The Lateralus Bytecode Format Specification

bad-antics | August 2024 | Virtual Machines

Abstract

This paper specifies the Lateralus bytecode format for the Lateralus Virtual Machine. The specification covers file structure, instruction encoding, type descriptors, constant pools, function tables, pipeline bytecode, debug information, and validation. The format is designed for compact representation, fast loading, and efficient execution.

1 Introduction

The Lateralus bytecode format defines the binary representation of compiled Lateralus programs for the Lateralus Virtual Machine (LVM). The bytecode format is designed for fast loading, efficient interpretation, and ahead-of-time compilation.

The format prioritizes: compact representation (small file sizes for distribution), fast validation (verify integrity before execution), and direct interpretation (minimal preprocessing before execution).

This paper specifies the bytecode format version 1.0, including: file structure, instruction encoding, type descriptors, constant pools, metadata sections, and debug information.

2 File Structure

A Lateralus bytecode file (.lbc) begins with a file header, followed by a sequence of sections. Each section has a type tag, size, and content. The format is little-endian throughout.

File header (16 bytes): magic number (4 bytes: 0x4C415458 'LATX'), format version (2 bytes: major.minor), flags (2 bytes), section count (4 bytes), total file size (4 bytes).

Section types: CODE (instruction bytecode), DATA (initialized constants), TYPE (type descriptors), FUNC (function table), META (metadata), DEBUG (debug info), LINK (import/export tables).

Section alignment: each section is aligned to 8 bytes. Padding bytes (zeros) are inserted between sections as needed. Alignment enables memory-mapped loading.

3 Instruction Encoding

Instructions use a variable-length encoding. The opcode byte determines the instruction length. Simple instructions (push, pop, add) are 1 byte. Instructions with operands are 2-9 bytes.

Opcode space (256 opcodes): 0x00-0x0F (stack operations), 0x10-0x1F (arithmetic), 0x20-0x2F (comparison), 0x30-0x3F (control flow), 0x40-0x4F (function calls), 0x50-0x5F (memory), 0x60-0x6F (pipeline), 0x70-0xFF (reserved).

Operand encoding: 8-bit operands are inline. 16-bit operands use 2 bytes little-endian. 32-bit operands (constant pool indices, jump offsets) use 4 bytes little-endian.

Stack operations: PUSH_I64 (push 64-bit integer), PUSH_F64 (push 64-bit float), PUSH_TRUE, PUSH_FALSE, PUSH_NULL, POP, DUP, SWAP, ROT3 (rotate top 3 stack values).

4 Constant Pool

The constant pool stores values that cannot be encoded inline in instructions: strings, large integers, floating-point numbers, function references, and type descriptors.

Pool entry format: type tag (1 byte), length (4 bytes), data (variable). Type tags: INT64 (0x01), FLOAT64 (0x02), STRING (0x03), FUNC_REF (0x04), TYPE_REF (0x05).

String encoding in the constant pool: UTF-8 encoded bytes preceded by a 4-byte length. Strings are not null-terminated. The length is the byte count, not the character count.

Constant pool deduplication: identical constants are stored once. The compiler assigns pool indices during compilation. Deduplication reduces file size by 10-15% for typical programs.

5 Type Descriptors

Type descriptors encode the structure of types for runtime type checking, garbage collection, and reflection. Each type has a unique descriptor in the TYPE section.

Primitive type descriptors: I8, I16, I32, I64, U8, U16, U32, U64, F32, F64, BOOL, CHAR, UNIT. Each is a single byte identifying the type.

Struct type descriptor: tag (STRUCT), field count, followed by field entries. Each field entry has: name index (constant pool), type descriptor index, and offset within the struct.

Enum type descriptor: tag (ENUM), variant count, followed by variant entries. Each variant has: name index, field count, and field descriptors. The runtime uses the tag byte to identify the active variant.

Function type descriptor: tag (FUNC), parameter count, parameter type indices, return type index. Function types are used for: function pointers, closures, and trait objects.

6 Function Table

The FUNC section contains an entry for each function in the module. Functions are indexed by their position in the table. The main function has index 0.

Function entry format: name index (constant pool), parameter count, local count, max stack depth, code offset (into CODE section), code length, type descriptor index.

Parameter encoding: each parameter has a name index and type descriptor index. Parameters are

loaded from the caller's stack frame into local variable slots 0..n-1.

Local variable encoding: locals occupy slots n..n+m-1 where n is the parameter count and m is the local count. Locals are initialized to their zero value.

7 Pipeline Bytecode

Pipeline expressions compile to a sequence of PIPE_* instructions. The pipeline bytecode enables: lazy evaluation, error propagation, and runtime pipeline optimization.

PIPE_BEGIN pushes a pipeline context onto the pipeline stack. PIPE_STAGE invokes a function as a pipeline stage. PIPE_END pops the pipeline context and leaves the result on the value stack.

Pipeline error handling: PIPE_STAGE_TRY invokes a stage that returns Result. If the stage returns Err, the pipeline short-circuits and the error is propagated. PIPE_UNWRAP extracts the Ok value.

Pipeline fusion: the LVM can detect consecutive PIPE_STAGE instructions operating on the same data and fuse them into a single operation, eliminating intermediate allocations.

8 Debug Information

The DEBUG section contains: line number tables, local variable tables, and source file references. Debug information is optional and can be stripped from release builds.

Line number table: maps code offsets to source locations. Each entry is: code offset (4 bytes), file index (2 bytes), line number (4 bytes), column number (2 bytes).

Local variable table: maps local slots to names and source spans. Each entry: slot index, name index (constant pool), start offset, end offset, type descriptor index.

9 Validation

Bytecode validation verifies structural integrity before execution. Validation checks: magic number, version compatibility, section bounds, instruction encoding, constant pool references, and type consistency.

Stack effect validation verifies that each instruction has sufficient operands on the stack. The validator simulates the stack depth at each instruction. Stack underflow is a validation error.

Type-safe validation: optional strict validation checks that operand types match instruction expectations. Type-safe validation prevents type confusion attacks in sandboxed environments.

10 Conclusion

The Lateralus bytecode format provides a compact, efficient, and safe representation for compiled

programs. The format supports fast loading, runtime validation, and ahead-of-time compilation to native code.

3.1 Arithmetic Instructions

Integer arithmetic: ADD_I64, SUB_I64, MUL_I64, DIV_I64, REM_I64. Each pops two operands, performs the operation, and pushes the result. Division by zero raises a runtime error.

Unsigned arithmetic: ADD_U64, SUB_U64, MUL_U64, DIV_U64, REM_U64. Unsigned operations differ in overflow behavior and division semantics.

Floating-point arithmetic: ADD_F64, SUB_F64, MUL_F64, DIV_F64, REM_F64. Floating-point operations follow IEEE 754 semantics. NaN propagation is automatic.

Bitwise operations: AND, OR, XOR, NOT, SHL (shift left), SHR (logical shift right), SAR (arithmetic shift right). Bitwise operations work on 64-bit integers.

Overflow checking: in debug mode, integer arithmetic checks for overflow. ADD_I64_CHECKED raises an error on overflow. Release mode uses wrapping arithmetic.

Conversion instructions: I64_TO_F64, F64_TO_I64, I32_TO_I64 (sign extend), I64_TO_I32 (truncate). Conversions check for value loss and raise errors when appropriate.

Comparison instructions: EQ, NE, LT, GT, LE, GE. Comparisons pop two operands and push a boolean result. Separate instructions exist for integer, float, and string comparisons.

