology stack is complex, and portability is mostly guaranteed for all VM components (through major compilers) apart from the JIT compiler. Prototyping extensions on the hardware side could use a simplified version of the JIT code region and assess the impact on the core through two main metrics: (1) the performance overhead through the measured number of cycles, and (2) the area overhead of the CPU the solution adds. We believe having an insight into the performance overhead of a solution will guide our choice for implementation in the real JIT compiler. To ease prototyping, we designed Gigue, a tool that generates executable binaries similar to the JIT code region, filling its methods with random instructions to generate a parametrizable workload.

With Gigue, our objective is to ease the complexity of the technology stack involved in the support for custom instructions in the JIT compiler. It was designed with three main objectives in mind:

- *Parametrization:* Gigue is parametrizable to accurately qualify application classes. The parameters are covered in Section 3.2 and qualify both the size of JIT elements, the type of instructions they contain, interactions between elements and generated data.
- *Modularity:* Gigue is designed to be modular and output a binary that resembles the JIT code region of the target engine. It provides facilities to add user-defined structures that are found in the JIT code region (methods, PICs, hidden classes, *etc.*) and is presented in more detail in Section 4.1.
- *Testing:* Gigue defines a test framework to guarantee the correct setup of JIT elements and the interpretation loop. It also checks the correct decoding and execution of the generated binary as presented in Section 4.2.

It provides a parametrizable JIT code region executable inspired by the Pharo VM JIT code region layout but adaptable to other JIT code memory layouts. It generates an executable ELF file that defines both an interpretation loop and a JIT code region that represents a static version of the JIT code region. The JIT code elements (methods and optimization structures) are filled with random (and sanitized) instructions based on input parameters.

3.1 Binary Structure and Execution

The JIT code region contains different elements and machine code constructs:

Methods: A *method* is characterized by its *size*, *call number*, *call depth*, number of *local variables*, and number of used *callee-saved registers*. It is composed of a *prologue*, adding space on the stack and saving callee-saved registers; a *body*, filled with random instructions and calls to other methods or PICs; and an *epilogue*, restoring register values saved on the stack and destroying the call frame.

PICs: A *polymorphic inline cache (PIC)* is characterized by a *case number* and composed of a machine code switch case checking a corresponding class register for a value and jumping to the corresponding method offset. We use simplified class values (simple integers) and add the corresponding methods right after the "switch" statement. Calling a method in a PIC requires loading a corresponding value into the fictive class register before issuing a jump to the switch statement.

Trampolines: A *trampoline* is a helper added to the JIT code region used to control the interoperability between the interpreter and the JIT code region. Trampoline usage covers a wide variety of utilities, from accessing particular object field offsets to type-checking JIT methods and PICs, and switching execution stacks. The base generation Gigue provides simply defines control-flow trampolines to correctly transfer control flow back and forth between the two parts of the binary: `callJITelt` and `returnToInterpreter`.

Execution: Gigue generates an ELF binary that follows the execution design presented in Figure 3. The JIT code region contains methods and PICs filled with random instructions. The interpretation loop calls every one of the JIT elements in random order and each element can call other JIT elements. Starred arrows represent a call and return through the trampolines but were shortened for readability. An assembly template incorporates the interpretation loop, the JIT code region (at a fixed offset), and generated data whose base address is stored in a dedicated register (ensuring correct data accesses from stores and loads). The template is then linked according to the test framework of the `riscv-tests` official repository, defining the correct behavior of a core when confronted with different instructions and scenarios as defined in the standards. The binary is ready to be executed bare-metal on any RISC-V core that supports the test suite (and implements the custom instructions if included).

