Formal Verification of JIT by Symbolic Execution

Boris Shingarov
LabWare
Ottawa, Canada

Abstract

This work-in-progress report presents ongoing experiments relating to formal verification of JIT compilers for language VMs. The native CPU code of the VM — which consists of statically-known code and variable output of the JIT — is executed in a symbolic simulation engine. This simulation yields identities that hold over the total range of inputs (or disproves them by providing a counterexample).

One obstacle we had to overcome, is executing CPU code which is itself symbolic, i.e. given as formulae over input variables. To solve this problem, we designed a new ISA-agnostic translator from ISA-specific binary machine language into an intermediate language which can be directly simulated by the symbolic engine.

CCS Concepts • **Software and its engineering → Virtual machines**; **Formal software verification**;

Keywords Virtual Machine, Formal Compiler Verification, Symbolic Execution

ACM Reference Format:
Boris Shingarov. 2019. Formal Verification of JIT by Symbolic Execution. In *Proceedings of ACM SIGPLAN International Workshop on Virtual Machines andIntermediate Languages (VMIL'19).* ACM, New York, NY, USA, 5 pages.

1 Introduction

Traditionally, software developers validate their software to ensure absence of programming errors using the approach called *testing*. In a test, we execute the software program giving it a *test vector* as input, and observe its behavior. If the program behaves correctly on each test vector we test it with, we deem that it *appears to work*. Unfortunately, testing says nothing about those inputs that are not in the set of the tested vectors, leaving us with the problem of *latent bugs*.

Latent bugs in *compilers* present an even more difficult problem. Due to the level of abstraction compilers operate at, these bugs often hide for longer, are more subtle, may manifest in a crash only billions of CPU instructions after the bug's true cause, and are generally more difficult to isolate than bugs in other software.

Approaches to solve this problem by formalizing the compiler in a logic suitable for mechanical proof, have been tried for several decades. More recently, large-scale, realistic compilers have benefited from formal proof — for example, Leroy [8] used the Coq proof assistant [4] to prove CompCert, a compiler from a subset of C; Lopez *et al.* [10] use the Z3 SMT solver [6] to prove correctness of LLVM optimizations.

Dynamic programming languages, such as Smalltalk, Python, Java *etc.*, rely on a Virtual Machine for runtime execution. Formalization of VMs has been an active area of research. Barthe *et al.* [3] use Coq to construct an executable model of JavaCard. M6 [9] (implemented in ACL2) and CoqJVM [1] (implemented in Coq) are two formalizations of the JVM. Despite being executable and rather complete, they essentially model the JVM specification. Real VMs rely on architecture-scale optimizations — such as Polymorphic Inline Caches and Just-in-Time compilation — to achieve practical performance. In this paper, we describe an experiment aimed towards automated reasoning about correctness of these optimizations, using the same mathematical apparatus that Lopes *et al.*'s Alive [10] uses for proving LLVM optimizations. We take a tiny bytecoded method parametrized by unknown input, JIT-compile it and execute the native CPU code — all in a symbolic execution environment. We compare the algebraically-variable output from the native code, to the expected output, and prove that they are equal for all inputs.

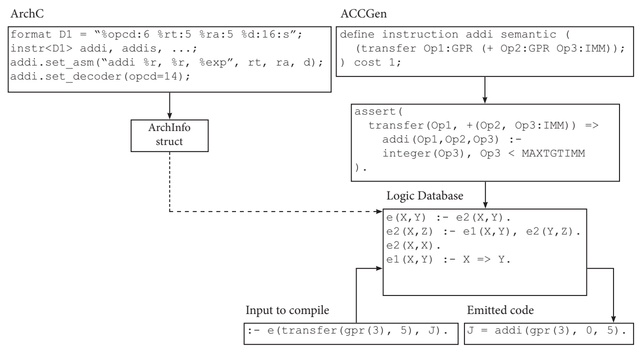
The remainder of the paper is organized as follows. In Section 2, we cite our sources which served as the starting point of our present work. Section 3 explains the general structure of the system, and delineates the contributions of this paper. In section 4, we illustrate how our method works, by walking through a simplest-possible example of translating a single IL instruction. In conclusion, in Section 5 we delineate the results we have obtained using the current proof-of-concept, from future results we aim at achieving with a full implementation.