Unary operations: NEG_I64 (negate integer), NEG_F64 (negate float), NOT_BOOL (boolean not). Each pops one operand and pushes the result.

Saturating arithmetic: ADD_SAT, SUB_SAT clamp results to the type's range instead of wrapping. Saturating arithmetic is used for audio processing and signal operations.

Wide multiplication: MUL_WIDE multiplies two 64-bit integers and produces a 128-bit result as two 64-bit values on the stack. Used for hash computation and cryptographic operations.

3.2 Control Flow Instructions

Unconditional jump: JUMP offset (4-byte signed offset from the current instruction). Forward jumps skip code. Backward jumps create loops.

Conditional jumps: JUMP_IF_TRUE and JUMP_IF_FALSE pop a boolean and jump if the condition holds. JUMP_IF_NULL and JUMP_IF_NOT_NULL test for null values.

Switch instruction: SWITCH count, followed by count (value, offset) pairs and a default offset. The LVM searches for a matching value and jumps to the corresponding offset.

Loop instructions: LOOP_START marks the beginning of a loop (for profiling and optimization). LOOP_END marks the end. The LVM uses loop markers for just-in-time compilation decisions.

Exception handling: TRY_BEGIN pushes an exception handler. TRY_END pops the handler. THROW raises an exception. The LVM searches the handler stack for a matching handler.

Return: RETURN pops the return value and returns to the caller. RETURN_VOID returns without a value. The LVM restores the caller's frame and instruction pointer.

Table jump: TABLE_JUMP count, followed by count offsets. The top-of-stack value indexes into the table. Out-of-bounds indices use the default offset (last entry).

Conditional move: CMOV pops a boolean, a true-value, and a false-value. Pushes the true-value if the boolean is true, otherwise pushes the false-value. Branchless conditional.

Break and continue: BREAK exits the innermost loop. CONTINUE jumps to the loop header. Both unwind the stack to the loop's stack depth before jumping.

Assert instruction: ASSERT pops a boolean and a string message. If the boolean is false, the LVM panics with the message. Assert instructions are stripped in release builds.

5.1 Generic Type Encoding

Generic types are monomorphized before bytecode generation. Each concrete instantiation gets its own type descriptor. The bytecode contains no generic type variables.

Monomorphization example: Vec[i32] and Vec[String] produce two distinct type descriptors with different field types and sizes. Each has its own method implementations.

Type descriptor indices are assigned sequentially during compilation. The first 16 indices are reserved for primitive types. User-defined types start at index 16.

Recursive type descriptors: a struct that contains a reference to itself uses the struct's own index in the field's type descriptor. The loader resolves circular references.

Tuple type descriptors: tag (TUPLE), element count, element type indices. Tuples are anonymous types. Two tuples with the same element types share the same descriptor.

Reference type descriptors: tag (REF), mutability flag, referent type index. The mutability flag distinguishes &T from &mut T. References have the same runtime representation (pointer).

Array type descriptors: tag (ARRAY), element type index. All arrays of the same element type share the descriptor. The array length is a runtime value, not part of the type.

Option type encoding: Option[T] is encoded as an enum with two variants: None (no data) and Some (contains T). The runtime uses a tag byte to distinguish the variants.

Result type encoding: Result[T, E] is encoded as an enum with Ok (contains T) and Err (contains E). Pipeline error handling uses Result type descriptors for type-safe error propagation.

Type descriptor caching: the LVM caches parsed type descriptors for fast access. Cache lookup by index is O(1). The cache is populated lazily on first access.

6.1 Calling Convention

Function call: CALL func_index pushes a new call frame. Arguments are passed on the operand stack. The callee reads arguments from local slots 0..n-1.

Tail call: TAIL_CALL func_index reuses the current frame. The LVM checks that the callee's frame size does not exceed the caller's. Tail calls enable unbounded recursion.

Indirect call: CALL_INDIRECT pops a function reference from the stack and calls the referenced function. Used for: function pointers, closures, and virtual dispatch.

Method call: CALL_METHOD trait_index, method_index. The receiver is the first argument. The LVM looks up the method in the receiver's vtable.

Native call: CALL_NATIVE native_index calls a native function registered with the LVM. Native functions are implemented in C or Rust and linked at load time.

Call frame layout: return address (instruction pointer), previous frame pointer, local variables (slots 0..n-1 for parameters, n..n+m-1 for locals), and operand stack.

Argument passing: the caller pushes arguments left-to-right. The callee sees arguments in local slots 0, 1, 2, etc. Variable-argument functions use an argument count parameter.

Return value: the callee pushes the return value onto the operand stack before RETURN. The caller finds the return value on top of the stack after the call completes.

Stack overflow detection: the LVM checks the stack depth before each call. If the depth exceeds the limit (default 1 MB), a stack overflow error is raised.

Call performance: the LVM's call instruction takes approximately 5 nanoseconds. Tail calls take 3 nanoseconds (no frame allocation). Native calls take 10 nanoseconds (FFI overhead).

7.1 Pipeline Optimization

Pipeline stage fusion: consecutive stages that perform simple transformations (map, filter) are fused into a single loop. Fusion eliminates intermediate collection allocation.

Pipeline parallelization: the LVM can detect independent pipeline stages and execute them on separate threads. Parallelization is automatic for pure stages (no side effects).

Pipeline short-circuit: filter stages that eliminate most elements are moved earlier in the pipeline. This reduces the work performed by subsequent stages.

Pipeline buffering: stages that produce variable-sized output use bounded buffers between stages. The buffer size is tuned based on runtime profiling data.

Pipeline materialization: the LVM decides when to materialize intermediate results (store in memory) versus streaming them (pass directly to the next stage).

Pipeline profiling: the LVM records execution time and data volume for each pipeline stage. Profiling

data is used for: optimization decisions, performance reporting, and bottleneck identification.

Pipeline error recovery: when a stage raises an error, the pipeline can: propagate the error (default), skip the element (filter_map semantics), or use a default value.

Pipeline backpressure: slow consumers cause fast producers to pause. Backpressure prevents memory exhaustion when pipeline stages have different throughput rates.

Pipeline tracing: in debug mode, the LVM logs each pipeline stage's input and output. Traces are written to a structured log for debugging and analysis.

Pipeline compilation: the ahead-of-time compiler converts pipeline bytecode to native code. Fused pipelines compile to tight loops. Non-fusible pipelines compile to function call sequences.

2.1 Section Format Details

Section header (12 bytes): type tag (4 bytes), content size (4 bytes), flags (4 bytes). The content follows immediately after the header, padded to 8-byte alignment.

CODE section: contains the bytecode instructions for all functions. Functions are stored contiguously. Function boundaries are defined in the FUNC section.

DATA section: contains initialized constant data. Data items are: string literals, numeric constants, and static arrays. Items are referenced by offset from the section start.

TYPE section: contains type descriptors for all types used in the module. Descriptors are referenced by index. The section starts with a count of descriptors.

META section: contains module metadata: module name, version, author, dependencies, and compilation options. Metadata is stored as key-value pairs.

LINK section: contains import and export tables. Imports list: function name, expected signature. Exports list: function name, function index, signature.

Section ordering: CODE and DATA appear first (for fast loading). TYPE and FUNC appear next (for validation). META, DEBUG, and LINK appear last (optional).

Compressed sections: sections can be compressed with LZ4 for smaller file sizes. The compression flag in the section header indicates compression. The LVM decompresses on load.

Section checksum: each section has an optional CRC32 checksum stored in the flags field. The LVM verifies checksums during loading to detect corruption.

