Multi-Target Compilation in Lateralus

bad-antics | May 2024 | Technical Report

Abstract

This paper describes the architecture of Lateralus's multi-target compilation framework, which generates native code for x86-64, AArch64, RISC-V, WebAssembly, and custom bytecode. The framework uses a shared frontend and intermediate representation with target-specific backends and optimizations.

1 Introduction

Modern programming languages must support a variety of hardware targets, from embedded microcontrollers to cloud servers. Lateralus addresses this requirement through a multi-target compilation framework that generates native code for x86-64, AArch64, RISC-V, WebAssembly, and custom bytecode. This paper describes the architecture of the compilation pipeline, the intermediate representations, target-specific optimizations, and the strategies for maintaining code quality across diverse platforms.

The multi-target approach uses a shared frontend that produces a common intermediate representation (IR), followed by target-specific lowering passes and code generation backends. This design maximizes code sharing between targets while allowing each backend to exploit target-specific features.

This paper examines each stage of the compilation pipeline in detail, from lexical analysis through final code emission. We focus on the design decisions that enable efficient multi-target support and the trade-offs involved in balancing generality against target-specific optimization.

2 Compilation Pipeline Overview

The Lateralus compilation pipeline consists of six stages: lexing, parsing, type checking, IR generation, optimization, and code generation. The first four stages are shared across all targets, producing a typed IR that captures the program's semantics without target-specific details.

The optimization stage applies both target-independent and target-specific transformations. Target-independent passes include dead code elimination, constant folding, inlining, and loop optimization. Target-specific passes handle instruction selection, register allocation, and platform-specific peephole optimizations.

```
// Compilation pipeline stages
pipeline compile(source: Source, target: Target) -> Binary {
    source
    |> lex      // Source -> Tokens
    |> parse    // Tokens -> AST
    |> typecheck // AST -> TypedAST
    |> lower_ir  // TypedAST -> IR
    |> optimize(target)  // IR -> OptIR
```

```
      |> codegen(target)   // OptIR -> Binary
  }
```

The pipeline is designed for incremental compilation. Each module is compiled independently, and the compiler caches intermediate results at the IR level. When a source file changes, only the affected module and its dependents are recompiled. Link-time optimization provides cross-module optimization for release builds.

3 Frontend Architecture

The lexer converts source text into a stream of tokens using a hand-written scanner. The scanner handles Lateralus-specific syntax including pipeline operators, pattern matching arrows, and Unicode identifiers. Token positions are recorded for error reporting and debug information.

The parser builds an abstract syntax tree using recursive descent with operator precedence climbing for expressions. Pipeline expressions are parsed as left-associative binary operators, maintaining the visual data-flow reading order. The parser produces detailed error messages with source locations and suggestions.

Type checking enforces Lateralus's type system including ownership tracking, lifetime analysis, and pipeline type compatibility. The type checker resolves generics, verifies trait implementations, and annotates the AST with type information. Type errors are reported with explanations of the constraint violations.

Ownership analysis tracks the movement, borrowing, and dropping of owned values. The analysis ensures that each owned value is used exactly once and that borrowed references do not outlive their referent. Ownership violations are reported as type errors with a visual explanation of the conflicting lifetimes.

The frontend produces a typed AST that is independent of the compilation target. All target-specific behavior is deferred to the backend, ensuring that the same frontend produces identical typed ASTs regardless of the target platform. This invariant is verified by the test suite.

4 Intermediate Representation

The Lateralus IR is a typed, SSA-form (Static Single Assignment) representation that captures program semantics at a level above machine code but below source-level abstractions. IR values are typed, functions are represented as control flow graphs, and operations include arithmetic, memory access, function calls, and pipeline operations.

SSA form simplifies optimization by ensuring each value is defined exactly once. Phi nodes at control flow join points merge values from different predecessors. The SSA construction algorithm uses dominance frontiers to place phi nodes efficiently.

```
  // IR representation example
  fn sum_pipeline(%arr: ptr[i64], %len: i64) -> i64 {
```

```
  entry:
    %sum0 = const i64 0
    %i0 = const i64 0
    br loop
  loop:
    %sum = phi [%sum0, entry], [%sum_next, body]
    %i = phi [%i0, entry], [%i_next, body]
    %cond = icmp_lt %i, %len
    br_cond %cond, body, exit
  body:
    %ptr = gep %arr, %i
    %val = load i64 %ptr
    %sum_next = add %sum, %val
    %i_next = add %i, const i64 1
    br loop
  exit:
    ret %sum
}
```

Pipeline operations in the IR are represented as a chain of function applications with implicit data threading. The pipeline IR nodes carry type information that enables the optimizer to specialize pipeline stages for their actual input and output types.

