LBC Format Deep Dive

Instruction encoding details, optimizer hints, and the debug information schema

Lateralus Language

bad-antics · May 2025 · Lateralus Language Research

ABSTRACT This paper is an extended reference for the Lateralus Bytecode (LBC) format that covers aspects not addressed in the introductory LBC paper: optimizer hint annotations embedded in the bytecode, the type annotation section used by the JIT compiler, the debug information schema, and the binary patch format for hot-swap updates. It assumes familiarity with the LBC fundamentals document and is intended as a reference for compiler and runtime implementers.

1. Optimizer Hint Annotations

LBC functions carry optimizer hint annotations in a dedicated section that precedes the instruction stream. The JIT compiler reads these hints before compiling a function. Hints are not required; their absence causes the JIT to use conservative defaults.

```
// Hint section layout (per function)
hint_count: u16
hints: [HintRecord; hint_count]

HintRecord {
    hint_type: u8,
    instr_offset: u16,  // instruction this hint applies to
    value: u32,         // hint-type-specific value
}
```

Defined hint types:

```
0x01  LIKELY_HOT       instr is on the hot path (branch prediction)
0x02  LIKELY_COLD      instr is rarely executed
0x03  PURE             function call has no side effects, safe to CSE
0x04  NO_ALIAS         two memory ops do not alias
0x05  PIPELINE_FUSABLE stage can be fused with adjacent stage
0x06  UNROLL_COUNT     loop unroll factor hint (value = count)
```

2. The JIT Type Annotation Section

The JIT compiler uses type annotations to specialize instruction sequences. Type annotations record the expected NaN-box tag for each register at each instruction, allowing the JIT to emit unboxed arithmetic for registers that are always integers or always floats.

```
// Type annotation record
TypeAnnotation {
    instr_offset: u16,
    reg:          u8,
    tag_mask:     u8,   // bitmask of possible tags (3 bits used)
}

// Example: register %r3 is always i64 at instruction 0x0042
TypeAnnotation { instr_offset: 0x0042, reg: 3, tag_mask: 0b010 }
```

The type annotation section is generated by the AOT compiler's type inference pass and embedded in the LBC file. When the JIT encounters a register with a known single tag, it specializes the instruction: an ADD on two registers annotated as i64 becomes a machine ADD without NaN-boxing overhead.

3. Debug Information Schema

Debug information is stored in a dedicated LBC section that maps instruction offsets to source locations, variable names, and type information. The schema follows the DWARF5 line number table format for source mapping, with Lateralus-specific extensions for type information.

```
// Debug section layout
DebugSection {
    source_file_table: [StringRef],  // paths to source files
    line_number_table: LineTable,     // DWARF5 format
    variable_table:    [VarRecord],   // name, scope, type
    type_table:        [TypeRecord],  // Lateralus type descriptors
}

VarRecord {
    name:       StringRef,
    reg:        u8,      // which register holds this variable
    scope_begin: u16,    // instruction offset where var enters scope
    scope_end:  u16,     // instruction offset where var leaves scope
    type_id:    u16,     // index into type_table
}
```

Debuggers (including the Lateralus VS Code extension) read the debug section to provide hover-type information, watch expressions, and breakpoint setting at the source level.

4. Pipeline Stage Metadata

Pipeline expressions in LBC carry stage metadata: for each PIPE_CALL instruction, a metadata record lists the operator variant, the stage index within the pipeline, and a reference to the pipeline's type annotation.

```
PipelineMetadata {
    pipeline_id:  u16,  // identifies which pipeline this stage belongs to
    stage_index:  u8,
    variant:      u8,   // 0=total, 1=error, 2=async, 3=fanout
    fusable:      bool,
    pure:         bool,
}
```

The pipeline metadata enables the profiler to report per-stage latency and throughput, making it possible to identify bottleneck stages in a production pipeline.

5. Hot-Swap Patch Format

The hot-swap format allows updating a single function in a running Lateralus process without stopping execution. A patch consists of the new function's LBC record and a patch header:

```
// Hot-swap patch header
magic:      u32   = 0x50415443  ('PATC')
version:    u16   = 1
target_fn:  u32   // index of function to replace
old_crc:    u32   // CRC32 of the current function (safety check)
new_record: FunctionRecord  // replacement function
new_hints:  HintSection
new_debug:  DebugSection
```

The runtime applies the patch by: (1) pausing threads that are currently executing the target function (they run to the next safepoint), (2) verifying the old_crc matches the current function, (3) replacing the

function record and invalidating the JIT cache entry, (4) resuming paused threads which will re-enter the new function on next dispatch.

6. Verifier

LBC files are verified before execution. The verifier checks:

- Magic bytes and version compatibility.
- All instruction register references are within bounds (reg < n_regs).
- All jump targets are valid instruction offsets.
- The constant table index in LOAD_CONST instructions is within bounds.
- The function call target in CALL and TAIL_CALL instructions exists in the file.
- The pipeline metadata references valid pipeline IDs.

Verification is O(N) in the number of instructions and runs in approximately 1 ms per 100,000 instructions. The verifier is always run on externally-loaded LBC files; files produced by the trusted compiler can optionally skip verification via a flag.

7. Encoding the |?> Short-Circuit

The |?> operator compiles to a sequence: a PIPE_CALL to invoke the stage, followed by a PIPE_ERR_CHECK that tests the result tag and jumps to the error exit block if the tag is Err:

```
// Compiled |?> pipeline with three stages
PIPE_CALL  %r1, fn:parse       // %r1 = parse(input)
PIPE_ERR_CHECK %r1, label:err  // if Err, jump to err
PIPE_CALL  %r2, fn:validate    // %r2 = validate(%r1.ok)
PIPE_ERR_CHECK %r2, label:err
PIPE_CALL  %r3, fn:serialize   // %r3 = serialize(%r2.ok)
PIPE_ERR_CHECK %r3, label:err
RET %r3
err:
    RET %current_err           // single error exit point
```

The JIT merges the three PIPE_ERR_CHECK targets into a single branch: all three jump to the same error block, so the branch predictor learns the single prediction for the hot path.

8. Versioning and Stability

The LBC format version is a monotonically increasing integer in the file header. The Lateralus runtime supports reading LBC files of any version <= its own version. It refuses to run files of a higher version.

Backward-incompatible changes to the format increment the major version field. The current format (described in this paper and the introductory LBC paper) is version 1. Version 2 will add the compressed instruction form and structured exception table; the magic bytes will remain the same and the version field will change from 1 to 2.

LBC files produced by the Lateralus 1.x compiler are guaranteed to be readable by the Lateralus 2.x runtime.