Custom sections: section types 0x80-0xFF are reserved for custom use. Tools can store: profiling data, optimization hints, and tool-specific metadata in custom sections.

8.1 Source Map Format

Source maps enable mapping from bytecode offsets to source code locations. The source map is

stored in the DEBUG section with a compact delta-encoded format.

Delta encoding: each line table entry stores the delta from the previous entry. Typical deltas are small (1-3 bytes) resulting in compact encoding. The first entry uses absolute values.

File table: the source map references files by index. The file table stores: file path (relative to the module root), file hash (for cache invalidation), and file size.

Inline expansion tracking: when functions are inlined, the source map records the inlining chain. Each bytecode offset maps to: the inlined function's source location and the call site.

Expression-level mapping: the source map provides column-level precision. This enables: highlighting the specific expression that caused an error, not just the line.

Source map size: typically 10-15% of the CODE section size. For a 100 KB CODE section, the source map is 10-15 KB. Stripping source maps reduces file size for distribution.

Source map queries: the LVM provides efficient lookup from bytecode offset to source location. The lookup uses binary search on the sorted offset entries. Lookup time is O(log n).

Breakpoint resolution: the debugger uses source maps to convert source line breakpoints to bytecode offset breakpoints. The map provides the first bytecode offset for each source line.

Source map versioning: source maps include a format version. Newer LVM versions can read older source maps. Older LVM versions ignore unrecognized source map extensions.

Source map generation: the compiler generates source maps during code generation. Each emitted instruction records its source location. Source maps are written as a final pass.

4.1 Constant Pool Encoding

Integer constants: small integers (-128 to 127) are encoded inline in instructions (PUSH_I8). Medium integers (-32768 to 32767) use PUSH_I16. Large integers use constant pool references.

Floating-point constants: all floating-point values use constant pool entries. The encoding is IEEE 754 binary64 (8 bytes). Special values: positive infinity, negative infinity, and NaN are supported.

String interning: identical strings in the constant pool share the same entry. The compiler sorts strings and uses binary search for deduplication. Interning saves 5-10% of pool size.

Function reference constants: a function reference entry contains: module index (0 for local), function index, and arity. Function references are used for: function pointers and closures.

Type reference constants: a type reference entry contains the type descriptor index. Type references are used in: instanceof checks, dynamic dispatch, and reflection.

Constant pool layout: entries are sorted by type (integers first, then floats, strings, function references, type references). Sorting enables type-specific loading optimizations.

Constant pool indexing: pool entries are referenced by 32-bit indices. The maximum pool size is 4

billion entries (sufficient for any practical program).

Lazy constant loading: the LVM loads constant pool entries on demand. Strings are loaded when first referenced. Numeric constants are loaded during module initialization.

Constant pool sharing: multiple modules can share constant pool entries through the LINK section. Shared constants are deduplicated at link time.

Constant pool memory layout: entries are stored contiguously in memory after loading. The LVM allocates a single block for the entire pool. Entry offsets are precomputed for O(1) access.

9.1 Validation Algorithm

Two-pass validation: the first pass validates structural integrity (section boundaries, magic number, version). The second pass validates instruction encoding and type safety.

Structural validation checks: file size matches header, section sizes sum correctly, section boundaries do not overlap, and all section types are recognized.

Instruction validation: each opcode is recognized, operand sizes match the opcode, jump targets are within bounds, and constant pool references are within bounds.

Stack validation: the validator simulates the stack depth at each instruction. At each control flow merge point (jump target), the stack depth from all incoming edges must agree.

Type validation (optional): the validator checks that operand types match instruction expectations. For example, ADD_I64 requires two integer operands on the stack.

Validation performance: structural validation runs in O(n) time where n is the file size. Instruction validation runs in O(m) where m is the code size. Type validation adds O(m * k) where k is the average type descriptor size.

Validation caching: once a module passes validation, the LVM caches the result. Subsequent loads skip validation if the file hash matches the cached hash.

Malformed bytecode handling: validation errors are reported with: the section, offset, and expected versus actual values. The LVM refuses to execute invalid bytecode.

Fuzzing the validator: the validator is tested with randomly mutated bytecode files. The fuzzer verifies that: invalid files are rejected, valid files are accepted, and the validator never crashes.

Validation and security: bytecode validation is the first line of defense against malicious code. Validated bytecode cannot: access out-of-bounds memory, execute invalid instructions, or corrupt the LVM state.

3.3 Memory Instructions

Load instructions: LOAD_LOCAL slot (load from local variable), LOAD_GLOBAL index (load from global), LOAD_FIELD offset (load struct field), LOAD_ELEMENT (load array element from index on

stack).

Store instructions: STORE_LOCAL slot, STORE_GLOBAL index, STORE_FIELD offset, STORE_ELEMENT. Store instructions pop the value and store it at the target location.

Allocation: ALLOC type_index allocates an object of the given type on the heap. The LVM's garbage collector manages the allocated memory.

Array allocation: ALLOC_ARRAY type_index pops the length from the stack and allocates an array. Elements are zero-initialized. The array object includes the length.

Reference operations: REF_LOCAL slot creates a reference to a local variable. DEREF loads the value through a reference. REF_FIELD creates a reference to a struct field.

Move semantics: MOVE_LOCAL slot moves the value from a local variable (sets the local to null). The LVM checks for use-after-move in debug mode.

Clone instruction: CLONE performs a deep copy of the value on top of the stack. Clone is used when ownership semantics require a copy instead of a move.

Slice operations: SLICE pops start, end, and array. Pushes a slice (pointer, length) referencing the array's elements. Slices do not own their data.

String operations: CONCAT pops two strings and pushes their concatenation. SUBSTR pops start, length, and string. STRLEN pushes the string length.

Memory fencing: FENCE ensures memory ordering for concurrent access. The LVM translates FENCE to appropriate hardware memory barriers on the target architecture.

6.2 Closure Representation

Closure encoding: a closure is a pair (function_index, environment). The environment is a heap-allocated array of captured variables. Captured variables are stored by value (moved or cloned).

Closure creation: CLOSURE func_index, capture_count pops capture_count values from the stack and creates a closure object. The closure is pushed onto the stack.

Closure call: CALL_CLOSURE pops the closure, pushes captured variables as locals, and transfers control to the function. The captured variables occupy the first local slots.

Capture modes: by-value (CAPTURE_VALUE, moves or clones the variable), by-reference (CAPTURE_REF, stores a reference), and by-mutable-reference (CAPTURE_MUT_REF).

Closure type descriptors: a closure type includes: the function signature, the capture types, and the capture modes. The runtime uses the descriptor for garbage collection.

Closure inlining: the LVM can inline small closures at call sites. Inlining eliminates: closure allocation, capture copying, and indirect call overhead.

Closure and ownership: closures that capture by value take ownership of the captured variables. The original variables are moved and cannot be used after closure creation.

Recursive closures: a closure that references itself is created with a two-step process: allocate the closure with a placeholder, then update the placeholder with the closure's address.

Closure serialization: closures can be serialized to bytecode for transmission across process boundaries. Serialization includes: the function code and the captured values.

Closure performance: closure calls add approximately 2 nanoseconds overhead compared to direct calls. The overhead is: one indirect jump and one environment pointer load.

7.2 Pipeline Bytecode Encoding

Pipeline instruction format: PIPE_BEGIN (1 byte), PIPE_STAGE func_index (5 bytes), PIPE_STAGE_TRY func_index (5 bytes), PIPE_END (1 byte).