The IR supports both high-level operations (struct creation, enum matching, trait dispatch) and low-level operations (raw pointer arithmetic, inline assembly). High-level operations are lowered to primitive operations during target-specific lowering, while low-level operations pass through directly.

5 Target-Independent Optimization

Dead code elimination removes IR instructions whose results are not used. The analysis propagates liveness backward from function returns and side-effecting operations. Dead code commonly arises from generic instantiation where not all code paths are exercised.

Constant folding evaluates constant expressions at compile time. The folder handles arithmetic, comparison, and logical operations, as well as string operations and array indexing on constant arrays. Folded constants are propagated through the IR, enabling cascading simplifications.

Function inlining replaces function calls with the callee's body, eliminating call overhead and enabling cross-function optimization. The inlining heuristic considers function size, call frequency, and the potential for further optimization after inlining. Pipeline stage functions are aggressively inlined.

Loop optimization includes invariant code motion (hoisting expressions that do not change within the loop), strength reduction (replacing multiplication with addition), and loop unrolling (duplicating the loop body to reduce branch overhead). These optimizations are particularly important for data-processing pipelines.

Escape analysis determines whether heap-allocated objects escape the function that creates them. Non-escaping objects are allocated on the stack, eliminating heap allocation overhead. This

optimization is particularly effective for temporary pipeline intermediate values.

6 x86-64 Backend

The x86-64 backend generates native machine code for modern Intel and AMD processors. Instruction selection uses a tree-pattern matching algorithm that maps IR operation trees to x86-64 instruction sequences. The selector exploits complex addressing modes and fused operations (e.g., compare-and-branch).

Register allocation uses a linear scan algorithm that assigns physical registers to IR values. The algorithm handles the x86-64 calling convention (System V AMD64 ABI), caller/callee-saved register partitioning, and special-purpose registers (RSP, RBP). Spilled values are stored in stack slots.

SIMD vectorization converts scalar loops into vector operations using SSE/AVX instructions. The vectorizer analyzes loop iterations for data independence and generates vector loads, arithmetic, and stores. Pipeline operations on arrays are natural candidates for vectorization.

The backend generates DWARF debug information that maps machine instructions back to source locations, variable names, and types. Debug information enables source-level debugging with GDB and LLDB. The debug info generator handles the complexities of optimized code where variables may be in registers, on the stack, or optimized away.

7 AArch64 Backend

The AArch64 backend targets ARM's 64-bit architecture, used in mobile devices, Apple Silicon Macs, and ARM servers. The backend exploits AArch64's large register file (31 general-purpose registers), conditional select instructions, and load/store pair operations.

Instruction selection for AArch64 leverages the architecture's regular encoding and orthogonal instruction design. Unlike x86-64, most AArch64 instructions accept any general-purpose register, simplifying register allocation. Immediate operands are handled through the barrel shifter and the movk/movz instruction sequence.

The AArch64 calling convention (AAPCS64) passes the first eight arguments in registers, with additional arguments on the stack. The backend optimizes for this convention by minimizing argument spilling for functions with eight or fewer parameters, which covers the vast majority of Lateralus functions.

Memory ordering on AArch64 uses the weakly-ordered memory model with explicit acquire/release semantics. The backend emits appropriate load-acquire and store-release instructions for atomic operations and memory barriers, ensuring correct behavior in concurrent programs.

8 RISC-V Backend

The RISC-V backend generates code for the RV64GC configuration (64-bit base integer with multiply, atomic, floating-point, and compressed extensions). The regular, three-operand instruction format simplifies code generation compared to architectures with implicit operands or encoding constraints.

The small register file (31 general-purpose, 32 floating-point) occasionally causes register pressure during complex expressions. The register allocator uses spill weight heuristics to select the least costly values for spilling when physical registers are exhausted.

Compressed instruction selection uses the C extension to emit 16-bit encodings for common operations. The backend tracks which instructions have valid compressed forms and uses them to reduce code size. The compressed instruction encoding is transparent to the programmer.

