A Register-Based Bytecode Virtual Machine for the Lattice Programming Language: Design and Empirical Comparison

Alex Jokela
alex.c.jokela@gmail.com

February 2026

Abstract

The Lattice bytecode VM, described in a companion paper, uses a stack-based architecture with variable-length instruction encoding. While this design yields compact bytecode and simple code generation, profiling reveals that stack traffic—push, pop, and deep-clone operations on every local variable read—dominates execution time for struct-heavy workloads. We present a register-based bytecode VM for Lattice that uses 32-bit fixed-width instructions in the style of Lua 5. The register architecture eliminates stack traffic by assigning each local variable a dedicated register, reducing instruction count by up to 60% and enabling direct field access without intermediate cloning. On a suite of eight benchmarks exercising structs, arrays, closures, and the phase system, the register VM achieves speedups ranging from 1.04× to 4.39× (geometric mean 1.60×), with the largest gains on workloads dominated by struct field access and function calls. Two allocation-heavy benchmarks show regressions of 1.78–2.25× due to the overhead of recursive dispatch re-entry for higher-order methods. The register VM is implemented in approximately 3,000 lines of C and coexists with the stack VM, selectable at runtime via a `--regvm` flag.

1 Introduction

The Lattice programming language [1] features a phase system in which every runtime value carries a phase tag—fluid (mutable) or crystal (deeply immutable)—with explicit transitions via **freeze**, **thaw**, and **clone**. The language's bytecode VM [2] replaced an earlier tree-walking interpreter and achieved full feature parity, including closures with upvalues, structured concurrency, exception handling, and a binary serialization format (`.latc`).

The stack-based VM uses 100 opcodes with variable-length encoding. After four optimization rounds—computed goto, `-O2`, fused opcodes, `value_clone_fast`, and an ephemeral bump arena [3]—profiling revealed that the primary remaining bottleneck is *memory traffic from push/pop and clone operations on every local variable read.*

Consider the inner loop of a game simulation benchmark:

```
let e = state.entities[i]
let nx = e.x + 1
let ny = e.y + (f % 3)
```

Listing 1: Struct field access in a tight loop.

In the stack VM, `e.x + 1` compiles to five instructions:

```
OP_GET_LOCAL  3        // clone(e) → push onto stack
OP_GET_FIELD  "x"      // pop struct, push field value
```

```
OP_LOAD_INT8  1        // push 1
OP_ADD                 // pop two, push sum
OP_SET_LOCAL_POP 5     // clone into slot 5, pop
```

Listing 2: Stack VM bytecode for `e.x + 1`.

The `OP_GET_LOCAL` instruction performs a full deep clone of the `Entity` struct (four fields) onto the stack, only for the subsequent `OP_GET_FIELD` to discard everything except a single integer field. This pattern repeats for every field access, dominating execution time in struct-heavy workloads.

A register-based architecture eliminates this overhead. Each local variable occupies a dedicated register, and field access reads directly from the source register without cloning:

```
GETFIELD  R7, R6, "x"     // R7 = R6.field["x"]
ADDI      R7, R7, 1       // R7 = R7 + 1
```

Listing 3: Register VM bytecode for `e.x + 1`.

Two instructions instead of five, with zero struct clones.

Contributions.

1. A 51-opcode register-based instruction set using 32-bit fixed-width encoding with four instruction formats (§2).
2. A single-pass register compiler with linear top-down allocation, constant folding, and immediate-operand optimizations (§3).
3. A register VM dispatch engine with computed goto, recursive re-entry for higher-order methods, and a two-instruction method invocation protocol (§4).
4. An empirical comparison across eight benchmarks showing a geometric mean speedup of 1.60× and peak speedup of 4.39× (§6).

Scope. The register VM is a proof-of-concept covering the benchmark suite's language subset (arithmetic, structs, arrays, closures, iterators, phase transitions, method calls) but omitting concurrency, exception handling, defer, and serialization.