Pipeline context: PIPE_BEGIN pushes a context containing: the current pipeline value, error status, and stage count. Each PIPE_STAGE reads and updates the context.

Lazy pipeline encoding: PIPE_LAZY_BEGIN creates a lazy pipeline. Stages are not executed until the result is consumed. PIPE_FORCE triggers evaluation.

Pipeline collection operations: PIPE_MAP func_index applies func to each element. PIPE_FILTER func_index keeps elements where func returns true. PIPE_REDUCE func_index accumulates.

Pipeline branching: PIPE_BRANCH count, offsets creates parallel pipeline branches. Each branch receives a copy of the input. PIPE_MERGE combines branch results.

Pipeline window operations: PIPE_WINDOW size creates sliding windows of size elements. PIPE_CHUNK size creates non-overlapping chunks. Both produce sequences of sequences.

Pipeline zip: PIPE_ZIP combines two pipelines element-by-element. PIPE_UNZIP splits a pipeline of pairs into two parallel pipelines.

Pipeline take/skip: PIPE_TAKE n takes the first n elements. PIPE_SKIP n skips the first n elements. Both modify the pipeline context's element count.

Pipeline flat_map: PIPE_FLAT_MAP func_index applies func (which returns a sequence) and flattens the results into a single sequence.

Pipeline distinct: PIPE_DISTINCT removes duplicate elements using hash-based comparison. PIPE_SORT_BY func_index sorts elements by the key function.

2.2 Binary Format Versioning

Version compatibility: the LVM supports loading bytecode from: the same major version (guaranteed compatible), and the previous major version (best-effort compatibility).

Major version changes: adding new instruction opcodes, changing instruction encoding, and modifying section formats require a major version increment.

Minor version changes: adding new section types, extending metadata, and adding optional features use minor version increments. Minor changes are backward compatible.

Version negotiation: when loading bytecode, the LVM checks the version field and selects the appropriate loader. Unsupported versions produce a clear error message.

Feature flags: the header flags field indicates optional features used in the file. Flags include: compressed sections, extended type descriptors, and debug information.

Migration tools: the lateralus-migrate tool converts bytecode between versions. The tool reencodes instructions, updates section formats, and regenerates metadata.

Deprecation policy: opcodes and section types are deprecated for one major version before removal. Deprecated features generate warnings during validation.

Extension mechanism: the format supports vendor-specific extensions through: custom section types (0x80-0xFF), custom metadata keys, and custom instruction prefixes.

Format stability: the bytecode format is stabilized through a specification process. Changes require: a proposal, review period, implementation, and testing before adoption.

Backward compatibility testing: the test suite includes bytecode files from all previous versions. Each LVM release verifies that old bytecode still loads and executes correctly.

9.2 Security Model

Sandboxed execution: the LVM can run bytecode in a sandbox that restricts: file system access, network access, memory usage, and CPU time. Sandbox limits are configurable.

Capability-based security: the LVM uses capability tokens for resource access. A module can only access resources for which it holds a capability. Capabilities are non-forgeable.

Memory safety: bytecode validation ensures that: array accesses are bounds-checked, references are non-null, and type confusion is impossible. Memory safety is enforced at the bytecode level.

Code signing: bytecode files can be cryptographically signed. The LVM verifies signatures before loading. Unsigned bytecode can be rejected by policy.

Taint tracking: the LVM can track information flow through pipeline stages. Tainted data (from untrusted sources) cannot flow to sensitive sinks (file system, network) without explicit declassification.

Resource limits: the sandbox enforces: maximum memory (default 256 MB), maximum CPU time (default 30 seconds), maximum file size (default 100 MB), and maximum network connections (default 10).

Audit logging: the LVM logs security-relevant events: module loading, capability creation, resource access, and sandbox violations. Audit logs enable forensic analysis.

Privilege escalation prevention: the LVM prevents: self-modifying code, direct memory access, and system call invocation. All resource access goes through the capability system.

Supply chain security: bytecode provenance is tracked through: code signing, build reproducibility, and dependency verification. The LVM can enforce that only audited dependencies are loaded.

Vulnerability response: the LVM supports: bytecode hotpatching (replace functions without restart), emergency capability revocation, and remote kill switches for critical vulnerabilities.

5.2 Trait Object Layout

Trait objects use fat pointers: a data pointer (8 bytes) and a vtable pointer (8 bytes). The total size of a trait object reference is 16 bytes.

Vtable layout: the first entry is the destructor function pointer. The second entry is the size of the concrete type. The third entry is the alignment. Subsequent entries are trait method pointers.

Vtable generation: the compiler generates a vtable for each (concrete type, trait) pair. Vtables are stored in the DATA section and referenced by constant pool entries.

Dynamic dispatch: CALL_METHOD loads the method pointer from the vtable and performs an indirect call. Dynamic dispatch adds approximately 5 nanoseconds compared to static dispatch.

Trait object creation: MAKE_TRAIT_OBJECT pops a concrete value and a vtable reference. Pushes a fat pointer containing the data pointer and vtable pointer.

Trait object casting: CAST_TRAIT_OBJECT converts a trait object from one trait to another. The LVM looks up the target trait's vtable for the concrete type.

Multi-trait objects: an object implementing multiple traits has a vtable for each trait. CAST_TRAIT_OBJECT selects the appropriate vtable.

Trait object size: the concrete value is heap-allocated. The trait object reference (16 bytes) can be passed on the stack. This enables heterogeneous collections without boxing overhead.

Vtable deduplication: identical vtables (same methods, same layout) are deduplicated by the linker. Deduplication reduces binary size for programs with many trait implementations.

Trait object and pipelines: pipeline stages can accept trait objects as input. This enables generic pipelines that process heterogeneous data streams.

8.2 Debug Extensions

Type information for debuggers: the DEBUG section includes full type descriptors for all user-defined types. Debuggers use type information to: display values, expand structures, and evaluate expressions.

Expression evaluator: the debugger can compile and execute Lateralus expressions in the context of a paused program. The evaluator accesses local variables and global state.

Hot code replacement: in debug mode, the LVM supports replacing function bytecode without stopping the program. Hot replacement enables: rapid development iteration and live debugging.

Memory inspection: the debugger can: list all heap objects, show object reference graphs, identify memory leaks (objects with no incoming references), and compute retained size.

Performance counters: the LVM exposes: instruction count, function call count, allocation count, garbage collection count, and pipeline stage execution count.

Conditional breakpoints: the debugger evaluates a Lateralus expression at each breakpoint hit. The program stops only when the expression evaluates to true.

Watchpoints: the debugger monitors a memory location or variable for changes. When the watched value changes, the program stops and reports the old and new values.

Call stack inspection: the debugger reconstructs the call stack from frame pointer chains. Each frame shows: function name, source location, local variable values, and parameter values.

Thread debugging: for multi-threaded programs, the debugger shows: all thread states, per-thread call stacks, and synchronization primitive states (mutex held/waiting).

Remote debugging: the LVM supports debugging over a network connection. The debug protocol is JSON-based. Remote debugging enables debugging on: embedded systems, cloud instances, and CI servers.

10.1 Performance Characteristics

Interpretation speed: the LVM's threaded interpreter executes 500 million instructions per second on modern x86-64 hardware. Pipeline instructions execute at 200 million stages per second.

Memory overhead: the LVM runtime uses 2 MB of memory. Each loaded module adds: module size + constant pool + type descriptors. Typical overhead per module is 50-200 KB.

Startup time: cold start (loading and validating a module) takes 5 milliseconds for a 100 KB module. Warm start (cached validation) takes 1 millisecond.

