

High-performance language virtual machines: an analysis and challenges

Stéphane Ducasse, Guillermo Polito, Pablo Tesone, Gaël Thomas, Loïc

Lagadec

▶ To cite this version:

Stéphane Ducasse, Guillermo Polito, Pablo Tesone, Gaël Thomas, Loïc Lagadec. High-performance language virtual machines: an analysis and challenges. 2022. hal-03770053

HAL Id: hal-03770053 https://hal.inria.fr/hal-03770053

Submitted on 6 Sep 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers. L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

High-Performance Language Virtual Machines: an analysis and challenges

S. Ducasse G. Polito [RMOD/Evref - Inria] P. Tesone [Pharo consortium] **G. Thomas [Telecom SudParis]** L. Lagadec [ENSTA]

March 2022



Language VMs are Ubiquitous

- They are everywhere: browsers, mobile phones, drones, robots...
 - Banks, servers, aircrafts
- Portability, self-optimisation and adaptation, high-level services (GC)
 - Java, Javascript, Pharo, PHP, Python, Ruby, C#...
 - Derivates: Typescript, Scala, web assembly

Language Virtual Machines **Modern Language Implementations**

Managed **Execution**



Managed Memory





Runtime Binary Translation



Hardware/System Interaction



Key Players

- Javascript: Safari (Apple), V8 (Google), SpiderMonkey (Mozilla)
- Java: Truffle, GraalVM (Oracle)
- .NET, C#, VB: (Microsoft) lacksquare





VMs as Competitive Advantage

Large companies developed their OWN

- Hack: Facebook's PHP
- Ruby: Shopify
- GemTalk Systems
- Netflix NETFLIX
- Many python, ruby are popping up

Virtual Machines Typical Architecture Overview





Compiler Pipeline Example source code - to - bytecode interpreter. Example: arithmetics

a + b

push a push b send +

source code bytecode

```
send_+(op1, op2){
if (isInteger(op1) && isInteger(op2)) {
   r = op1 + op2;
   if (!overflow){
     return push(r);
send message(+)
```

interpreter code

Ş

Interpreter and Compiler Semantics

1	Interpreter >> bytecodePrimAdd
2	rcvr arg result
3	rcvr := self internalStackValue: 1.
4	arg := <mark>self</mark> internalStackValue: 0.
5	(objectMemory areIntegers: rcvr and: arg) if True: [
6	result := (objectMemory integerValueOf: rcvr) + (
	objectMemory integerValueOf: arg).
7	"Check for overflow"
8	(objectMemory isIntegerValue: result) if True: [
9	self
10	internalPop: 2
11	thenPush: (objectMemory integerObjectOf: result).
12	^ self fetchNextBytecode "success"]].
13	"Slow path, message send"
14	self normalSend

1	# previous bytecode IR
2	checkSmallInteger t0
3	jumpzero notsmi
4	checkSmallInteger t1
5	jumpzero notsmi
6	t2 := t0 + t1
7	jumplfNotOverflow continue
8	notsmi: #slow case first send
9	t2 := send #+ t0 t1
10	continue:
11	# following bytecode IR



Quid Complexity and Cost of VMs? Apple's Safari JavascriptCore[2021]

https://webkit.org/blog/10308/speculation-in-javascriptcore/ https://ponyfoo.com/articles/an-introduction-to-speculative-optimization-in-v8

'n	11	0	n	п
		c		u

DFG Byte Code Parser (the frontend)
Live Catch Variable Preservation
CPS Rethreading
Unification
Prediction Injection
Static Execution Count Estimation
Backwards Propagation
Prediction Propagation
Fixup
InvalidationPoint Injection
Type Check Hoisting
Strength Reduction
CPS Rethreading
Abstract Interpreter
Constant Folding
CFG Simplification
Local Common Subexpression Elimination
CPS Rethreading
Abstract Interpreter
Constant Folding
Clean Up
Critical Edge Breaking
Loop Pre Header Creation
CPS Rethreading
SSA Conversion
SSA Lowering
Arguments Elimination
Put Stack Sinking
Constant Hoisting
Global Common Subexpression Elimination
Liveness Analysis

¥
Abstract Interpreter
Constant Folding
Clean Up
Strength Reduction
Critical Edge Breaking
Object Allocation Sinking
ValueRep Reduction
Liveness Analysis
Abstract Interpreter
Constant Folding
Liveness Analysis
Abstract Interpreter
Loop Invariant Code Motion
Liveness Analysis
Integer Range Optimization
Clean Up
Integer Check Combining
Global Common Subexpression Elimination
Liveness Analysis
Abstract Interpreter
Global Store Barrier Insertion
Store Barrier Clustering
MovHint Removal
Clean Up
Dead Code Elimination
Stack Layout
Liveness Analysis
OSR Availability Analysis
Watchpoint Collection
Lower DFG to B3
Reduce Double To Float