2 Related work

The system we describe in this paper, is a major update to our previous system, Target-Agnostic JIT [14–16, 18, 19], illustrated in Figure 1. TA-J automatically infers the code generator (thus aiming at being correct by construction) from a formal specification of the ISA given in two Processor Description Languages: ArchC [13], a SystemC-based DSL for architectural modeling; and ACCGen [2], an extension DSL complementing ArchC by providing a specification for instruction semantics and platform ABI in a declarative manner.

TA-J synthesizes a superoptimizer-like compiler backend by treating ACCGen's instruction semantics statements as

VMIL'19, October 22, 2019, Athens, Greece
2019.



```
format D1 = "%opcd:6 %rt:5 %ra:5 %d:16:s";
instr<D1> addi, addis, ...;
addi.set_asm("addi %r, %r, %exp", rt, ra, d);
addi.set_decoder(opcd=14);
```

```
define instruction addi semantic (
  (transfer Op1:GPR (+ Op2:GPR Op3:IMM));
) cost 1;
```

```
assert(
  transfer(Op1, +(Op2, Op3:IMM)) =>
    addi(Op1,Op2,Op3) :-
    integer(Op3), Op3 < MAXTGTIMM
).
```

```
e(X,Y) :- e2(X,Y).
e2(X,Z) :- e1(X,Y), e2(Y,Z).
e2(X,X).
e1(X,Y) :- X => Y.
```

```
:- e(transfer(gpr(3), 5), J).
```

```
J = addi(gpr(3), 0, 5).
```

Figure 1. Target-agnostic JIT synthesis from PDL

symbolic equations in an uninterpreted first-order logic. Said statements are asserted into a logic database in a form of Cheng–van Emden–Parker Regular Equational Theory [5], and executed as a PROLOG program using the RET rewriting algorithm. TA-J's use of uninterpreted terms is at the same time its strength and its weakness, because it is not clear how to reason about combining effects of instructions based on structures other than pure composition of terms. In practice this leads to emitting correct but prohibitively inefficient code.

Our Verified Linear Smalltalk [17–19] attempts formal proof of one phase (final machine instruction selection) of a JIT backend using dependent types. The lessons from VLS — principally, that we have built a working proof-of-concept but implementing a production backend requires engineering effort beyond affordable in practical application — has motivated our further search.

Our present system reuses several parts of Shoshitaishvili *et al.*'s angr library [21]. Initially angr was conceived by the software security community. It integrates and systematizes many modern binary analysis techniques. The aspect of angr of most interest to us, is code analysis via algebraic execution of native CPU code by a simulation engine based on the SMT solver Z3 [6]. The idea is briefly summarized in Figure 2 [1]. Let's start from a basic block[2]. Angr uses Valgrind's "libVEX" — a collection of CPU-specific translators from binary machine code into "VEX IL", an intermediate language specifically designed for program analysis. This IL is then interpreted by a "VEX Simulation Engine", resulting in a number of "paths" and "states" and constraints between them. These constraints are used to gain *semantic insight* into the behavior of the machine code; one might, for example, ask questions such as "what input must have been presented to the program for the execution to branch into such-and-such path?" We will mostly be interested in the question: "prove that some P always holds" — or equivalently, that some assertion, usually expressed like

```
if (x < 0) {
  /* can never happen */
  die("Bug!");
}
```

can never take the "then" branch and reach "die()". To achieve this, we ask the solver to satisfy the constraints on the "die()" path, and hope for the "UNSAT" answer.