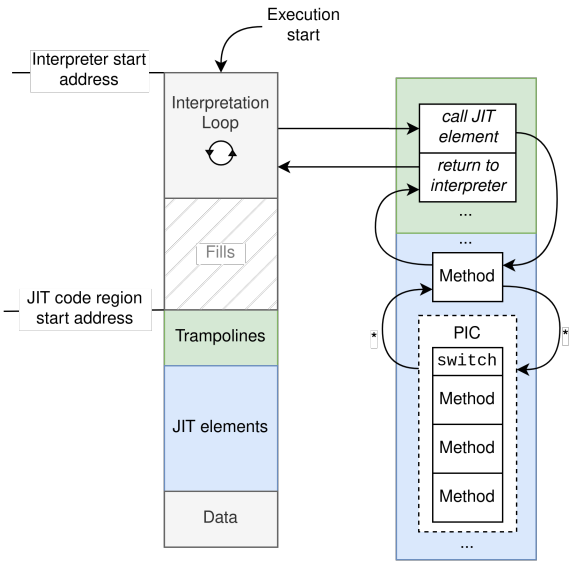

Figure 3. Binary structure and execution.

3.2 Parametrization

Gigue parameters are shown in Table 1 and split into two categories: *VM Characterization* tuning the JIT code region and *Application Characterization* tuning the contents of JIT elements. Their internal usage is described in more detail in Appendix A.

Table 1. List of Gigue input characterization parameters.

Type	Description	Name
VM	JIT code region size	$size_{JIT}$
VM	Frequency of JIT elements	$weights_{elts}$
VM	Usable registers	$regs$
App	Total number of methods	$nb_{methods}$
App	Method size variation	$\mu_{size}, \sigma_{size}$
App	Call occupation in methods	$\mu_{calls}, \sigma_{calls}$
App	Call intricacy and depth	λ_{depth}
App	PIC case number	λ_{PIC}
App	Frequency of instructions	$weights_{instrs}$
App	Data characterization	$size_{data}, generator$

VM Characterization: Parameters describe the JIT code region for a specific VM by specifying a fixed JIT code region size. It also specifies the types and associated weights of the different JIT elements (methods and optimized structures) along with available registers, *i.e.* registers used by the VM itself that will not be used by the application. We believe that these values help distinguish VM JIT code regions from one another.

Application Characterization: Parameters characterize an application through the contents of the JIT elements. The methods are characterized by a mean method size, derived from the method number. Method bodies are incrementally sized by applying the method size variation parameters (variance and standard deviation, more in the next sections). The methods are also characterized by a call number derived from call occupation parameters (variance and standard deviation, more in the next sections) and a call depth derived from a mean input value. PICs are given a case number derived from a mean input value. The application is also characterized by the weights of the different instruction types to represent more arithmetic operations, branches and jumps frequency, or memory access intensity. Finally, the associated JIT data is sized and generated along with the code to better represent types of applications. We show in Section 5 how different types of application classes can be defined using those parameters.

Algorithm 1 Binary generation.

procedure AddMethod
  $size_{method} \leftarrow Random(\mu_{size}, \sigma_{size})$ ▷ *see next section*
  $nb_{calls} \leftarrow Random(\mu_{calls}, \sigma_{calls})$ ▷ -
  $call_{depth} \leftarrow Random(\lambda_{depth})$ ▷ -
  Fill($weights_{instrs}$) ▷ -
  return **Method**

procedure AddPIC
  $nb_{cases} \leftarrow Random(\lambda_{cases})$ ▷ -
  $i \leftarrow 0$
  while $i < nb_{cases}$ **do**
    AddMethod
    $i \leftarrow i + 1$
  Add *switch stub*
  return **PIC**

procedure GenerateBinary
  Add *trampolines* ▷ Phase 1
  Add *leaf method*
  $method_{count} \leftarrow 1$
  while $method_{count} < nb_{methods}$ **do**
    element $\leftarrow Random(weights_{elts})$ ▷ *see next section*
    if element = method **then**
      AddMethod
      $method_{count} \leftarrow method_{count} + 1$
    else if element = pic **then**
      AddPIC
      $method_{count} \leftarrow method_{count} + nb_{cases}$
  for each method **do** ▷ Phase 2
    Determine *possible callees*
    Patch *calls*
  Generate *interpretation loop* ▷ Phase 3
  Generate *data* ▷ Phase 4
  Write *binary*

3.3 Instruction and Constructs Generation

Gigue is designed around a `Generator` object, responsible for the construction of the target binary following input parameters. It manages the JIT code region by filling it with the different constructs, starting with the chosen trampolines, then with methods and PICs characterizing them with input parameters and probability distributions.