Garbage collection: the LVM uses a generational garbage collector. Young generation collection takes 0.5 milliseconds. Full collection takes 5 milliseconds for 100 MB heap.

Ahead-of-time compilation: the lateralus-aot tool compiles bytecode to native code. AOT-compiled code runs 10-50x faster than interpreted bytecode. Compilation adds 100 milliseconds per 1000 functions.

JIT compilation: the LVM includes an optional JIT compiler that compiles hot functions to native code. JIT compilation triggers after a function executes 1000 times.

Instruction cache efficiency: the bytecode format's compact encoding keeps hot code in the instruction cache. Variable-length instructions reduce code size by 30% compared to fixed-width.

Branch prediction: the LVM's interpreter loop uses computed gotos (GCC extension) for efficient dispatch. Each instruction dispatches directly to the next handler without a central loop.

Profiling overhead: runtime profiling adds 5% overhead. Profiling data includes: per-function execution counts, per-instruction timing, and memory allocation tracking.

Comparison with other VMs: the LVM interprets bytecode 2x faster than CPython and 0.5x the speed of the JVM interpreter. With AOT compilation, performance matches native C within 2x.

3.4 Pipeline-Specific Instructions

PIPE_IDENTITY: passes the input through unchanged. Used as a placeholder in pipeline construction and for debugging. The no-op pipeline stage.

PIPE_INSPECT func_index: calls func with the pipeline value for side effects (logging, debugging) without modifying the pipeline value. The function's return value is discarded.

PIPE_TIMEOUT milliseconds: wraps the next pipeline stage with a timeout. If the stage does not complete within the time limit, the pipeline raises a timeout error.

PIPE_RETRY count: wraps the next pipeline stage with retry logic. If the stage fails, it is retried up to count times with exponential backoff.

PIPE_CACHE: memoizes the next pipeline stage's results. Repeated inputs return cached outputs. Cache size is bounded (LRU eviction). Useful for expensive pure computations.

PIPE_BATCH size: batches input elements into groups of size. The next stage receives arrays instead of individual elements. Batching improves throughput for I/O-bound stages.

PIPE_DEBOUNCE milliseconds: suppresses rapid pipeline invocations. Only the last invocation within the time window is executed. Used for: user input handling and event processing.

PIPE_RATE_LIMIT count, period: limits pipeline throughput to count invocations per period. Excess invocations are queued. Used for: API rate limiting and resource protection.

PIPE_SAMPLE rate: randomly samples elements at the given rate (0.0 to 1.0). Sampling reduces data volume for: monitoring, profiling, and approximate computation.

PIPE_MEASURE: records execution time of the next pipeline stage. Measurement results are stored in the pipeline context and accessible via the profiling API.

4.2 Constant Pool Optimization

Small integer caching: integers -1 through 255 are pre-allocated in the LVM runtime. References to these integers use a special PUSH_SMALL_INT instruction instead of pool lookups.

String deduplication at link time: the linker merges identical strings from different modules. Cross-module string deduplication saves 5-10% of total constant pool size.

Constant folding in bytecode: the compiler evaluates constant expressions (2 + 3) and stores only the result (5) in the constant pool. No runtime computation needed for constants.

Pool compaction: after dead code elimination, unreferenced pool entries are removed. Pool indices are reassigned and all references are updated. Compaction reduces file size.

Float constant canonicalization: floating-point constants with identical bit patterns share a single pool entry. Negative zero (-0.0) and positive zero (0.0) are distinct entries.

Constant pool alignment: numeric constants are aligned to their natural alignment within the pool. Alignment enables: direct memory-mapped access without copying.

Pool entry classification: entries are classified as: hot (referenced from frequently executed code), cold (referenced from rarely executed code), or dead (unreferenced). Hot entries are placed first.

Constant pool streaming: for large modules, the LVM loads the constant pool incrementally. Only entries referenced by currently executing code are loaded. This reduces startup memory.

Pool index compression: most pool references use 16-bit indices (64K entries). A 32-bit escape mechanism handles larger pools. 16-bit indices save 2 bytes per reference for typical programs.

Constant pool statistics: average pool size is 5% of total bytecode size. Strings account for 60% of pool entries. Integers and floats account for 30%. Function and type references account for 10%.

6.3 Exception Handling

Exception model: Lateralus uses a hybrid exception model. Recoverable errors use Result types (checked at compile time). Unrecoverable errors use panics (caught at runtime).

Panic bytecode: PANIC pops a string message and unwinds the stack. The LVM searches for a catch handler (TRY_BEGIN..TRY_END). If no handler is found, the program terminates.

Catch handler format: TRY_BEGIN handler_offset establishes a catch handler at the given offset. TRY_END removes the handler. Handlers are stored in a stack for nested try blocks.

Cleanup actions: CLEANUP_BEGIN..CLEANUP_END mark code that must execute during unwinding (drop calls). Cleanup actions run for every frame between the panic site and the catch handler.

Exception value: the panic value is a string message. Custom exception types are possible by wrapping them in a string representation. Future versions may support typed exceptions.

Unwinding performance: panic unwinding is slow (microseconds) compared to normal execution. Panics are intended for: logic errors, assertion failures, and unrecoverable conditions.

No-unwind optimization: the compiler marks functions that cannot panic. No-unwind functions do not

need cleanup actions, reducing code size and improving optimization opportunities.

Catch-all handler: a catch handler can catch any panic. The handler receives the panic message as a string. After catching, execution resumes after the TRY_END instruction.

Poison detection: when a panic occurs during a mutex lock hold, the mutex is marked as poisoned. Subsequent lock attempts return an error. Poison detection prevents use of inconsistent state.

Stack trace generation: during unwinding, the LVM records each frame's function name and source location. The stack trace is available in the catch handler for error reporting.

5.3 Type Descriptor Compression

Structural sharing: type descriptors that share common prefixes (same initial fields) store only the extension. A base descriptor is referenced by derived descriptors.

Small type descriptors: types with 4 or fewer fields use a compact encoding (1 byte tag + 1 byte count + inline field descriptors). Compact descriptors fit in a single cache line.

Type descriptor deduplication: identical descriptors from different modules are deduplicated by the linker. The deduplication key is the canonical encoding of the descriptor.

Lazy type descriptor resolution: forward references to type descriptors (in recursive types) are resolved lazily. The LVM patches descriptor references on first use.

Type descriptor versioning: when a type changes between module versions, the LVM can migrate old instances to the new layout. Migration uses: field mapping, default values, and conversion functions.

Inline type descriptors: small types (primitives, references, small tuples) use inline descriptors encoded directly in instructions. Inline descriptors avoid constant pool lookups.

Type descriptor statistics: average module has 50-200 type descriptors. Struct descriptors average 40 bytes. Enum descriptors average 60 bytes. Function descriptors average 20 bytes.

Type descriptor caching: the LVM maintains a hash map from type descriptor hash to resolved descriptor. Cache hit rate exceeds 99% for typical programs.

Type descriptor reflection API: the LVM provides runtime access to type descriptors for: serialization, deserialization, debugging, and dynamic type checking.

Type descriptor generation from source: the compiler generates descriptors bottom-up. Primitive types first, then compound types. Circular references use placeholder indices.

9.3 Ahead-of-Time Compilation

AOT compilation translates bytecode to native machine code before execution. The AOT compiler reads .lbc files and produces native executables or shared libraries.

Instruction lowering: each bytecode instruction is translated to one or more native instructions.

PUSH_I64 becomes a register load or immediate move. ADD_I64 becomes an add instruction.