3 Overall structure and Contributions

Our system is a combination and extension of Target-Agnostic JIT and angr. The contributions of the present work are:

- a method for producing proofs of properties of self-modifying code by symbolic execution of algebraically-variable binary program text;
- a translator from algebraically-variable binary program text into a VEX-like IL suitable for symbolic execution;
- a method for automated synthesis of said translator from a processor description DSL designed for such synthesis.

The overall structure of our system is illustrated in Figure 3. We start from some IL input to the JIT compiler; this input may contain some "unknowns" (algebraic variables). We run the JIT inside the SimEngine[3]. The output is some bytes (the emitted binary) in the JIT's code cache. Some of those bytes will be variable (their value will be formulae over the input IL). Now we want to jump to the emitted code and run it. The problem is, libVEX will not do because

- it takes non-symbolic contents of instruction memory, and therefore cannot translate the JIT's symbolic output;
- it is unavailable for some ISAs of critical interest (e.g. for our RISC-V port [20])[4].

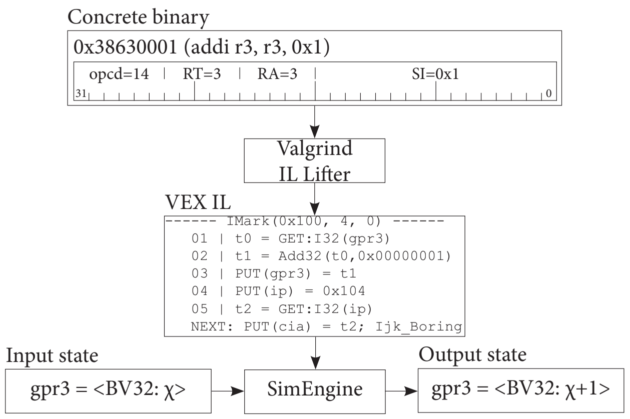
For these reasons, we replaced the VEX lifter with our own "LWISEM" lifter. The lifter is synthesized by parsing the PDL (there is no CPU-specific code). We kept the ArchC-proper part of the PDL syntax; its parsing results in an object serving the role of ArchInfo in the angr framework. We gave up the ACCGen part of the PDL and replaced it with our LWISEM DSL.

[1]Because our main focus is on proving JITs, we drew Figure 2 to facilitate the understanding of the difference between our system and angr by comparing with Figure 3, and not necessarily to be the most complete representation of angr *per se.*

[2]Such a block may (or may not) originate, for example, from loading an executable by angr's CLE Loader.

[3]It must be noted that this translator-execution stage is of illustrative and experimental nature, as opposed to being an essential characteristic of the method, as it does not guarantee the correctness of the JIT compiler's source: the compiler binary is what GCC happened to have produced. This can be seen as good or bad, but in any case this matter does not concern us now.

[4]Ideally we would like the lifter to work for any arbitrary ISA in a target-agnostic manner. As it is, libVEX contains e.g. 45342 lines of PowerPC-specific C code. Even if libVEX were suitable in principle, implementing that amount of RISC-V-specific code would be a prohibitively large side project.



```
------ IMark(0x100, 4, 0) ------
   01 | t0 = GET:I32(gpr3)
   02 | t1 = Add32(t0,0x00000001)
   03 | PUT(gpr3) = t1
   04 | PUT(ip) = 0x104
   05 | t2 = GET:I32(ip)
   NEXT: PUT(cia) = t2; Ijk_Boring
```

Figure 2. One scenario of symbolic execution. A basic block containing one concrete (non-symbolic) PowerPC instruction "addi r3, r3, 0x1" is lifted to VEX IL and executed by angr SimEngine.

LWISEM uses an intermediate instruction set similar to VEX in all respects except that it allows symbolic machine code. We had to extend the interface between this instruction set and the SimEngine, but it did not result in dramatic modifications to the SimEngine itself.

With this, symbolic-execution analysis of emitted code can be done as if it were a regular static executable.