The main generation routine operates in four main phases, as presented in Algorithm 1 and the `GenerateBinary` procedure. (1) JIT elements are added to the JIT binary through their weights (calling the corresponding helper defined in procedures `AddPIC` and `AddMethod`). The helpers to add the different elements to the JIT code region are given parameters derived from the input configuration and using random distributions and weights as defined in the following sections. (2) Calls between JIT elements are added after generation, and a method can only call another element with a lower call depth to ensure no infinite loops are generated. We also restrain methods from calling themselves due to the lack of termination conditions. These two rules define a low-cost guarantee of complete execution for calls. In addition, branches and jumps are sanitized to always target the content of the current method and avoid eventual calls within the body. (3) Calls from the interpretation loop to the JIT elements are generated. Finally, (4) corresponding data is generated according to the chosen strategy (examples are `random` or `incremental`). Data accesses are performed using a base register set in the linker script and an offset randomized within the maximum data size. Additional details on the different randomization techniques used in Gigue generations are provided in Appendix A.

4 Modularity and Guarantees

As presented at the beginning of the previous section, Gigue was designed to generate low-level hardware-testable binaries to test new instructions or hardware-enforced concepts in the application context of JIT compilation and the JIT code region. In addition to *parametrization* as presented earlier, we wanted to provide *modularity*, and a strong *testing framework*.

4.1 Modularity and Extensions

For VM characterization, each JIT element (trampolines, methods, and optimizations) answers a simple API to be added to the binary through the `Generator`. New constructs are added through a subclass of a JIT element along with its helper and probability weight. The only element that handles machine code instructions is the `Builder`, responsible for both outputting lone instructions and constructs such as calls or prologues/epilogues. It also enforces correct alignment for data access and sanitizes the landing of branch/jump instructions. Mechanisms using new instructions are integrated through a new subclass of the main builder, redefining the API of the main instructions and constructs it wishes to add or overload.

The three extension examples presented in Section 2.3, **E1** (rotations), **E2** (shadow stack), and **E3** (instruction-level domain isolation) were implemented in Gigue. Overall, the complete handling of the three different extensions requires the addition of the instructions and the corresponding builder for changed constructs. The `Builder` element is subclassed to: add the new instructions to their corresponding types for **E1**, define different prologues/epilogues for **E2**/**E3**, and extend control-flow trampolines for **E3**. Gigue is composed of 3193 lines of object-oriented Python code, the addition of **E1** took 161 lines, **E2** 154 lines, and **E3** 502 lines of code (counted using `cloc`, code formatted using `black`) respectively. These additions cover the instruction details and `Builder/Generator` extension. In addition to the code portion of the generator, and as presented in the next section, the test framework is also extended with a total of: **E1** 33, **E2** 31, and **E3** 236 to support the software execution of custom instructions.

4.2 Test Framework

Gigue uses a test suite of around 2500 parametrized unit tests to assert guarantees on the generated binaries. Part of those tests directly executes the generated machine code. To support the decoding and execution of the machine code, we use the Capstone disassembler [14], an in-house disassembler, and the Unicorn CPU emulator [15]. It is a lightweight wrapper on top of QEMU that allows for simple binary instrumentation and tracing.

Using Unicorn, we guarantee the correct execution of the binary before porting on core implementations. It allows for quick tracing and verification of the executed instructions and defuses early any issues that might be encountered when dealing with the real hardware. In addition to simple baseline testing of unmodified binaries, we also adapted the Unicorn hooks to catch any unknown instruction. This way, even binaries implementing custom instructions and extensions can be executed completely, with the custom instructions emulated in software directly before redirecting control flow to the binary. It defines the additional structures and execution model that should be followed by the actual core implementing the extension and guarantees the correct usage of new instructions.