Register allocation: the AOT compiler uses graph coloring for register allocation. Graph coloring produces better code than the interpreter's implicit stack-based allocation.

Stack elimination: the AOT compiler eliminates the operand stack. Values are kept in registers. Stack spills occur only when register pressure exceeds available registers.

Function call optimization: direct calls are translated to native call instructions. Indirect calls use computed branch targets. Tail calls are converted to jumps.

Pipeline compilation: pipeline bytecode is compiled to tight native loops. Fused stages share registers. Non-fusible stages use the standard calling convention.

Garbage collection integration: AOT-compiled code uses the same garbage collector as the interpreter. GC safe points are inserted at: function calls, loop back edges, and allocation sites.

Debug information: AOT-compiled code includes DWARF debug information derived from the bytecode's source maps. Native debuggers (GDB, LLDB) can debug AOT-compiled Lateralus programs.

AOT compilation time: compiling a 100 KB bytecode module to native code takes 200 milliseconds. The resulting native code is 2-3x larger than the bytecode.

AOT deployment: AOT-compiled programs do not require the LVM runtime for basic execution. The AOT output links against a minimal runtime (50 KB) for: memory allocation and panic handling.

7.3 Pipeline Type Checking at Load Time

Pipeline type verification: the LVM verifies pipeline types during module loading. Each PIPE_STAGE instruction's function must accept the previous stage's output type.

Type mismatch detection: if a pipeline stage receives an incompatible type, the LVM reports: the stage index, expected type, actual type, and source location (from debug info).

Generic pipeline stages: stages that accept generic types are verified against the concrete type flowing through the pipeline. Generic resolution happens at load time.

Pipeline type inference: for dynamically constructed pipelines, the LVM infers types at construction time. Type errors are caught before the pipeline executes.

Pipeline composition type checking: PIPE_COMPOSE creates a new pipeline by connecting two existing pipelines. The output type of the first must match the input type of the second.

Error type propagation: when a stage returns Result[T, E], the pipeline tracks both the success type T and the error type E. Error types are merged at pipeline branch join points.

Type-erased pipelines: PIPE_DYNAMIC allows pipelines with runtime type checking instead of load-time checking. Type-erased pipelines are slower but more flexible.

Pipeline type metadata: the FUNC section stores pipeline type chains for each pipeline-creating function. Type chains enable: validation, optimization, and documentation.

Pipeline subtyping: a stage accepting a base trait can receive any implementing type. The LVM checks trait implementation at load time using the type descriptor's trait list.

Pipeline type visualization: the debug tools can display pipeline type chains as flow diagrams. Each stage is annotated with its input and output types for debugging.

2.3 Module Linking

Module imports: the LINK section lists imported functions by: module name, function name, and expected type signature. The LVM resolves imports when loading dependent modules.

Module exports: the LINK section lists exported functions with: function name, function index, and type signature. Only exported functions are visible to other modules.

Lazy module loading: modules are loaded on demand when an imported function is first called. Lazy loading reduces startup time for programs with many dependencies.

Module search path: the LVM searches for modules in: the current directory, the LATERALUS_PATH environment variable, and the standard library directory.

Version constraints: module imports can specify version constraints. The LVM selects the newest compatible version. Incompatible versions produce a link error.

Circular dependency detection: the module loader detects circular dependencies and reports them as errors. Circular dependencies are resolved by restructuring modules.

Dynamic linking: modules can be loaded at runtime using the load_module() function. Dynamic linking enables: plugin systems, hot code replacement, and modular architectures.

Link-time type checking: the linker verifies that exported function types match imported function types. Type mismatches are reported with both the exporter's and importer's type signatures.

Module sealing: a sealed module cannot be replaced at runtime. Sealing enables: security-critical code protection, optimization (direct calls instead of indirect), and API stability.

Module metadata: each module stores: name, version, author, license, and dependency list. The package manager uses module metadata for: version resolution and registry publishing.

8.3 Profiling Integration

Instruction-level profiling: the LVM counts executions of each instruction. Hot instructions (executed more than 10,000 times) are candidates for JIT compilation.

Function-level profiling: the LVM tracks: call count, total execution time, self time (excluding callees), and maximum stack depth for each function.

Allocation profiling: the LVM tracks: allocation count, total bytes allocated, and allocation site (function and instruction offset) for each allocation.

Pipeline profiling: the LVM tracks per-stage: execution count, total time, input element count, output element count, and error count. Pipeline profiles identify bottleneck stages.

Profiling data format: profiling data is stored in a binary format: function entries (32 bytes each), instruction entries (16 bytes each), and allocation entries (24 bytes each).

Profiling overhead: basic profiling (function-level) adds 2% overhead. Detailed profiling (instruction-level) adds 10% overhead. Allocation profiling adds 5% overhead.

Profiling visualization: the lateralus-prof tool reads profiling data and produces: flame graphs, call graphs, allocation timelines, and pipeline stage comparisons.

Continuous profiling: the LVM supports always-on profiling at low overhead (1%). Continuous profiling enables: production performance monitoring and regression detection.

Profiling and optimization: the AOT compiler reads profiling data to guide optimization. Hot functions are optimized aggressively. Cold functions are optimized for size.

Sampling profiler integration: the LVM supports external sampling profilers (perf, Instruments) through: frame pointer metadata, symbol tables, and source maps.

10.2 Format Evolution

Bytecode format versioning follows semantic versioning. Major versions break backward compatibility. Minor versions add features. Patch versions fix bugs.

Current version: 1.0. Planned features for version 1.1: compressed code sections, extended pipeline instructions, and improved debug information.

Planned features for version 2.0: generic bytecode (no monomorphization), effect system encoding, algebraic effect handlers, and module-level concurrency.

Format extension process: proposals are submitted as RFCs. Each RFC includes: motivation, specification, backward compatibility analysis, and implementation plan.

Reference implementation: the LVM is the reference implementation. All format changes are implemented in the LVM first. Third-party implementations follow the specification.

Conformance testing: a test suite of bytecode files verifies format compliance. The test suite covers: all instruction types, all section types, and edge cases.

Format comparison: Lateralus bytecode is 30% smaller than Java bytecode and 20% larger than WebAssembly for equivalent programs. The size difference is due to pipeline instructions and type descriptors.

Interoperability: the bytecode format includes provisions for calling WebAssembly modules. A bridge

translates between Lateralus and WebAssembly calling conventions.

Long-term stability: the bytecode format is designed for decade-long stability. Core instructions (arithmetic, control flow, memory) are unlikely to change. Pipeline instructions may evolve.

Archival format: bytecode files include enough metadata to be understood without external context. A bytecode file from 2024 should be loadable by an LVM from 2034.

3.5 Ownership Instructions

MOVE_CHECK slot: verifies that the local variable at slot has not been moved. If moved, raises a use-after-move error. Debug mode only; stripped in release builds.

BORROW slot: creates an immutable reference to the value at slot. The LVM increments the borrow count. The original value cannot be moved while borrowed.

BORROW_MUT slot: creates a mutable reference to the value at slot. The LVM sets the exclusive borrow flag. No other borrows can exist while the mutable borrow is active.

UNBORROW: releases a borrow. The LVM decrements the borrow count or clears the exclusive flag. UNBORROW is inserted by the compiler at the end of the borrow's scope.

DROP slot: destroys the value at slot and frees its resources. The destructor is called if the type has one. The slot is marked as moved.

DROP_IF_ALIVE slot: conditionally drops the value if it has not been moved. Used at scope exits where the value may or may not have been moved (depends on control flow).