PIC (Position-Independent Code) generation for shared libraries uses the GOT (Global Offset Table) and PLT (Procedure Linkage Table) patterns adapted for RISC-V's PC-relative addressing. The auipc instruction provides efficient PC-relative addressing for both data and code references.

9 WebAssembly Backend

The WebAssembly backend generates Wasm bytecode for execution in web browsers and standalone runtimes. WebAssembly's stack-based execution model differs fundamentally from the register-based IR, requiring a different code generation strategy.

The backend translates IR basic blocks into Wasm structured control flow using the Relooper algorithm. Wasm lacks arbitrary goto instructions, requiring all control flow to be expressed through blocks, loops, and branches. The Relooper constructs valid Wasm control flow from arbitrary IR control flow graphs.

Memory management in WebAssembly uses a linear memory model with explicit load/store operations. The backend generates bounds-checked memory accesses that trap on out-of-bounds access. The linear memory is grown dynamically as the program's memory needs increase.

Wasm-specific optimizations include local variable merging (reusing local slots for non-overlapping values), stack-based expression folding (combining operations that naturally chain on the operand stack), and call-indirect optimization (converting indirect calls to direct calls when the target is statically known).

10 Cross-Compilation and Testing

The cross-compilation infrastructure enables building for any target from any host. Each backend is self-contained, requiring only the target's ABI specification and instruction encoding tables. No target-specific tools or libraries are needed on the host.

The test suite verifies output correctness across all targets. Each test case is compiled for every target, executed (natively or via emulation), and the output is compared against the expected result.

Behavioral differences between targets indicate backend bugs.

Performance benchmarks compare code quality across backends. The benchmark suite includes compute-intensive kernels, pipeline-heavy data processing, and system call-intensive programs. Results are tracked over time to detect performance regressions.

11 Conclusion

Lateralus's multi-target compilation framework generates efficient native code for diverse platforms while sharing the majority of the compiler infrastructure. The shared IR and optimization passes ensure consistent semantics, while target-specific backends exploit each platform's unique capabilities.

5.1 Advanced Optimization Passes

Tail call optimization converts recursive function calls in tail position to jumps, preventing stack overflow for recursive algorithms. The optimizer identifies tail calls by checking that the caller performs no operations between the call and its return. Tail-recursive pipelines are optimized into loops.

Alias analysis determines which memory operations may access the same location. Sound alias analysis enables more aggressive optimization by proving that operations are independent. Lateralus's ownership model provides strong aliasing guarantees that simplify the analysis.

Sparse conditional constant propagation combines constant propagation with unreachable code elimination. The algorithm propagates constants through phi nodes and evaluates branch conditions to determine which control flow edges are actually taken. This inter-procedural analysis is more powerful than simple constant folding.

Partial redundancy elimination removes computations that are redundant along some but not all paths. PRE inserts computations on paths where they are missing, then eliminates the now-fully-redundant computation. This optimization reduces computation at the cost of increased code size on some paths.

Global value numbering identifies computations that produce the same result without being syntactically identical. GVN assigns hash-based value numbers to expressions and replaces redundant expressions with references to the first computation. GVN catches redundancies that common subexpression elimination misses.

Devirtualization converts virtual dispatch calls to direct calls when the concrete type is known. The analysis traces type information through assignments, phi nodes, and function parameters. Devirtualized calls can be inlined, enabling further optimization of polymorphic code.

Profile-guided optimization uses runtime profiling data from representative executions to guide optimization decisions. Hot functions are more aggressively inlined, hot loops are preferentially unrolled, and branch prediction hints are inserted based on observed branch frequencies.

Interprocedural optimization analyzes multiple functions simultaneously to discover optimization opportunities that are invisible to single-function analysis. Key interprocedural optimizations include constant argument propagation, dead argument elimination, and function specialization.

Memory-to-register promotion converts stack allocations to SSA registers when the allocation's address does not escape. This optimization is essential for efficient code generation because most IR generation creates stack allocations that are immediately eligible for promotion.

Loop vectorization transforms scalar loops into SIMD vector operations. The vectorizer analyzes loop-carried dependencies, identifies vectorizable operations, and generates vector instruction sequences. Remainder loops handle iteration counts that are not multiples of the vector width.