4 Proof-of-concept experiments

We experimented with the native code generators of Modtalk [7], Bee Smalltalk [12], OpenSmalltalk [11], and OpenJ9/OMR TestaRossa [22] running on various ISAs. For illustration, in the following section we consider TestaRossa's "`iconst32`" IL instruction, and its compilation on the (32-bit) PowerPC ISA, because this is the easiest case to understand.

4.1 One instruction

Let's consider one IL instruction, `iconst32`. To verify the correctness of its JIT translation, we begin by executing the JIT compiler symbolically. The TestaRossa IL assembly which we start from, is

```
(method return="Int32"
   (block
      (ireturn (iconst v))
   ) )
```

The crux of our approach is to pass an algebraic variable χ of type "Bit Vector of length 32" as the value of actual parameter v:

v = <BV32 χ>.

We concentrate on verifying the "Instruction Selection" phase of the JIT compilation. The compiler dispatches on the `iconst32` TRIL instruction into `loadConstant()`:

```
loadConstant(..., int32_t v, Register *trgReg,...)
  {
  int32_t hi = getHighBits(v);
  int32_t lo = getLowBits(v);
  generateTrg1ImmInstruction(OpCode::lis, trgReg, hi);
  generateTrg1Src1ImmInstruction(OpCode::ori, trgReg, trgReg, lo);
  }
```

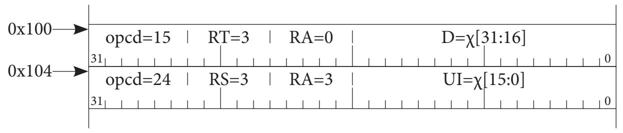
(These `generate*Instruction()` functions perform PowerPC binary encoding in a rather trivial way, by OR'ing the opcode which is the numeric value of an enum, with the operands — for example, in the case of `lis`, the enum value 0x3c000000 is OR'd with RT and D).

Simplifying somewhat for the sake of illustration[5], let's say the JIT code-cache begins from 0x100, and let's not worry about the prologue/epilogue code. Essentially, the JIT emits the

```
addis r3, r0, <upper half of v>
ori r3, r3, <lower half of v>
```

pair of instructions at 0x100. Because the OMR TestaRossa compiler is just a C++ program statically compiled by GCC, we can execute this translation within the `angr` simulation environment — we can give the v argument the value χ.

At the end of the simulated translation, the memory is in the following state — notice how the four bytes encoding D and UI are formulae over χ:



Next, we perform a jump to 0x100. In our experiment, this jump is conceptual. The important thing about it is that it requires a retranslation of PowerPC code at 0x100 into VEX IL. We cannot use VEX for this, because that memory contains algebraically-variable bits. This is the task for our LWISEM lifter.

The LWISEM lifter processes the CPU instructions in sequence. Picking up the first instruction, it passes it to the DISASM procedure. For our first instruction at 0x100 (beginning of the code cache), `addis`, DISASM's argument *arg* is the bit vector