CLONE slot: deep-copies the value at slot. Clone creates a new independently-owned copy. The original remains unchanged. Clone is explicit in the source code.

RC_INC: increments the reference count of a reference-counted value. Used when an Rc value is cloned (shared, not deep-copied).

RC_DEC: decrements the reference count. If the count reaches zero, the value is dropped. RC_DEC is inserted at scope exits for Rc values.

WEAK_UPGRADE: attempts to upgrade a weak reference to a strong reference. Pushes Some(strong_ref) if the value is alive, or None if it has been dropped.

6.4 Inline Caching

Inline caching accelerates method dispatch in the LVM. At each CALL_METHOD site, the LVM caches the last resolved method pointer and the receiver type.

Monomorphic cache: if the same receiver type appears repeatedly (common case), the cached method pointer is used directly. The LVM checks the receiver type against the cached type.

Polymorphic cache: if 2-4 different receiver types appear, the LVM uses a small inline cache with

multiple entries. Each entry maps a type to a method pointer.

Megamorphic fallback: if more than 4 receiver types appear, the LVM falls back to full vtable lookup. Megamorphic sites are rare in practice (fewer than 1% of call sites).

Cache invalidation: caches are invalidated when: a new module is loaded (adding new trait implementations), or when hot code replacement modifies a function.

Cache statistics: the LVM tracks: cache hit rate, monomorphic rate, polymorphic rate, and megamorphic rate. Typical programs achieve 95% monomorphic hit rate.

PIC (polymorphic inline cache) layout: each cache entry is 16 bytes (8 bytes type pointer, 8 bytes method pointer). A 4-entry PIC uses 64 bytes (one cache line).

IC and AOT: the AOT compiler generates specialized code for monomorphic call sites. Polymorphic sites use guard-based dispatch. Megamorphic sites use indirect calls.

Type profiling: the LVM records receiver types at each call site. Type profiles guide: AOT compilation (specialization), JIT compilation (guard insertion), and optimization decisions.

Devirtualization: when the LVM determines that a call site is always monomorphic, it replaces CALL_METHOD with a direct CALL. Devirtualization eliminates vtable lookup overhead.

9.4 Bytecode Optimization

Peephole optimization on bytecode: the compiler applies local patterns to reduce instruction count. Common patterns: PUSH/POP elimination, LOAD/STORE fusion, and redundant branch removal.

Constant propagation in bytecode: PUSH followed by a constant-foldable instruction is replaced with the result. For example, PUSH 2; PUSH 3; ADD becomes PUSH 5.

Dead code elimination in bytecode: instructions whose results are never used (popped immediately, overwritten) are removed. Dead code elimination runs after constant propagation.

Jump threading: a conditional jump to another jump is replaced with a direct jump to the final target. Jump threading reduces branch overhead in generated code.

Branch inversion: if a conditional branch targets the next instruction (no-op branch), the condition is inverted and the branch is eliminated. Branch inversion simplifies control flow.

Block reordering: basic blocks are reordered to maximize fall-through paths. Fall-through is faster than taken branches on most architectures. Reordering uses profiling data when available.

Instruction combining: sequences like LOAD slot; LOAD slot are replaced with LOAD slot; DUP. Combining reduces code size and improves interpretation speed.

Bytecode shrinking: after optimization, the compiler renumbers instructions and updates jump offsets. Shorter jumps use 16-bit offsets. Longer jumps use 32-bit offsets.

Optimization levels: -O0 (no optimization), -O1 (peephole only), -O2 (full optimization), -Os (optimize

for size). Each level builds on the previous.

Optimization metrics: -O2 reduces bytecode size by 20% and improves interpretation speed by 15% compared to -O0. Optimization adds 30% compilation time.

2.4 Memory-Mapped Loading

The LVM supports memory-mapped file loading for fast startup. The bytecode file is mapped into memory with mmap(). Sections are accessed directly from the mapped memory.

Alignment requirements: sections are aligned to page boundaries (4 KB) when memory-mapped loading is enabled. Page-aligned sections avoid partial page reads.

Copy-on-write: the mapped memory is marked copy-on-write. Runtime modifications (patching, relocation) create private copies. Unmodified pages are shared across processes.

Demand paging: memory-mapped files use demand paging. Only accessed pages are loaded into physical memory. Large modules with rarely-used functions benefit from lazy loading.

File locking: the LVM acquires a shared lock on the bytecode file during execution. The lock prevents modification while the file is memory-mapped.

Memory-mapped constant pool: constants are accessed directly from the mapped file without copying. String constants are pointers into the mapped memory.

Preloading hints: the LVM uses madvise(MADV_WILLNEED) to hint the kernel about upcoming page accesses. Preloading reduces page fault latency during startup.

Memory-mapped vs. read: small files (under 64 KB) are read into a buffer. Large files are memory-mapped. The crossover point is configurable.

Mapped file caching: the operating system caches mapped file pages. Multiple LVM instances sharing the same module share physical memory pages.

Security considerations: memory-mapped bytecode is marked non-executable. The LVM's interpreter reads instructions from the mapped data. AOT code uses separate executable memory.

5.4 Ownership Type Encoding

Ownership qualifiers in type descriptors: each field has an ownership qualifier: OWNED (default), BORROWED (&T), MUT_BORROWED (&mut T), or SHARED (Rc[T]).

Lifetime encoding: lifetimes are represented as scope indices in the bytecode. Each scope has a unique index. Borrow constraints reference scope indices.

Drop flag encoding: types with conditional drops (values that may or may not have been moved) include drop flags in their runtime layout. Each drop flag is one byte.

Move semantics in bytecode: the MOVE instruction transfers ownership. The source is marked as

moved. The destination becomes the new owner. Move is a zero-cost operation at runtime.

Borrow encoding in bytecode: borrow instructions (BORROW, BORROW_MUT) create reference values. References carry: the address of the borrowed value and the borrow scope index.

Ownership verification at load time: the LVM optionally verifies ownership invariants during module loading. Verification checks: no use-after-move, no double-borrow, and lifetime containment.

Copy type marking: type descriptors include a COPY flag for types that implement copy semantics. Copy types are duplicated instead of moved. LOAD of a copy type produces a copy.

Pin encoding: pinned values cannot be moved after creation. The type descriptor includes a PIN flag. The LVM enforces pinning by rejecting MOVE instructions on pinned values.

Ownership and closures: closure type descriptors include capture mode (by-value, by-reference, by-mutable-reference) for each captured variable. The LVM uses capture modes for correct resource management.

Ownership debugging: in debug mode, the LVM maintains an ownership log. The log records: every move, borrow, unborrow, and drop with timestamps and source locations.

7.4 Pipeline Execution Model

Push-based execution: the default pipeline model is push-based. Producers push data through the pipeline. Each stage processes input immediately and pushes output to the next stage.

Pull-based execution: lazy pipelines use pull-based execution. Consumers request data from producers. Each stage requests input from the previous stage on demand.

Hybrid execution: the LVM can mix push and pull within a single pipeline. Buffered stages act as boundaries between push and pull segments.

Pipeline cancellation: a stage can cancel the pipeline by returning a special CANCEL value. Upstream stages are notified and stop producing. Resources are cleaned up.

Pipeline state management: stateful pipeline stages maintain state between invocations. State is stored in the pipeline context. State is initialized when the pipeline starts and cleaned up when it ends.

Pipeline error handling strategy: fail-fast (default, first error stops the pipeline), collect-errors (all errors are collected and returned), skip-errors (errors are logged and skipped).