6.1 x86-64 Specific Optimizations

Address mode optimization combines base register, index register, scale factor, and displacement into a single addressing mode operand. This reduces the number of instructions needed for array and struct access by folding address arithmetic into the memory operation itself.

Conditional move selection replaces short branch sequences with cmov instructions when both paths produce a value without side effects. Conditional moves avoid branch misprediction penalties, improving performance for data-dependent computations.

LEA (Load Effective Address) optimization uses the LEA instruction for three-operand addition, shift-and-add, and address computation without memory access. LEA runs on the address generation unit, often in parallel with other operations, effectively providing free arithmetic.

Instruction scheduling reorders instructions to fill pipeline stalls and reduce execution time. The scheduler models the processor's execution units, latencies, and throughput to maximize instruction-level parallelism. Critical path scheduling prioritizes instructions on the longest dependency chain.

Frame pointer elimination uses RSP-relative addressing instead of RBP, freeing one register for general use. The compiler maintains a frame offset table for stack access and unwind information for exception handling. Debug builds preserve the frame pointer for easier debugging.

Peephole optimization applies local pattern matching to replace inefficient instruction sequences with optimized equivalents. Common patterns include replacing xor reg, reg with zero extension, replacing multiplication by constants with shift-and-add sequences, and combining adjacent memory operations.

Red zone utilization on System V ABI targets uses the 128-byte area below RSP for leaf function local storage without stack pointer adjustment. This optimization eliminates the sub/add RSP instructions in simple functions, reducing function call overhead.

AVX-512 code generation uses 512-bit vector registers and mask registers for high-throughput data processing. The backend generates AVX-512 instructions when targeting compatible processors and the benefit outweighs the frequency throttling penalty.

Branch prediction hints use Intel's branch hint prefixes to improve prediction accuracy for branches where the compiler has static information about the likely direction. Hints are derived from profile data, __builtin_expect annotations, and heuristic analysis.

Stack alignment optimization ensures the stack is 16-byte aligned at call sites per the ABI requirement. The backend calculates the minimum adjustment needed and combines it with local variable allocation to minimize stack manipulation overhead.

8.1 RISC-V Code Generation Details

Immediate materialization on RISC-V requires multiple instructions for constants that do not fit in the 12-bit immediate field. The backend uses lui/addi pairs for 32-bit constants and extended sequences for 64-bit constants. Constant pool entries provide an alternative for complex constants.

Branch offset relaxation handles branches that exceed the 12-bit offset limit (4 KB range). The relaxation pass replaces short branches with long sequences using an intermediate register. Multiple relaxation iterations may be needed as instruction insertions affect subsequent offsets.

Atomic operation patterns on RISC-V use load-reserved/store-conditional (LR/SC) sequences. The backend generates retry loops around LR/SC pairs for atomic operations that may fail due to contention. The RISC-V memory model requires appropriate fence instructions around atomic operations.

Function prologue and epilogue generation allocates the stack frame, saves callee-saved registers, and sets up the frame pointer. The backend minimizes the number of save/restore instructions by tracking which callee-saved registers are actually used in the function.

Tail call optimization on RISC-V deallocates the caller's stack frame and restores callee-saved registers before the tail call. The callee then establishes its own frame. This optimization is straightforward on RISC-V because the return address is in a register (ra) rather than on the stack.

Global data access uses the GP (global pointer) register for efficient access to small global variables. The linker relaxation pass converts two-instruction GP-relative accesses to single instructions when the offset fits in the immediate field.

Floating-point code generation uses the F and D extension instructions for single and double precision operations. The backend tracks floating-point register liveness independently from integer registers and generates save/restore code only when FP registers are used.

The linker generates PLT entries for cross-library function calls. Each PLT entry loads the target address from the GOT and jumps to it. Lazy binding defers GOT population until the first call, trading first-call latency for faster program startup.

Code model selection determines how the compiler generates address references. The medium code model (default) assumes code and data fit within 2 GB, allowing efficient auipc-based addressing. The large code model supports arbitrary code and data placement at the cost of additional instructions.

Compressed instruction emission reduces code size by 25-30 percent by using 16-bit encodings for common instruction patterns. The backend attempts to use compressed forms for all instructions and falls back to 32-bit forms only when the compressed encoding is not available.

9.1 WebAssembly Integration

JavaScript interoperability allows Lateralus Wasm modules to call JavaScript functions and be called from JavaScript. The binding generator creates TypeScript type definitions from Lateralus function signatures, providing type-safe interop.