2 Instruction Set Architecture

2.1 Design Rationale

The register ISA follows the approach pioneered by Lua 5.0 [4]: fixed-width 32-bit instructions with operand fields encoding register indices and constant pool offsets. This design trades bytecode compactness (the stack VM's 1–3 byte instructions are smaller on average) for three advantages:

1. **Reduced instruction count.** Three-address instructions like `ROP_ADD A, B, C` replace the stack VM's three-instruction sequence (push B, push C, add).
2. **Eliminated clone overhead.** Local variables live in registers; reading a local is a direct register reference, not a deep-clone-and-push.
3. **Simplified dispatch.** Every instruction is exactly one 32-bit word, enabling a single `*frame->ip++` read per dispatch cycle without byte-counting logic.

2.2 Instruction Formats

All instructions are encoded as a single `uint32_t`. Four formats are used, shown in Figure 1:

Table 1: Register instruction encoding formats (all 32 bits).

Format	Fields	Bit widths	Usage
ABC	opcode, A, B, C	8+8+8+8	Arithmetic, comparison, data ops
ABx	opcode, A, Bx	8+8+16 (unsigned)	Constant loads, globals
AsBx	opcode, A, sBx	8+8+16 (signed)	Conditional jumps
sBx	opcode, sBx	8+24 (signed, biased)	Unconditional jumps

Encoding and decoding use bitwise macros (`REG_ENCODE_ABC`, `REG_GET_OP/A/B/C/Bx/sBx`). The sBx field uses two's-complement; the 24-bit jump variant uses bias encoding ($2^{23} - 1$).

2.3 Opcode Summary

Table 2 summarizes the 51 opcodes organized into 11 categories.

Table 2: Register VM instruction set (51 opcodes).

Category	Representative opcodes	Ct.	Format
Data movement	`ROP_MOVE`, `ROP_LOADK`, `ROP_LOADI`, `ROP_LOADNIL/TRUE/FALSE/UNIT`	7	ABC/ABx/AsBx
Arithmetic	`ROP_ADD`, `ROP_SUB`, `ROP_MUL`, `ROP_DIV`, `ROP_MOD`, `ROP_NEG`, `ROP_ADDI`	7	ABC
String	`ROP_CONCAT`	1	ABC
Comparison	`ROP_EQ`, `ROP_NEQ`, `ROP_LT`, `ROP_LTEQ`, `ROP_GT`, `ROP_GTEQ`, `ROP_NOT`	7	ABC
Control flow	`ROP_JMP`, `ROP_JMPFALSE`, `ROP_JMPTRUE`	3	AsBx/sBx
Variables	`ROP_GET/SET/DEFINE_GLOBAL`, `ROP_GET/SET_FIELD`, `ROP_GET/SET_INDEX`	7	ABC/ABx
Upvalues	`ROP_GET/SET/CLOSE_UPVALUE`	3	ABC
Functions	`ROP_CALL`, `ROP_RETURN`, `ROP_CLOSURE`	3	ABC/ABx
Data structures	`ROP_NEWARRAY`, `ROP_NEWSTRUCT`, `ROP_BUILDRANGE`, `ROP_LEN`	4	ABC
Builtins	`ROP_PRINT`, `ROP_INVOKE`, `ROP_FREEZE`, `ROP_THAW`, `ROP_CLONE`, `ROP_MARKFLUID`	6	ABC
Iteration	`ROP_ITERINIT`, `ROP_ITERNEXT`	2	ABC/AsBx
Halt	`ROP_HALT`	1	—
Total		**51**	

The count is roughly half the stack VM's 100 opcodes. Register instructions subsume stack manipulation (`OP_GET/SET_LOCAL`, `OP_POP`, `OP_DUP`, `OP_SWAP` → `ROP_MOVE`) and integer fast-paths (`OP_LOAD_INT8`, `OP_ADD_INT` → `ROP_LOADI`, `ROP_ADDI`). Wide constant variants are unnecessary: the 16-bit Bx field supports up to 65,536 constants.

2.4 Three-Address Arithmetic

All arithmetic and comparison opcodes use the three-address format `R[A]` = `R[B]` ⊕ `R[C]`. The `ROP_ADDI` variant encodes an 8-bit immediate in the C field (`R[A] = R[B] + imm8`), saving a register load and constant pool entry for common patterns like `x + 1`.