<BV32 0x3c60#16 .. χ[31:16]>

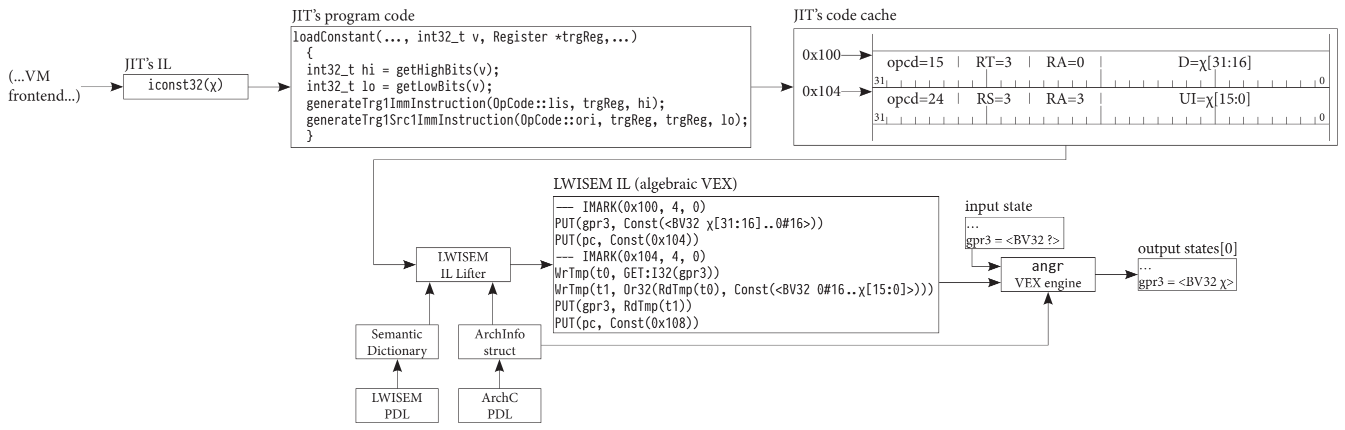
DISASM works by adding the predicate

$P = (instr.\text{binaryEncoding} == arg)$

as a constraint to the Z3 solver. Here P is a boolean AST: a two-operand "equals" node with two children both of which are bitvector ASTs. Solving this constraint immediately yields

instr = `addis`
ra = 0
rt = 3
d = <BV16: χ[31:16]>

[5] `loadConstant()` in the TestaRossa compiler on PowerPC includes code to differentiate the case when the 32-bit immediate constant fits in the SI field of li. It also branches on a "isPicSite" flag which indicates that the code might need to be patched with a longer constant in the future. We defer the consideration of such branching until Section 4.2.



```
loadConstant(..., int32_t v, Register *trgReg,...)
  {
  int32_t hi = getHighBits(v);
  int32_t lo = getLowBits(v);
  generateTrg1ImmInstruction(OpCode::lis, trgReg, hi);
  generateTrg1Src1ImmInstruction(OpCode::ori, trgReg, trgReg, lo);
  }
```

```
--- IMARK(0x100, 4, 0)
PUT(gpr3, Const(<BV32 χ[31:16]..0#16>))
PUT(pc, Const(0x104))
--- IMARK(0x104, 4, 0)
WrTmp(t0, GET:I32(gpr3))
WrTmp(t1, Or32(RdTmp(t0), Const(<BV32 0#16..χ[15:0]>)))
PUT(gpr3, RdTmp(t1))
PUT(pc, Const(0x108))
```

Figure 3. Symbolic execution of one JIT-translated VM bytecode

Indeed, Z3 can easily deem all other instructions to result in P being `false` for all valuations of χ and of all instruction operands. For example, adde.binaryEncoding $=$ *arg* gives

P = (<BV32: 31#6..*rt*#5..*ra*#5..*rb*#5..0x114#11> $=$ <BV32: 0x3c60#16..χ[31:16]>)

which is UNSAT; or in other words,

$\forall ra, rb, rt, \chi . \neg P$

holds.

Once the lifter knows it is dealing with the `addis` instruction, it looks up "`addis`" in the Semantics Dictionary. This dictionary is constructed by parsing the LWISEM DSL. The dictionary's keys are instructions, and the entries are simply templates of VEX IL. During a lift, VEX IL objects are instantiated by calling usual constructors from `pyVEX` library. In our `addis` example, the lifted IL is

```
IMARK(0x100, 4, 0)
PUT(gpr3, Constant(<BV32 χ[31:16] .. 0#16>))

PUT(pc, Constant(0x104))
```

In a similar manner, the `ori` is lifted to

```
IMARK(0x104, 4, 0)
WrTmp(t0, GET:I32(gpr3))
WrTmp(t1, Or32(RdTmp(t0), Constant(<BV32 0#16..χ[15:0]>)))
PUT(gpr3, RdTmp(t1))

PUT(pc, Constant(0x108))
```

Note how the `Constant` expressions are parametrized by AST trees containing variable χ.