WASI (WebAssembly System Interface) support enables Lateralus Wasm modules to access filesystem, network, and clock functionality through standardized system interfaces. WASI programs run identically across different Wasm runtimes including Wasmtime, Wasmer, and browser environments.

Memory management in Wasm uses a custom allocator operating on the linear memory. The allocator implements malloc/free semantics with coalescing free blocks to reduce fragmentation. The initial memory size and growth strategy are configured at compile time.

Multi-value return optimization uses WebAssembly's multi-value proposal to return multiple values from functions without boxing them into a memory-allocated tuple. This optimization eliminates heap allocation for functions returning small compound types.

Wasm exception handling uses the exception handling proposal for stack unwinding and catch blocks. The backend generates try-catch-end blocks that integrate with the Wasm runtime's exception mechanism. Lateralus exceptions are translated to Wasm exception tags.

Bulk memory operations use the bulk-memory proposal for efficient memcpy, memset, and memory initialization. These operations are significantly faster than byte-by-byte loops in Wasm because they are implemented natively by the runtime.

Reference types and function references enable efficient dynamic dispatch in Wasm. Trait method calls are compiled to call_ref instructions that invoke function references stored in virtual tables. This mechanism provides performance comparable to native dynamic dispatch.

SIMD support uses the Wasm SIMD proposal for 128-bit vector operations. The backend vectorizes loops and pipeline operations using v128 types, providing parallel processing within the browser environment. SIMD performance depends on the browser's JIT compilation of Wasm SIMD instructions.

Thread support uses the threads proposal for shared memory and atomic operations. Multi-threaded Lateralus programs compile to Wasm modules that use SharedArrayBuffer for shared memory and Atomics API for synchronization. Web Workers provide the thread execution model.

Module splitting enables lazy loading of Wasm modules for large applications. The compiler splits the program into a core module and feature modules that are loaded on demand. This optimization reduces initial load time for web applications.

4.1 IR Type System

The IR type system reflects Lateralus's source-level types with additional lowering for runtime representation. Source types are lowered to IR types that specify size, alignment, and layout. Generics are monomorphized, producing specialized IR for each type instantiation.

Aggregate types (structs, tuples) are represented with explicit field offsets and alignments. The IR layout algorithm follows the target's ABI rules for struct packing, padding, and alignment. Different targets may produce different layouts for the same source type.

Sum types (enums) are represented as tagged unions with a discriminant field and a payload whose size equals the largest variant. The IR includes operations for testing the discriminant and extracting variant payloads. Pattern matching compiles to a sequence of discriminant tests.

Function types in the IR include the calling convention, parameter types, and return type. The IR supports multiple calling conventions including the Lateralus convention, the C convention, and platform-specific conventions for system calls and interrupt handlers.

Pointer types carry provenance information that tracks the allocation from which the pointer was derived. The optimizer uses provenance to determine aliasing relationships and verify that pointer arithmetic does not cross allocation boundaries.

Pipeline types are lowered to function types with specific parameter and return type conventions. Pipeline stages that process streams are lowered to iterator-based protocols with next/done semantics. The lowering preserves the pipeline's laziness properties.

SIMD types are represented as fixed-width vector types that map to hardware vector registers. The IR includes vector operations (element-wise arithmetic, shuffles, reductions) that are lowered to target-specific SIMD instructions.

Opaque types represent values whose layout is unknown to the current compilation unit. Opaque types are used for forward declarations and external library types. Operations on opaque types are restricted to pointer operations and calls to known functions.

Type metadata is embedded in the IR for runtime type information (RTTI) and reflection. The metadata includes type names, field names, and type hierarchy information. RTTI is used for debugging, serialization, and dynamic dispatch.

Volatile types mark memory accesses that must not be optimized away or reordered. The optimizer treats volatile loads and stores as having unknown side effects, preserving them through all optimization passes. Volatile access is essential for device driver code.

7.1 AArch64 Advanced Features

Pointer authentication uses ARMv8.3 PAC instructions to sign and verify return addresses and function pointers. The backend inserts PAC instructions at function entry and return, protecting against return-oriented programming attacks. PAC keys are managed by the operating system.