5 Example Application Case and Setup

As an example, we chose the Rocket chip [2] to execute Gigue-generated binaries. It is written in Chisel [1], a Hardware Description Language at Register Transfer Level (RTL) that extends Scala. The Chisel code is then translated to Verilog which can be run through a synthesis tool to deploy on hardware. Rocket also provides a cycle-accurate C++ simulator generated through Verilator [18]. The execution framework is presented in Figure 4: the input parameters

are fed to Gigue, and binaries are generated according to them and then executed on top of the Rocket emulator. As presented here, even when reducing the technology stack involved in JIT-dedicated hardware instructions, it still is complex to manage.

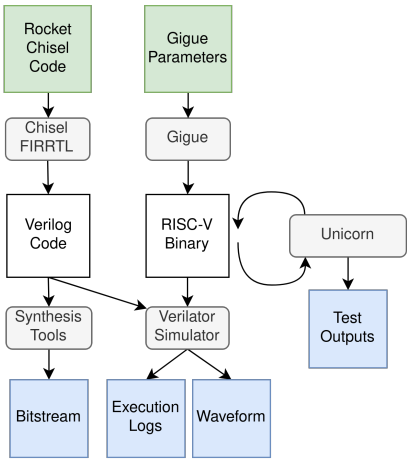

Figure 4. Execution setup and test framework.

To demonstrate the Gigue generation capabilities, we chose to change two parameters independently to define several application classes: *call occupation* and *memory access intensity*. We fix the binary size as the maximum size of the JIT code region is fixed and allocated at startup. We vary the number of methods and then independently change *call occupation* parameters (variance and standard deviation, along with the call depth variance) and *memory access* parameters (weights on `stores` and `loads`). Binaries are generated from the base RISC-V ISA and run on an unmodified version of the core

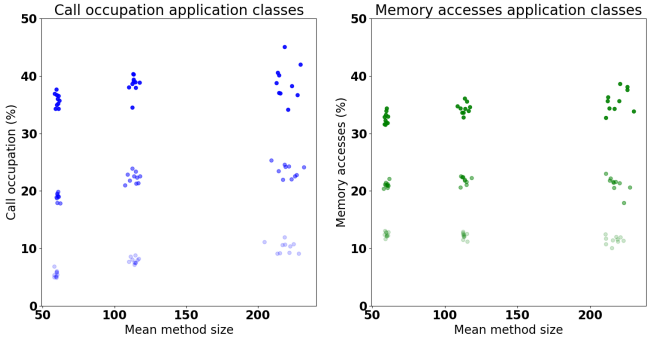

Figure 5. Different application classes generated with varying call occupations/memory access intensities to methods number.

Binaries are generated using a fixed size of 7000 instructions (or 28kB). The choice of a fixed value for the binary size is motivated by the fact that this value is sized and fixed on a

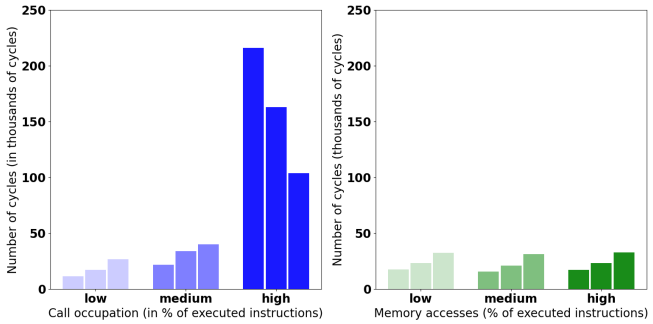

Figure 6. Mean number of cycles for each application class. Each group of three bars uses 50, 100, and 200 methods respectively.

per-VM basis. We run each of the 18 application classes (9 for call occupation, 9 for memory access intensity) 10 times. To generate them, we used a varying method number common to both *calls* and *memory accesses*: 50, 100, and 200 methods. This number of methods is inversely proportional to the mean method size as the total binary size is fixed.

Call occupations using the call parameters Gigue provides, $(\mu_{calls}, \sigma_{calls}, \lambda_{depth})$: *low* using $(0.1, 0.1, 1)$, *medium* using $(0.3, 0.1, 2)$, and *high* using $(0.5, 0.2, 3)$ (other parameters are shown in Appendix B). They all use fixed instruction weights corresponding to the *medium* version of memory accesses. On the left side of Figure 5 are displayed the 9 corresponding application classes, each containing 10 samples. Figure 6 presents the corresponding mean number of cycles for each application class.