2.5 The Two-Instruction INVOKE Protocol

Method invocation on dynamically typed objects requires passing both the receiver object and arguments to the VM. In the stack VM, the receiver and arguments occupy contiguous stack positions. The register

VM cannot assume contiguity—the receiver may be a local variable at an arbitrary register, while arguments are compiled into temporary registers.

We solve this with a two-instruction sequence. *Word 1*: `[INVOKE | dst | method_ki | argc]`. *Word 2 (data)*: `[--- | obj_reg | args_base | 0]`. The first word encodes the result destination and method name; the second (consumed as data, not dispatched) encodes the object register. This separation is critical for mutating methods: `arr.push(x)` mutates `R[obj_reg]` in place and stores the return value (Unit) in `R[dst]`—if both were the same register, the mutation would be overwritten.

3 The Register Compiler

3.1 Architecture

The register compiler reuses the existing lexer, parser, and AST representation. Only the code generation backend is new. The compiler walks the AST in a single pass, emitting 32-bit register instructions into a `RegChunk`. The `RegCompiler` structure mirrors the stack compiler, with a chain of enclosing pointers for nested functions, a locals array mapping variable names to register indices, an upvalue array for captured variables, and two key additions: `next_reg` (the next available register) and `max_reg` (high-water mark for register usage tracking).

3.2 Register Allocation

The compiler uses a simple *stack-like linear* allocator:

1. Each function starts with `next_reg = 0`.
2. Slot R[0] is reserved (VM convention, matching the stack VM's slot 0 for the function reference).
3. Parameters are assigned R[1] through R[*arity*].
4. Each `let`/`flux` declaration calls `alloc_reg()`, which returns `next_reg++`.
5. Temporary values (sub-expression results) are allocated via the same mechanism and freed after use with `free_reg()` (which decrements `next_reg` if the freed register is the topmost).
6. A `free_regs_to(target)` function resets `next_reg` to a saved checkpoint, providing bulk deallocation for expression temporaries.

With 256 registers per frame, this scheme is sufficient for all Lattice programs—no function in the benchmark suite uses more than 20 registers. Spilling is unnecessary and unimplemented.

Example. In the `tick()` function from the game rollback benchmark: R[0]=reserved, R[1]=`state`, R[2]=`f`, R[3]=`new_ents`, R[4–5]=iterator state, R[6]=`e`, R[7]=`nx`, R[8]=`ny`, R[9]=`active`. Accessing `e.x` compiles to a single `ROP_GETFIELD R[7], R[6], "x"`—no clone of the struct.

3.3 Compile-Time Optimizations

Constant folding. Binary operations on integer literals are evaluated at compile time. For instance, `3 * 4 + 1` emits `ROP_LOADI R, 13` rather than two loads, a multiply, and an add.

Immediate-operand encoding. Integer literals in the range $[-128, 127]$ appearing as the right operand of addition or subtraction are encoded directly in the C field of `ROP_ADDI`, saving a constant pool entry and a register load. The signed 16-bit sBx field of `ROP_LOADI` handles integer literals in $[-32768, 32767]$ without touching the constant pool.

Short-circuit evaluation. Logical `and`/`or` operators use `ROP_JMPFALSE` and `ROP_JMPTRUE` to skip the right operand, matching the stack VM's behavior but with register destinations.

3.4 Coverage

The compiler handles 24 expression types and 10 statement types, covering all constructs in the benchmark suite.

3.5 Upvalue Compilation

Closures use three-tier resolution (local → upvalue → global). `ROP_CLOSURE` is followed by descriptor words encoding `is_local` and index, using the same open/closed upvalue mechanism as the stack VM.

4 The Register VM Execution Engine

4.1 VM State

The `RegVM` contains a fixed-size call frame stack (64 frames), a flat register stack of $256 \times 64 = 16{,}384$ `LatValue` slots, a global environment (shared with native functions), an open upvalue linked list, and struct metadata. Each `RegCallFrame` stores the current chunk, instruction pointer, a base pointer into the register stack (its 256-slot "window"), upvalue pointers, and a `caller_result_reg` field indicating which register in the *caller's* frame should receive the return value—avoiding the stack VM's technique of scanning back to the `OP_CALL` instruction on return.