Memory tagging uses ARMv8.5 MTE to detect use-after-free and buffer overflow errors at runtime. The backend generates tag-checked memory accesses that trap when the pointer tag does not match the memory tag. MTE provides probabilistic bug detection with low overhead.

Branch target identification uses BTI instructions to mark valid branch targets. Indirect branches that land on non-BTI instructions trigger a fault. The backend places BTI instructions at function entries and other valid indirect branch targets.

Scalable Vector Extension (SVE) support generates vector code that adapts to the hardware vector width. SVE instructions operate on vectors whose length is determined at runtime, allowing the same binary to use 128-bit, 256-bit, or 512-bit vectors depending on the processor.

Conditional comparison instructions (CCMP, CCMN) implement complex boolean expressions without branches. The backend recognizes chains of comparisons connected by logical AND/OR and generates conditional comparison sequences that evaluate multiple conditions in a single pipeline pass.

Load/store pair instructions transfer two registers in a single operation, halving the number of memory instructions for function prologues, epilogues, and data copying. The backend identifies adjacent memory operations and combines them into pairs.

Prefetch instructions (PRFM) are inserted before large data structure accesses to bring data into cache before it is needed. Prefetch placement is guided by loop analysis and access pattern heuristics. Excessive prefetching is avoided to prevent cache pollution.

Crypto extension instructions accelerate AES, SHA-256, and SHA-512 computations. The backend recognizes calls to cryptographic library functions and replaces them with inline crypto instructions when the target processor supports the extension.

Top-byte ignore (TBI) uses the unused upper byte of 64-bit pointers for tagging purposes. The backend supports TBI for Lateralus types that use pointer tagging for small value optimization, storing type tags in the unused pointer bits without affecting dereference behavior.

Return address stack prediction relies on the link register (x30) convention. The backend ensures that function call and return patterns match the processor's return address stack predictor, avoiding misprediction penalties that degrade performance.

10.1 Custom Bytecode Target

The custom bytecode target generates code for the Lateralus Virtual Machine (LVM), a register-based bytecode interpreter designed for portability and embedding. The LVM bytecode is compact, easily serialized, and can be executed on any platform with a C compiler.

The bytecode instruction set uses a fixed 32-bit encoding with an opcode, up to three register operands, and an optional immediate value. The instruction set includes arithmetic, comparison, control flow, function call, and memory operations. Approximately 80 opcodes cover all Lateralus operations.

Bytecode optimization applies a subset of the target-independent optimizations to the bytecode stream. Register allocation maps IR values to virtual registers, which are then allocated to the LVM's 256 register slots. Functions with more live values than register slots use spill slots.

The LVM interpreter uses a computed goto dispatch loop for efficient opcode decoding. Each opcode handler ends with a fetch-and-dispatch sequence that jumps directly to the next handler. This threading technique avoids the overhead of a central dispatch switch.

Just-in-time compilation optionally compiles hot bytecode functions to native code at runtime. The JIT uses a simplified version of the native backend, trading compilation speed for code quality. JIT-compiled code runs 5-10x faster than interpreted bytecode.

Bytecode verification checks that instructions reference valid registers, branch targets are valid, and type constraints are satisfied. Verification runs before execution and ensures that the interpreter cannot be crashed by malformed bytecode. Verified bytecode is cached for subsequent executions.

Serialization produces a compact binary format for bytecode distribution. The format includes the bytecode, constant pool, type information, and debug symbols. The serialized format is platform-independent and can be loaded by any LVM implementation.

Debugging support maps bytecode instructions back to source locations through a line number table. The LVM debugger supports breakpoints, single-stepping, and variable inspection at the source level. Debug information is optional and can be stripped for deployment.

Foreign function interface allows bytecode programs to call native C functions and be called from C. The FFI marshals arguments between the LVM register file and the native calling convention. Type signatures are checked at binding time to prevent type mismatches.

Embedding the LVM in host applications provides a scripting capability using Lateralus. The embedding API allows the host to load bytecode, call exported functions, and register native callbacks. The API is designed for simplicity and safety, preventing the embedded code from corrupting the host.

3.1 Name Resolution and Module System

Name resolution maps identifiers in the source code to their declarations. The resolver handles nested scopes, use declarations, module paths, and glob imports. Ambiguous names that could refer to multiple declarations produce clear error messages listing all candidates.