Legend

DFG SSA IR

B3 IR

*
Reduce Strength
Hoist Loop Invariant Values
Eliminate Common Subexpression
Eliminate Dead Code
Infer Switches
Reduce Loop Strength
Duplicate Tails
Fix SSA
Fold Path Constants
Lower Macros
Optimize Associative Expression Tre
Reduce Strength
Lower Macros After Optimizations
Legalize Memory Offsets
Move Constants
Eliminate Dead Code
Lower B3 to Air
Simplify CFG
Lower Macros
Eliminate Dead Code
Allocate Registers By Graph Colori
Fix Obvious Spills
Lower After Reg Alloc
Allocate Stack By Graph Coloring
Lower Stack Args
Report Used Registers
Fix Partial Register Stalls
Lower Entry Switch
Simplify CFG
Optimize Block Order
Generate (the backend)

DFG IR

Managed Execution Remarkable Challenges

- Challenge 1: What are optimal organisations of multi-tier engines?
 - Combining interpreters with many levels of optimising compilers
- Challenge 2: What is a better/minimal runtime support for developer tooling?
 - Better debugging support
 - Runtime (speed, energy...) profiling
 - Benchmark automatic generation

Runtime Binary Translation Remarkable Challenges

VMs are *auto-adaptive* systems

- Challenge 3: How can runtime-compilers better speculate on application behaviour?
 - Speculate on more than types
 - Speculate for more than speed
- Challenge 4: How can we improve the efficiency of *cold code*?
 - Better interpreter optimisations
 - Low overhead binary translators

Managed Memory Remarkable Challenges

patterns?

- Scalability to *multi-TB* heaps
- Automatically memory re-organisation
- Reduce pauses
- encrypted memory (arm trustzone/intel sgx), compressed memory
- OS and System VM Interations

Challenge 5: How can *managed memory adapt* to memory consumption

• Support for modern hardware (e.g., disaggregated memory, non-volatile memories)

Hardware/System Interaction **Remarkable Challenges**

Challenge 6: How can modern VMs exploit hardware-software codesign?

- Automatic deport computation to dedicated hardware
 - GPU
 - FPGA
 - Extensible ISAs (e.g., RISC-V)

Cross-Cutting Challenges (And Contradictory Challenges!)

Energy Consumption

Execution Speed

Security

Correctness

Modularity

Cross-Cutting Challenges Selected Challenges

- Security threats of multi-tier execution engines
- Speculative runtime compilation for frugal systems
- Securing VMs through dedicated hardware
- Minimising energy impact of garbage collection algorithms

Selected Software Engineering Challenges

- Automatic detection of performance regressions
- Automatic validation of multi-tier execution engines
- Controling the construction COST of efficient JIT compilers

AlaMVic: a generative approach

- **Compiler generation**
- Exchangeable components
- Optimization heuristics
- Open exploratory platform

Early RMOD achievements Dev side of things

- JIT for Apple M1, Windows, Raspberry ARM 64bits in production
- Helping ENSTA Bretagne to develop a RISC-V JIT
- Streamlining transpilation/compilation chain
- Taking advantage of VM tests [MPLR, MoreVM paper]
- Some productivity enhancer tools (Unicorn simulator, assembly browser, interactive CFG navigation,...)

Early RMOD achievements Research side

Hansen BB reordering)

- Reducing manual code (~100 bytecodes, ~300 primitives)
 - RQ: Are interpreted and compiled code equivalent? Concolic + differential testing
 - RQ: Can we generate JIT compilers? Abstract interpreter for compiled code generation (underway)

• RQ: static code reordering: is it worth ? (alternative to Pettis-

Benagil Research side

- J-NVM: Efficient integration of a persistent memory in a Java Virtual Machine
- PrivaDSL: Use of Intel SGX in a Java virtual machine
- Study of a Java virtual machine for disaggregated memory
- A shell language and runtime for serverless applications

Early ENSTA achievements Dev side of things

- RISC-V JIT for a production level VM Pharo consortium
- RISC-V board

Research side

- Study language VM level attacks

Starting to propose protections against language VM-level attacks

Language VMs are strategical assets

- Controlling the execution engine, controls the world
- French research should not miss the opportunity
- Independence from the will of big companies is crucial for research
- Rare french teams on the topic should be supported!