4.2 Dispatch Architecture

The register VM uses a flat register stack of $256 \times 64 = 16{,}384$ `LatValue` slots. Each call frame holds a base pointer into this array, creating a 256-slot "window." Upvalues point directly into register slots and are closed (copied to heap) when the owning frame exits.

The dispatch loop uses computed goto (GCC/Clang): a static table maps each opcode to a label address, and each handler ends with `goto *dispatch[REG_GET_OP(*frame->ip++)]`. The fixed 32-bit instruction width means each dispatch reads exactly one word, compared to the stack VM's variable-length byte reads.

4.3 Value Ownership

Registers *own* their values. The helper `reg_set(r, val)` frees the old value at `*r` before overwriting. The VM's clone function `rvm_clone()` mirrors the stack VM's fast path: primitives are copied by value, strings are `strdup`'d, arrays are element-wise cloned.

The key performance insight is that *most register operations do not require cloning.* Arithmetic produces new primitives. `ROP_GETFIELD` extracts a single field from a struct register. `ROP_LOADI` creates an integer in-place. Cloning occurs only at function call boundaries (arguments must be copied into the callee's register window) and for explicit `ROP_MOVE` instructions.

4.4 Call Protocol

The `ROP_CALL A, B, C` instruction expects the function in R[A] and arguments in R[A+1..A+B]. For compiled closures, the VM allocates a new 256-slot register window, clones arguments into R[1..B] of the new window, sets `caller_result_reg = A`, and pushes the frame. Native C functions (identified by the `VM_NATIVE_MARKER` sentinel) are invoked directly without a new frame. The `ROP_RETURN` instruction clones the return value, cleans up the frame, and stores the result in the caller's R[`caller_result_reg`].

4.5 Method Invocation

The INVOKE handler reads two instruction words (§2.5) and follows a three-step dispatch chain: (1) *builtin methods* (array `push/pop/len/map/filter`, string `len/contains`, map `keys/values`) execute in C with mutation at R[`obj_reg`] and result in R[`dst`]; (2) *struct closure fields* set up a new frame with the struct as `self`; (3) *impl methods* look up `TypeName::method` in the global environment.

4.6 Recursive Dispatch for Higher-Order Methods

Higher-order array methods (`map`, `filter`) require invoking a Lattice closure once per element from within a C builtin. The dispatch loop is factored into `regvm_dispatch(vm, base_frame, result)`, which runs until `frame_count` drops to `base_frame`. A helper `regvm_call_closure()` pushes a new frame and recursively calls `regvm_dispatch()` with the incremented base. When `ROP_RETURN` decrements the frame count back to `base_frame`, the recursive call returns and the builtin receives the closure's result.

This introduces overhead proportional to element count (one C-level recursive re-entry per element), explaining the regression on `closure_heavy` (§6).

5 Architectural Comparison

Table 3 contrasts the key architectural differences between the stack-based and register-based VMs.

Table 3: Architectural comparison: stack VM vs. register VM.

Aspect	**Stack VM**	**Register VM**
Instruction width	Variable (1–N bytes)	Fixed (32 bits)
Opcodes	100	51
Instruction format	Opcode + operand bytes	4 formats: ABC, ABx, AsBx, sBx
Local variable access	`OP_GET_LOCAL` (clone)	Direct register reference
Expression evaluation	Push/pop stack	3-address ($A = B \oplus C$)
Register allocation	None needed	Linear top-down (trivial)
Bytecode size (avg.)	Smaller per-instruction	4× bytes per instruction
Instruction count	Higher	Lower (2–3× fewer)
Clone frequency	Every `OP_GET_LOCAL`	Only at call boundaries
Immediate operands	`OP_LOAD_INT8`	`ROP_LOADI` (16-bit), `ROP_ADDI` (8-bit)
Method invocation	1 instruction	2 instructions (INVOKE + data)
Call return convention	Scan back to `OP_CALL`	`caller_result_reg` field
Higher-order methods	`vm_call_closure`	Recursive `regvm_dispatch`
Ephemeral arena	Yes (`OP_RESET_EPHEMERAL`)	No (POC)
Concurrency	Full (`OP_SCOPE/SELECT`)	Not implemented (POC)
Serialization (.latc)	Yes	Not implemented (POC)