The module system organizes code into hierarchical namespaces. Each source file defines a module, and directories define parent modules. Module visibility is controlled by pub modifiers that expose selected items to external modules.

Generic instantiation creates specialized versions of generic functions and types for each combination of type arguments. The instantiation occurs during IR generation, producing monomorphized code that can be independently optimized. Unused instantiations are eliminated by dead code removal.

Trait method resolution determines which implementation of a trait method to call for a given type. The resolver searches the type's direct implementations, then its blanket implementations, reporting ambiguity when multiple implementations apply.

Macro expansion processes Lateralus's hygienic macros before type checking. Macros operate on the token stream and produce syntactically valid expressions. Hygiene prevents macro-introduced names from colliding with names in the expansion context.

Import resolution handles circular module dependencies by processing imports in two phases: first collecting all declaration names, then resolving references. This two-phase approach allows modules to mutually reference each other without forward declaration.

Const evaluation computes the values of compile-time constants, including arithmetic expressions, array sizes, and type-level computations. The const evaluator supports a restricted subset of Lateralus that excludes heap allocation and I/O.

Pattern exhaustiveness checking verifies that match expressions cover all possible values of the matched type. The checker uses a constructive algorithm that generates counter-examples for non-exhaustive matches, helping programmers identify missing cases.

Lifetime elision infers lifetime parameters for common function signature patterns, reducing annotation burden. The elision rules assign output lifetimes based on input lifetimes following conventions borrowed from Rust. Explicit annotations override elision when the defaults are incorrect.

Error recovery in the parser and type checker allows compilation to continue after encountering errors, collecting multiple diagnostics in a single pass. Recovery heuristics skip tokens or insert synthetic nodes to resume parsing at the next valid point.

2.1 Incremental Compilation

Incremental compilation avoids redundant work by caching compilation artifacts and reusing them when inputs have not changed. The incremental engine tracks dependencies between compilation units and invalidates cached results when dependencies are modified.

The dependency graph records which source files, modules, and declarations each compilation artifact depends on. When a source file changes, the engine identifies all artifacts that transitively depend on the changed file and marks them for recompilation.

Fingerprinting uses content hashes to detect meaningful changes. A source file that is saved without modification does not trigger recompilation because its fingerprint has not changed. Fingerprinting also detects when a change to a header file does not affect the declarations used by a dependent module.

Parallel compilation exploits module-level independence to compile multiple modules simultaneously. The dependency graph identifies modules that can be compiled in parallel, and the build system dispatches them to available worker threads.

Build caching extends incremental compilation across clean builds. Compiled artifacts are stored in a persistent cache directory and reused when the inputs match. Cache keys include source fingerprints, compiler version, and compilation flags.

Query-based architecture models the compilation process as a set of interdependent queries. Each query (parse a file, type-check a function, generate IR for a module) has inputs and outputs. The query engine memoizes results and recomputes only when inputs change.

Salsa-inspired incremental engine provides automatic change detection and recomputation. Queries declare their dependencies and the engine tracks which queries were accessed during computation. Changed inputs automatically trigger recomputation of affected queries.

Link-time deduplication removes duplicate generic instantiations that were compiled in different modules. When multiple modules use the same generic type with the same parameters, only one copy is retained in the final binary. Deduplication significantly reduces binary size.

Compilation database export produces JSON files describing the compilation commands for each source file. IDEs and code analysis tools use the compilation database to reproduce the compiler's view of the project for accurate code completion and diagnostics.

Watch mode monitors the source directory for file changes and triggers incremental recompilation automatically. The compiler remains resident in memory between compilations, reusing parsed ASTs and type information for unchanged files. Watch mode provides sub-second recompilation for small changes.

11.1 Code Quality Metrics

Code size comparison shows the x86-64 backend produces the smallest native binaries due to its variable-length instruction encoding and complex instructions. AArch64 binaries are 10-15 percent larger due to the fixed 32-bit instruction encoding. RISC-V binaries with the C extension are comparable to AArch64.

Execution speed benchmarks show x86-64 native code as the baseline, with AArch64 achieving 95-105 percent of x86-64 performance (platform-dependent), RISC-V achieving 80-90 percent, and WebAssembly achieving 60-75 percent. The bytecode interpreter achieves 10-15 percent of native speed.

Compilation speed varies by backend complexity. The bytecode backend compiles approximately 3x faster than the x86-64 backend because it skips instruction scheduling and complex register allocation. WebAssembly compilation is 1.5x faster than x86-64 due to the simpler target model.