Memory access intensities vary using weights of `stores` and `loads`: *low* using 2% for each, *medium* using 10%, and *high* using 20%. They all use fixed call parameters corresponding to the *medium* version of call occupation. On the right side of Figure 5 are displayed the 9 corresponding application classes, each containing 10 samples. Figure 6 presents the corresponding mean number of cycles for each application class.

Note that while those are the input Gigue parameters, the displayed values are the effective ones measured by parsing the Rocket execution logs. The amount of memory accesses is higher due to prologues/epilogues accessing the stack. Using Gigue, we managed to generate different classes of binaries and directly execute them bare-metal on top of the Rocket CPU.

6 Discussion

Gigue allows for quick prototyping of custom instructions in the context of JIT compilation. It generates a set of application classes through random binaries. Those binaries qualify both a JIT code region and an application, and act as a snapshot of the JIT code region at a given moment.

The existence of Gigue is motivated by the need for a hardware testing utility that does not imply handling the complete technology stack. Hardware development and extension involve a time-consuming process and framework. We believe that while major OS and compiler support will provide soundness guarantees over the other VM components, the JIT compiler and its corresponding memory region benefit from being tested separately. Our objective is to design JIT-specific instructions to accelerate and secure the generated machine code.

Gigue was designed to assess the impact on execution overhead when introducing new JIT-specific mechanisms. We believe it provides important insight and first-hand impact qualification of new constructs and instructions. However, it cannot serve as a complete evaluation of new instructions as it proposes a simplified version of the JIT code region and does take into account the runtime reconfiguration of this region. Additionally, while the generated binaries correspond to input parameters, they cannot describe a fully-featured, high-level application. Further testing and benchmarking should be performed through the extended JIT compiler to assess the impact of such features.

7 Conclusion

In this paper, we presented the process of JIT compilation and how it could benefit from custom instructions. As the RISC-V ISA is gaining traction, several fully-featured processors are available open-source and enable application-specific prototyping and benchmarking. However, as the testing framework is already consequent for hardware development, adding the complete VM through the OS would slow down an already tedious process.

We presented Gigue, an open-source binary generator that is modular, parametrizable, and provides a test framework to model and guarantee the correct execution of binaries implementing custom instructions. Gigue defines the binary structure and execution to better represent different VMs and their associated JIT code region. It also provides parameters to qualify applications running on top of the VM with, for example, varying call occupations or memory access intensities. We presented how Gigue generates such applications and how the generated binaries can be integrated into a larger tooling suite designed around a fully-featured RISC-V core, Rocket. Gigue helps design the implementation of new instructions, testing them in software before generating the corresponding binaries to execute on top of a custom core. We believe it speeds up the hardware testing by providing meaningful insight into the new instructions and implementation guidelines through its testing framework.

However, Gigue is by no means a drop-in replacement for an impact assessment on the full technology stack from the core itself to fully-featured applications on top of the OS running through the VM. Future works involve testing security and acceleration solutions on the Rocket processor before implementing the solutions in the Pharo VM. We also plan on extending Gigue with default parameters extracted from the Pharo VM itself and new elements to better represent other VMs.

A Randomness Characterization

The Gigue-generated binary is heavily randomized. It uses *sample weighting* to select random instructions, JIT elements, and register arguments as presented in this subsection. It characterizes JIT element attributes using *distribution laws* presented in the next subsection.

Weighted sampling.

JIT element type: Each JIT element type is given a corresponding weight that the generator uses to choose the next element to add to the JIT code.
Instruction type: Available instructions are split by type (`R`-egister, `I`-mmediate, `U`-pper immediate, `J`-umps, `B`-ranches, `S`-tores, and `L`-oads) with their corresponding weight that is selected by the builder when filling the body of a method. Note that while load instructions officially are `I`-type instructions, they were isolated to have a better separation between arithmetic instructions (`R` and `I` instructions) and memory access instructions (`L` and `S` instructions). The builder returns an instruction from a group of instructions at random using equal weights for all of them.
Register pressure: All registers are defined with their corresponding weights, translating into their pressure. Temporary registers (`t0-t6`) or the hardwired zero (`x0`) are expected to be more frequently used than others.