Once the LWISEM IL is lifted, simulation begins from 0x100. We go until the end of the basic block and examine the value in GPR3 at the final state. Due to the simplifications performed by the solver on the algebraic ASTs, GPR3's value is now χ. This is the evidence we are looking for, that the code emitted by the JIT, behaves correctly. More formally, to compute the proof we ask Z3 to satisfy

$\neg$(finalState.regs.r3 $= \chi$)

Here we are asking to try to find a counterexample which would invalidate the correctness of our JIT translation. This results in UNSAT, which is the proof we are looking for (and Z3 can give us a Gentzen-style proof tree). If there were a bug — meaning that there is a path through the code for *some* input resulting in $r3 \neq \chi$ — the solver would give us that counterexample, i.e. show us how to trigger the bug.

4.2 Branching

The example in Section 4.1 is trivial in the sense that there are no branches — both `loadConstant()` and the generated code, each consist of one basic block, and we connected them by an unconditional jump, so this case doesn't have any interesting control structures. We can go beyond this linear case even when staying within our example of single `iconst32` IL instruction. On RISC-V, to load a 32-bit integer immediate into a register, one splits it into a 24-bit "*hi*" and a 12-bit "*lo*"; however, it is not enough to simply concatenate $\chi[31 : 12]$ with $\chi[11 : 0]$. The problem is that there are no "unsigned integers" on RISC-V, so if $\chi[11] = 1$, when we load "*lo*" the processor will sign-extend it, effectively subtracting 1 from "*hi*". Therefore we need to compensate:

```
#define RISCV_IMM_BITS 12

uint32_t lo = (uint32_t)value & ~(0xFFFFFFFF << RISCV_IMM_BITS);
uint32_t hi = (uint32_t)value & (0xFFFFFFFF << RISCV_IMM_BITS);

if (lo & (1 << (RISCV_IMM_BITS - 1)))
        {
        hi += 1 << RISCV_IMM_BITS;
        }

generateUTYPE(OpCode::_lui, node, hi, trgReg);
generateITYPE(OpCode::_addiw, node, trgReg, trgReg, lo);
```

We test for $\chi[11] = 1$ and increment "*hi*" if needed.

Now our compiler has a conditional branch. The symbolic execution of `loadConstant()` *diverges* onto two "*paths*"; so at the time we reach the return from `loadConstant()`, we have a "*stash*" of two unrelated "*active states*". In this case, the SimEngine doesn't do all the work for us for free. This is because with the jump,

```
j 0x100
```

we had to pass control manually, because we had to re-translate the LWISEM. This piece of simulation functionality needs to be programmed manually to re-connect the fragments of the execution paths. Then we can merge the final states into one with GPR3 looking something like

If β then χ else χ

where β is some complex boolean AST involving a test of χ[11]; which again simplifies to GPR3 = χ.

5 Conclusions and Future work

We have constructed a working proof of concept which shows that native-code analysis via symbolic execution can be done in the presence of self-modification, if we augment the lifter with the capability to lift from symbolic code.

In this experiment, we arbitrarily selected the properties to give our system to reason about, looking only for simple examples. Our next step will be to work with an actual formal model of a whole intermediate language of a VM.

JIT-translating methods to native code is only one flavor of self-modification in VMs. We intend to repeat our experiment to watch a Polymorphic Inline Cache patch itself under symbolic execution.

Many aggressive optimizations are based on refinement in the presence of undefined behavior. We intend to explore how the techniques in [10] may apply in the JIT context.

Certain critical aspects of the design of a JIT compiler are dictated by multiprocessing. In this area, correctness is especially difficult to achieve [22]. Applying our approach to this problem is an important future direction.

References

[1] Robert Atkey. 2008. *CoqJVM: An Executable Specification of the Java Virtual Machine Using Dependent Types.* Springer Berlin Heidelberg, Berlin, Heidelberg, 18–32. https://doi.org/10.1007/978-3-540-68103-8_2

[2] R. Auler, P. C. Centoducatte, and E. Borin. 2012. ACCGen: An Automatic ArchC Compiler Generator. In *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing.* 278–285.