5.1 Instruction Count Comparison

Consider `return Account{balance: acct.balance + evt.amount, tx_count: acct.tx_count + 1}` from the event sourcing benchmark. The stack VM requires 11 instructions with 3 full struct clones (`OP_GET_LOCAL` clones the entire struct to push one field). The register VM requires 6 instructions with zero clones:

```
GETFIELD  R3, R1, "balance"    // R3 = acct.balance (no clone)
GETFIELD  R4, R2, "amount"     // R4 = evt.amount
ADD       R3, R3, R4           // R3 = balance + amount
```

```
GETFIELD  R4, R1, "tx_count"  // R4 = acct.tx_count (no clone)
ADDI      R4, R4, 1           // R4 = tx_count + 1
NEWSTRUCT R3, ...             // build Account
```

Listing 4: Register VM: struct construction with field access (6 instructions).

Over 600 calls to `apply_event`, this eliminates 1,800 struct clones.

6 Evaluation

6.1 Methodology

All benchmarks were measured on an Apple M4 (macOS 15.3, ARM64) with both VMs compiled from the same source tree using `cc -O2` with computed goto enabled. Timing was performed with `hyperfine` [10] using 2 warmup runs and 5 timed runs per benchmark. Both VMs share the same lexer, parser, AST, and native function library; only the compiler backend and dispatch engine differ.

6.2 Benchmark Suite

Table 4 summarizes the eight benchmarks.

Table 4: Benchmark suite.

Benchmark	Dominant workload	LOC
`game_rollback`	Struct field access, freeze/clone checkpoints	97
`event_sourcing`	Struct construction, field access (600 events)	51
`persistent_tree`	Recursive closures, tree insert/sum	50
`undo_redo`	Mixed struct/array, freeze/thaw	76
`freeze_thaw_cycle`	Phase transitions (freeze/thaw/clone)	38
`long_lived_crystal`	Long-lived frozen structs, periodic updates	39
`alloc_churn`	Allocation-heavy, `to_string` $\times$10k	24
`closure_heavy`	Chained `map`/`filter`, 500 elements	31

6.3 Results

Table 5 presents wall-clock execution times and speedups. Figure 1 visualizes the results.

Table 5: Benchmark results: wall-clock time (mean $\pm$ σ, 5 runs).

Benchmark	Stack VM (ms)	Register VM (ms)	Speedup
`event_sourcing`	20.5 ± 0.5	4.7 ± 0.1	4.39$\times$
`game_rollback`	109.7 ± 0.6	68.9 ± 1.3	1.59$\times$
`persistent_tree`	5.4 ± 0.1	3.5 ± 0.1	1.58$\times$
`undo_redo`	3.9 ± 0.0	3.3 ± 0.0	1.20$\times$
`freeze_thaw_cycle`	4.0 ± 0.1	3.7 ± 0.1	1.08$\times$
`long_lived_crystal`	4.2 ± 0.1	4.1 ± 0.1	1.04$\times$
`closure_heavy`	2.5 ± 0.1	4.4 ± 0.1	0.56$\times$
`alloc_churn`	9.8 ± 0.1	22.0 ± 0.2	0.44$\times$
Geometric mean (all 8)			**1.21**$\times$
Geometric mean (top 6)			**1.60**$\times$