Distribution laws.

Method body size: Mean method size is extracted from the input fixed total binary size and the input number of JIT methods. To determine the body size of a method, the mean size is mitigated by a size variation as follows:

$$size_{method} = \lceil \frac{size_{bin}}{nb_{methods}} * (1 + s * v) \rceil \quad (1)$$

where:

- $s = 2X - 1$ with $X \sim \text{Bernoulli}(0.5)$
- $v \sim \text{TruncNorm}(\mu_{size}, \sigma_{size}, 0, 1)$

The method body size is derived from a mean method size and then either shrunk or amplified (s is 1 or -1 with equal probabilities) based on the size variation parameters. The variation is expressed as a percentage of the overall method size. The truncated normal distribution is used to guarantee a result between 0 and 1.
Call number: Input variance and standard deviation for *call occupation* (percentage of instructions responsible for

calls in a method body) help determine the call number each method will employ as follows:

$$nb_{calls} = \lfloor \frac{size_{body}}{size_{call}} * v \rfloor \quad (2)$$

where:

- $v \sim \text{TruncNorm}(\mu_{calls}, \sigma_{calls}, 0, 1)$

The number of calls is extracted from the maximum number of calls a method can hold and uses *call occupation* parameters to add variation. The truncated normal distribution is used to guarantee a result between 0 and 1.

Call depth: Binary execution is determined to complete by fixing the call depth (or amount of nested calls leading to calling this method) of each method and forcing, in turn, this method to only call other elements with call depth inferior by one level. It is determined as follows:

$$depth \sim \text{Poisson}(\lambda_{depth}) \quad (3)$$

The Poisson distribution with $\lambda_{depth} \leq 4$ provides a good amount of leaf methods in the JIT code region and guarantees method interactions. The *call depth* parameters are used when patching calls to force methods of *call depth* level i to only call methods of *call depth* level $i-1$. This simplifies the call graph generation and restrains recursive functions and infinite loops from appearing.

PIC case number: Inline caches start with only one entry (*monomorphic*), growing up to multiple (*polymorphic*), and eventually ending with many (*megamorphic*) if needed. It is determined as follows:

$$nb_{cases} \sim \text{ZeroTruncPoisson}(\lambda_{cases}) \quad (4)$$

The zero-truncated Poisson distribution guarantees that no empty PIC is generated and ensures the mean number of cases is respected.

B Gigue Input Parameters

The following parameters are the default ones used by Gigue. The `elts_weights` parameter defines the representation of methods (80%) against PICs (20%). The `isolation_solution` parameter is used to request a binary implementing a shadow stack according to the FIXER paper [4], a shadow stack according to the RIMI paper [10], or a complete domain isolation with RIMI. Three additional registers can be defined: PIC-related registers corresponding to a class register and a current class comparison register (to ease the writing of PIC switches), as well as a data register pointing to the data section to ease memory access. The interpreter and JIT start addresses are defined according to the `riscv_tests` framework linker script whose execution model is supported by major open-source cores.

```
"nb_runs": 10,
"run_seeds": [],
"input_data": {
    "isolation_solution": "none",
    "registers": [
        5, 6, 7, 10, 11, 12, 13, 14,
        15, 16, 17, 28, 29, 30, 31
    ],
    "elts_weights": [80, 20],
    "instr_weights": [25, 30, 10, 5, 10, 10, 10],
    "interpreter_start_address": 0,
    "jit_start_address": 12288,
    "jit_size": 10000,
    "jit_nb_methods": 100,
    "method_variation_mean": 0.2,
    "method_variation_stdev": 0.1,
    "call_depth_mean": 2,
    "call_occupation_mean": 0.2,
    "call_occupation_stdev": 0.1,
    "pics_mean_case_nb": 2,
    "data_size": 1600,
    "data_generation_strategy": "random",
}
```

References

[1] CHIPS Alliance. 2023. *Chisel/FIRRTL Hardware Compiler Framework.* https://www.chisel-lang.org/

[2] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. *The Rocket chip generator.* Technical Report.