Pipeline composition: pipelines can be composed into larger pipelines using PIPE_COMPOSE. Composed pipelines share the same execution context and error handling strategy.

Pipeline cloning: PIPE_CLONE creates an independent copy of a pipeline. The clone has its own state and execution context. Cloning is used for: parallel execution and testing.

Pipeline serialization: pipeline definitions can be serialized to bytecode for: transmission, storage,

and deferred execution. Serialization includes: stage functions and configuration.

Pipeline resource management: the pipeline context tracks allocated resources (file handles, network connections). Resources are cleaned up when the pipeline ends, even on error.

8.4 Structured Logging

The LVM generates structured log events during execution. Events include: module load, function entry/exit, garbage collection, pipeline stage, and error.

Log event format: timestamp (8 bytes), event type (2 bytes), thread ID (4 bytes), data length (2 bytes), event data (variable). Events are stored in a ring buffer.

Log filtering: events are filtered by: type (module, function, GC, pipeline, error), severity (debug, info, warning, error), and module name (glob patterns).

Log output: events can be written to: file (binary format for tools), console (human-readable format), network (JSON format for log aggregation), or memory (ring buffer for debugging).

Performance impact: logging at INFO level adds 1% overhead. DEBUG level adds 5% overhead. Logging can be disabled entirely for zero overhead.

Log analysis tools: lateralus-logview reads binary log files and provides: timeline visualization, event filtering, pipeline stage timing, and GC pause analysis.

Correlation IDs: each pipeline execution gets a unique correlation ID. All log events from the same pipeline share the ID. Correlation enables: tracing pipeline execution across threads.

Audit events: security-relevant events (module load, capability use, sandbox violation) are logged separately in a tamper-resistant audit log.

Log rotation: file-based logging supports: size-based rotation (default 100 MB), time-based rotation (daily), and compression of rotated files.

Remote logging: the LVM can stream log events to a remote server. The streaming protocol supports: TLS encryption, compression, and batching for efficiency.

10.3 Ecosystem Integration

WebAssembly interop: the LVM can load and execute WebAssembly modules alongside Lateralus bytecode. Data is exchanged through a shared linear memory or typed value passing.

C FFI: the LVM supports calling C functions through the CALL_NATIVE instruction. C functions are registered at load time. Data types are marshaled according to the C ABI.

Python embedding: the LVM can be embedded in a Python application. The Python API provides: module loading, function calling, and data conversion between Python and Lateralus types.

JavaScript bridge: for web deployments, the LVM compiles to WebAssembly. JavaScript can call

Lateralus functions and vice versa through a generated binding layer.

Database integration: the LVM provides pipeline stages for database access. SQL queries are compiled to bytecode at module load time. Query results flow through pipelines.

Network protocol support: the LVM includes pipeline stages for: HTTP client/server, WebSocket, gRPC, and custom binary protocols. Protocol stages handle serialization and deserialization.

File format support: the LVM provides pipeline stages for reading and writing: JSON, CSV, XML, TOML, and binary formats. Format stages integrate with the pipeline type system.

IDE integration: the LVM exports language server protocol (LSP) hooks for: bytecode disassembly, runtime inspection, and performance profiling.

Package registry: the Lateralus package registry distributes: source code, bytecode, and documentation. The registry supports: version pinning, dependency resolution, and security audits.

Cloud deployment: bytecode modules can be deployed to: serverless platforms (AWS Lambda compatible), container environments, and edge computing nodes (Cloudflare Workers compatible).

3.6 String Instructions

STR_CONCAT pops two strings and pushes their concatenation. The LVM allocates a new string large enough for both. Small strings (under 64 bytes) use stack allocation.

STR_SLICE pops start index, end index, and string. Pushes a substring. Slicing is O(1) as it creates a view into the original string. The original string's reference count is incremented.

STR_LEN pushes the byte length of the top-of-stack string. STR_CHAR_COUNT pushes the Unicode code point count. The two values differ for strings with multi-byte characters.

STR_EQ compares two strings for equality. Comparison is byte-by-byte. STR_CMP performs lexicographic comparison and pushes -1, 0, or 1.

STR_FIND pops a pattern and a string. Pushes the byte offset of the first occurrence, or -1 if not found. Uses Boyer-Moore algorithm for patterns longer than 4 bytes.

STR_REPLACE pops a replacement, pattern, and string. Pushes a new string with all occurrences of pattern replaced. The replacement is allocated on the heap.

STR_SPLIT pops a delimiter and string. Pushes an array of substrings. Substrings are views into the original string for zero-copy splitting.

STR_JOIN pops a separator and an array of strings. Pushes the concatenation of all strings with the separator between them. Pre-computes total length to allocate once.

STR_TO_INT and STR_TO_FLOAT parse numeric strings. Invalid formats push an Err result. Parsing handles: decimal, hexadecimal (0x prefix), binary (0b), and scientific notation (1e10).

STR_FORMAT implements template strings. Format specifiers include: {0} positional, {name}

named, {:d} integer, {:f} float, {:x} hexadecimal, {:b} binary, {:s} string.

STR_INTERN interns a string in the global string table. Interned strings are compared by pointer equality (O(1)) instead of byte-by-byte comparison (O(n)).

STR_ENCODE_UTF8 and STR_DECODE_UTF8 convert between Lateralus strings and byte arrays. Encoding validates UTF-8. Decoding replaces invalid sequences with the replacement character.

6.5 Garbage Collection Integration

The LVM uses a generational garbage collector with three generations: young (nursery), old, and permanent. Young objects are promoted to old after surviving two collections.

GC root scanning: roots include: local variables on the call stack, global variables, and pipeline context values. The compiler generates stack maps that identify root locations.

Stack maps: each function's entry in the FUNC section includes a stack map. The map identifies which local slots contain object references at each GC safe point.

GC safe points: safe points are inserted at: function calls, loop back edges, allocation sites, and pipeline stage boundaries. The LVM checks for pending GC at each safe point.

Write barriers: stores to old-generation objects check whether the stored value is a young-generation object. If so, the old object is added to the remembered set for young GC.

Young generation collection: copies live young objects to a survivor space. Dead objects are reclaimed. Collection time is proportional to the number of live objects (not dead).

Old generation collection: uses mark-and-sweep. Marking traverses all reachable objects. Sweeping reclaims unmarked objects. Compaction is optional (triggered when fragmentation exceeds 30%).

Permanent generation: constants, interned strings, and type descriptors are allocated in the permanent generation. Permanent objects are never collected.

GC pause minimization: young GC pauses average 0.5 ms. Old GC pauses average 5 ms. Incremental marking reduces old GC pauses to 1 ms at the cost of 5% throughput.

GC tuning: configurable parameters include: young generation size (default 4 MB), old generation size (default 64 MB), promotion threshold (default 2 collections), and compaction threshold (default 30% fragmentation).

Finalization: objects with destructors are queued for finalization after GC. Finalizers run on a dedicated thread. Finalizer execution order is undefined.

GC statistics: the LVM reports: collection count, total pause time, promotion rate, allocation rate, and heap occupancy. Statistics are accessible via the profiling API.

References

[1] Lindholm, T. et al. The Java Virtual Machine Specification. Oracle, 2023.

[2] ECMA-335. Common Language Infrastructure. ECMA International, 2012.

[3] WebAssembly Specification. W3C, 2023.

[4] Ierusalimschy, R. et al. The Implementation of Lua 5.0. JUCS, 2005.

[5] Nystrom, R. Crafting Interpreters, Chapter 14: Chunks of Bytecode. 2021.