[3] Gilles Barthe, Guillaume Dufay, Line Jakubiec, Bernard P. Serpette, and Simão Melo de Sousa. 2001. A Formal Executable Semantics of the JavaCard Platform. In *Proceedings of the 10th European Symposium on Programming Languages and Systems (ESOP '01).* Springer-Verlag, London, UK, UK, 302–319. http://dl.acm.org/citation.cfm?id=645395.757559

[4] Y. Bertot and P. Castéran. 2004. *Interactive Theorem Proving and Program Development.* Springer, Berlin Heidelberg.

[5] Mantis H. M. Cheng, Douglas Stott Parker, and M. H. van Emden. 1995. A Method for Implementing Equational Theories as Logic Programs. In *ICLP.*

[6] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08).* Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[7] J. Fridstrom, A. Jacques, K. Kilpela, and J. Sarkela. 2015. Testing Modtalk. In *Lecture Notes in Business Information Processing, Agile Processes, in Software Engineering, and Extreme Programming — 16th International Conference, XP.* Helsinki, Finland.

[8] Xavier Leroy. July 2008. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2008), 107–115.

[9] H. Liu and J. S. Moore. 2003. Executable JVM Model for Analytical Reasoning: A Study. In *Workshop on Interpreters, Virtual Machines and Emulators.* ACM SIGPLAN, San Diego, California, USA.

[10] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15).* ACM, New York, NY, USA, 22–32. https://doi.org/10.1145/2737924.2737965

[11] Eliot Miranda, Clément Béra, Elisa Gonzalez Boix, and Dan Ingalls. 2018. 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 (VMIL 2018).* ACM, New York, NY, USA, 57–66. https://doi.org/10.1145/3281287.3281295

[12] Javier Pimás, Javier Burroni, and Gerardo Richarte. 2014. Design and implementation of Bee Smalltalk Runtime. In *Proceedings of the International Workshop on Smalltalk Technologies (IWST '14).* ACM.

[13] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. 2004. ArchC: a systemC-based architecture description language. In *16th Symposium on Computer Architecture and High Performance Computing.* https://doi.org/10.1109/SBAC-PAD.2004.8

[14] Boris Shingarov. 2014. Modern Problems for the Smalltalk VM. In *International Workshop on Smalltalk Technologies.* Cambridge, UK.

[15] Boris Shingarov. 2014. The Synthesis of Target-Agnostic JIT. 8th Smalltalks — Argentina Conference, Córdoba, Argentina.

[16] Boris Shingarov. 2015. Live Introspection of Target-Agnostic JIT in Simulation. In *Proceedings of the International Workshop on Smalltalk Technologies (IWST '15).* ACM. https://doi.org/10.1145/2811237.2811295

[17] Boris Shingarov. 2017. Programming a Smalltalk VM in Coq. In *Proceedings of the International Workshop on Smalltalk Technologies (IWST '17).* ACM.

[18] Boris Shingarov. 2017. Two JIT backends for RISC-V. In *7th RISC-V Workshop Proceedings.*

[19] Boris Shingarov. 2018. Dynamic Language Runtimes on RISC-V. In *8th RISC-V Workshop Proceedings.*

[20] Boris Shingarov and Jan Vraný. 2019. Status Update: Dynamic Language Runtimes on RISC-V. Week of Open Source Hardware — Zürich, Switzerland.

[21] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP).* IEEE Computer Society, Los Alamitos, CA, USA, 138–157. https://doi.org/10.1109/SP.2016.17

[22] Vijay Sundaresan, Daryl Maier, Pramod Ramarao, and Mark Stoodley. 2006. Experiences with Multi-threading and Dynamic Class Loading in a Java Just-In-Time Compiler. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '06).* IEEE Computer Society, Washington, DC, USA, 87–97. https://doi.org/10.1109/CGO.2006.16