[3] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Conference (ATEC'05).* USENIX, 41–46. https://www.usenix.org/legacy/events/usenix05/tech/freenix/bellard.html

[4] Asmit De, Aditya Basu, Swaroop Ghosh, and Trent Jaeger. 2019. FIXER: Flow integrity extensions for embedded RISC-V. In *Proceedings of the 26th Design, Automation & Test in Europe Conference & Exhibition (DATE'19).* IEEE, 348–353. https://doi.org/10.23919/date.2019.8714980

[5] Quentin Ducasse. 2023. *Gigue: Benchmark Setup and Code Generator for JIT code on RISC-V.* https://github.com/qducasse/gigue

[6] Quentin Ducasse, Guillermo Polito, Pablo Tesone, Pascal Cotret, and Loïc Lagadec. 2022. Porting a JIT Compiler to RISC-V: Challenges and Opportunities. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes (MPLR'22).* ACM, 112–118. https://doi.org/10.1145/3546918.3546924

[7] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca Benini. 2020. PULP-NN: accelerating quantized neural networks on parallel ultra-low-power RISC-V processors. *Philosophical Transactions of the Royal Society A (RSTA)* 378 (2020). https://doi.org/10.1098/rsta.2019.0155

[8] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the 6th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (ECOOP'91).* Springer, 21–38. https://doi.org/10.1007/bfb0057013

[9] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97).* ACM, 318–326. https://doi.org/10.1145/263698.263754

[10] Haeyoung Kim, Jinjae Lee, Derry Pratama, Asep Muhamad Awaludin, Howon Kim, and Donghyun Kwon. 2020. RIMI: instruction-level

memory isolation for embedded systems on RISC-V. In *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD'20)*. ACM, 1–9. https://doi.org/10.1145/3400302.3415727

[11] Arjun Menon, Subadra Murugan, Chester Rebeiro, Neel Gala, and Kamakoti Veezhinathan. 2017. Shakti-T: a RISC-V processor with light weight security extensions. In *Proceedings of the 6th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP'17)*. 1–8. https://doi.org/10.1145/3092627.3092629

[12] Eliot Miranda. 2011. The Cog Smalltalk virtual machine. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL'11)*. ACM.

[13] Guillermo Polito, Pablo Tesone, Stéphane Ducasse, Luc Fabresse, Théo Rogliano, Pierre Misse-Chanabier, and Carolina Hernandez Phillips. 2021. 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 (MPLR'21)*. ACM, 16–25. https://doi.org/10.1145/3475738.3480715

[14] Nguyen Anh Quynh. 2014. Capstone: Next-gen disassembly framework. (2014). https://www.capstone-engine.org/BHUSA2014-capstone.pdf Black Hat USA.

[15] Nguyen Anh Quynh and Dang Hoang Vu. 2015. Unicorn: Next generation CPU emulator framework. (2015). https://www.unicorn-engine.org/BHUSA2015-unicorn.pdf Black Hat USA.

[16] riscv-software src. 2023. *riscv-tests*. https://github.com/riscv-software-src/riscv-tests

[17] Moritz Schneider, Ramya Jayaram Masti, Shweta Shinde, Srdjan Capkun, and Ronald Perez. 2022. SoK: hardware-supported trusted execution environments. *Computing Research Repository (CoRR)* (2022). https://doi.org/10.48550/arXiv.2205.12742

[18] Veripool. 2023. *Verilator*. https://www.veripool.org/verilator/

[19] Andrew Waterman, Krste Asanovic, and SiFive Inc. 2019. *The RISC-V instruction set manual, Volume I: unprivileged ISA*. https://github.com/riscv/riscv-isa-manual/releases/tag/Ratified-IMAFDQC

[20] Andrew Waterman, Krste Asanovic, and SiFive Inc. 2021. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. https://github.com/riscv/riscv-isa-manual/releases/tag/Priv-v1.12