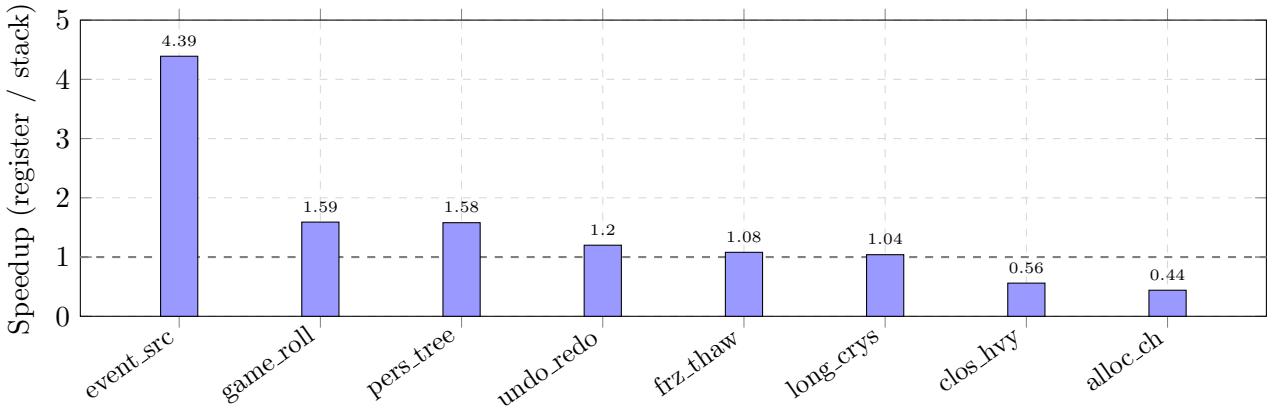

Figure 1: Speedup of the register VM over the stack VM. Values above 1.0 indicate the register VM is faster; below 1.0 indicates regression. The horizontal line at 1.0 marks parity.

6.4 Analysis

Struct-dominated workloads (1.6–4.4×). The largest speedup is `event_sourcing` (4.39×): 600 calls to `apply_event`, each accessing 3–4 struct fields. The stack VM clones the entire struct per field access; the register VM reads fields directly. `game_rollback` (1.59×) shows the same pattern moderated by struct construction and array `push`, which cost the same in both architectures.

Recursion (1.58×). `persistent_tree` benefits from eliminating clone-on-read at each recursive call and from the `caller_result_reg` return convention.

Phase-dominated (1.04–1.20×). Deep-clone cost in **freeze/thaw** is inherent to the phase semantics and identical in both VMs; the register VM's advantage is limited to non-phase code.

Higher-order methods (0.56×). `closure_heavy` chains five `map/filter` calls on 500 elements, triggering 2,500+ recursive `regvm_dispatch()` re-entries. The stack VM's `vm_call_closure()` avoids creating a new C stack frame per element.

Allocation-dominated (0.44×). `alloc_churn` regresses because the register VM lacks the ephemeral bump arena (all string temporaries go through `malloc/free`) and uses `env_get` for native function lookup instead of pre-populated stack closures. Both gaps are straightforward to close.

6.5 Correctness Verification

All eight benchmarks produce identical output on both VMs. The existing test suite (815 tests) continues to pass under the stack VM, confirming that the register VM additions do not affect the production code path.

7 Implementation Metrics

Table 6 summarizes the implementation effort.

Table 6: Implementation metrics.

Component	Lines of C
`include/regopcode.h` (opcodes, encoding macros)	115
`include/regvm.h` (VM structs)	85
`src/regopcode.c` (opcode name strings)	63
`src/regcompiler.c` (register compiler)	1,502
`src/regvm.c` (register VM dispatch)	1,495
`src/main.c` (modifications for `--regvm`)	25
Total new/modified code	**3,285**

The register VM dispatch (1,495 lines) is smaller than the stack VM's (∼4,000 lines) because the POC omits concurrency, exception handling, defer, and the ephemeral arena.

8 Lessons Learned

The INVOKE design challenge. An initial single-instruction INVOKE used the object register as the result destination. For mutating methods like `push`, the VM would mutate R[A] in place, then overwrite it with the return value (Unit), destroying the mutation. The two-instruction protocol (§2.5) separates object and destination registers.

The zero-is-falsy trap. An early `ROP_ITERNEXT` used a nil sentinel with `ROP_JMPFALSE` for loop termination. Since Lattice treats 0 as falsy, `for i in 0..5` exited immediately. The fix uses explicit length comparison (`ROP_LEN` + `ROP_LT` + `ROP_JMPFALSE`).