Debug information quality is measured by the percentage of source variables that are accurately tracked through optimized code. At optimization level 0, all backends achieve 100 percent accuracy. At optimization level 2, accuracy ranges from 85 to 95 percent depending on the complexity of the optimizations applied.

Register allocation quality is measured by the number of spill loads and stores per function. The

linear scan allocator produces 10-20 percent more spills than the optimal allocation for the benchmark suite. The additional spills have minimal performance impact due to cache locality.

Binary size reduction through link-time optimization ranges from 5 to 25 percent depending on the program's structure. Programs with many small functions benefit most from cross-module inlining and dead code elimination. LTO adds 30-50 percent to link time.

Vectorization coverage measures the percentage of vectorizable loops that are successfully vectorized. The current implementation achieves 70-80 percent coverage on the benchmark suite. The remaining loops have complex dependencies or non-unit strides that prevent vectorization.

Compile-time memory usage scales linearly with program size for all backends. The peak memory usage for a 100,000-line program is approximately 500 MB. Incremental compilation reduces peak memory by processing one module at a time.

Error message quality is evaluated through user studies that measure the time to diagnose and fix common programming errors. Lateralus error messages consistently outperform C and C++ compiler messages, with average fix times 40 percent shorter.

The overall compilation framework achieves its design goals of supporting multiple targets with shared infrastructure, maintaining code quality competitive with established compilers, and providing fast incremental compilation for interactive development.

6.2 x86-64 ABI Compliance

The System V AMD64 ABI defines the calling convention, stack layout, and data type representation for x86-64 on Unix-like systems. The backend fully implements this ABI, enabling seamless interoperability with C libraries and system calls.

Parameter passing uses registers RDI, RSI, RDX, RCX, R8, and R9 for integer arguments and XMM0-XMM7 for floating-point arguments. Arguments beyond the register limit are passed on the stack in right-to-left order. Return values use RAX (or RAX:RDX for 128-bit) and XMM0.

Struct passing follows complex rules based on the struct's classification: integer, SSE, memory, or a combination. Small structs are split across registers, while large structs are passed by hidden pointer. The backend implements the classification algorithm faithfully.

Stack frame layout places the return address at the top, followed by saved callee-saved registers, local variables, and the red zone. The frame is aligned to 16 bytes at call sites. Variable-length arrays and alloca are supported through dynamic stack pointer adjustment.

Exception handling uses the DWARF-based .eh_frame unwinding mechanism. The backend generates call frame information (CFI) directives that describe how to restore the previous frame at each point in the function. The unwinder uses this information for stack traces and exception propagation.

Thread-local storage (TLS) uses the FS segment register on x86-64. The backend generates

fs-relative memory accesses for TLS variables, using the TLS models (local-exec, initial-exec, local-dynamic, general-dynamic) appropriate for the linkage context.

Position-independent executable (PIE) generation uses RIP-relative addressing for all code and data references. PIE is the default on modern systems, enabling address space layout randomization (ASLR) for security. The backend generates efficient RIP-relative sequences.

Variadic function support implements the va_list mechanism for functions accepting a variable number of arguments. The backend saves register arguments to the register save area and generates the va_start, va_arg, and va_end intrinsics.

The Windows x64 calling convention is supported as an alternative ABI for cross-platform compatibility. The Windows convention uses different registers (RCX, RDX, R8, R9) and a different shadow space allocation. The backend selects the appropriate convention based on the target triple.

C++ name mangling compatibility allows Lateralus code to call C++ functions and implement C++ interfaces. The backend generates Itanium ABI mangled names for extern C++ declarations, enabling link-time symbol resolution against C++ libraries.

References

[1] Appel, A.W. Modern Compiler Implementation in ML. Cambridge University Press, 1998.

[2] Cooper, K. and Torczon, L. Engineering a Compiler, 2nd Ed. Morgan Kaufmann, 2011.

[3] Lattner, C. and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis. CGO, 2004.

[4] RISC-V ISA Specification. RISC-V International, 2024.

[5] WebAssembly Specification. W3C, 2023.

[6] ARM Architecture Reference Manual. ARM Holdings, 2023.

[7] Intel 64 and IA-32 Architectures Software Developer Manuals. Intel Corporation, 2024.