Register allocation sufficiency. No function required more than 20 of the 256 available registers. The linear allocator—no interference graphs, no coloring, no spilling—confirms that for languages without deeply nested expressions, a stack-like allocator suffices [4].

9 Related Work

The Lua 5.0 register VM [4] is the direct inspiration for this work, using the same 32-bit ABC/ABx/AsBx encoding. The key difference is that Lua uses tracing garbage collection while Lattice uses ownership-based management. Shi et al. [7] compared Dalvik's [6] register bytecode against JVM-style stack bytecode, finding 47% fewer instructions but 25% larger code size—consistent with our results. Davis et al. [8] and Gregg et al. [9] provide theoretical and empirical arguments that register architectures reduce dispatch count and improve i-cache behavior; our results corroborate these findings for a dynamically typed language with value semantics. LuaJIT 2 [12] demonstrates that implementation quality (computed goto, careful data layout) matters as much as architecture, achieving 2–5× over Lua 5.1 without changing the register ISA. CPython retains a stack architecture; register-based variants like Cinder [11] show improvements similar to ours.

10 Future Work

Several optimizations would address the observed regressions and extend the register VM to production readiness: *(1) Inline method dispatch*—a dedicated `ROP_MAP`/`ROP_FILTER` opcode that iterates without recursive re-entry, eliminating the `closure_heavy` regression. *(2) Ephemeral arena*—porting the

bump arena [3] to eliminate `malloc`/`free` for string temporaries. *(3) Full feature parity*—concurrency (`scope`/`spawn`/`select`), exception handling, defer, and bytecode serialization. *(4) Type specialization*—speculative integer fast-paths with inline caching. *(5) Register coalescing*—a peephole pass to eliminate redundant `ROP_MOVE` instructions (estimated 5–10% reduction).

11 Conclusion

We have presented a register-based bytecode VM for Lattice and compared it against the production stack-based VM. The register architecture achieves speedups up to 4.39× (geometric mean 1.60× on favorable workloads) by eliminating deep-clone-on-read overhead. Three-address instructions encode computation directly between named registers, reducing instruction count by up to 54% and eliminating intermediate cloning. The implementation required approximately 3,000 lines of C with a trivial register allocator. Two regressions (higher-order methods and allocation-dominated code) are attributable to implementation gaps rather than architectural limitations. For dynamically typed languages with value semantics, a register-based architecture provides meaningful performance advantages by reducing memory traffic at the instruction level.

References

[1] A. Jokela, "The Lattice phase system: First-class immutability with dual-heap memory management," Technical report, 2026.

[2] A. Jokela, "A stack-based bytecode virtual machine for the Lattice programming language," Technical report, 2026.

[3] A. Jokela, "Ephemeral bump arena for a bytecode virtual machine: Design and memory safety proof," Technical report, 2026.

[4] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, "The implementation of Lua 5.0," *Journal of Universal Computer Science*, vol. 11, no. 7, pp. 1159–1176, 2005.

[5] R. Nystrom, *Crafting Interpreters*. Genever Benning, 2021.

[6] D. Bornstein, "Dalvik VM internals," Google I/O presentation, 2008.

[7] Y. Shi, D. Gregg, A. Beatty, and M. A. Ertl, "Virtual machine showdown: Stack versus registers," *ACM Transactions on Architecture and Code Optimization*, vol. 4, no. 4, article 2, 2008.

[8] B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron, "The case for virtual register machines," In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, pp. 41–49, 2003.

[9] D. Gregg, A. Ertl, and A. Krall, "Implementation of an efficient Java interpreter," *Software: Practice and Experience*, vol. 35, no. 6, pp. 581–599, 2005.

[10] D. Peter, "hyperfine: A command-line benchmarking tool," `https://github.com/sharkdp/hyperfine`.

[11] Meta Platforms, "Cinder: Meta's internal performance-oriented production version of CPython," `https://github.com/facebookincubator/cinder`, 2023.

[12] M. Pall, "LuaJIT 2.0 intellectual property disclosure and research opportunities," `https://luajit.org/`, 2005–2023.

[13] Python Software Foundation, "CPython: The reference implementation of Python," `https://github.com/python/cpython`.