Low-Level Theory: From Source Code to Running Process

Table of Contents

Current Architecture Analysis

Your Existing Components (Low-Level Perspective)

Arena Memory Allocator (`arena.h`)

Theoretical Foundation: Your arena allocator implements a **bump pointer allocation** strategy - the simplest and fastest allocation algorithm possible.

```c
// Core data structure
struct Region {
    Region *next;        // Linked list of memory chunks
    size_t count;       // Current allocation offset (in uintptr_t units)
    size_t capacity;     // Total capacity (in uintptr_t units)
    uintptr_t data[];    // Flexible array member - actual memory
};
```

Memory Layout in Physical RAM:

```
Region Structure:
┌─────────────┬─────────────┬─────────────┬─────────────────────────┐
│   next     │   count    │ capacity   │        data[]          │
│ (8 bytes)  │ (8 bytes)  │ (8 bytes)  │  (capacity * 8 bytes)    │
└─────────────┴─────────────┴─────────────┴─────────────────────────┘

Virtual Memory View:
┌─────────────┬─────────────┬─────────────┬─────────────────────────┐
│ 0x7fff...00 │ 0x7fff...08 │ 0x7fff...10 │    0x7fff...18...       │
└─────────────┴─────────────┴─────────────┴─────────────────────────┘
```

CPU-Level Operations:

```c
void *arena_alloc(Arena *a, size_t size_bytes) {
    // Convert bytes to uintptr_t alignment
    size_t size = (size_bytes + sizeof(uintptr_t) - 1)/sizeof(uintptr_t);

    // CPU instruction: LEA (Load Effective Address)
    // result_address = &a->end->data[a->end->count]
    void *result = &a->end->data[a->end->count];

    // CPU instruction: ADD
    // a->end->count += size (atomic on x64)
    a->end->count += size;

    return result; // Return in RAX register
}
```

Why This is Fast:

- **O(1) allocation** - single pointer arithmetic
- **No fragmentation** - linear allocation
- **Cache friendly** - sequential memory access
- **No metadata overhead** - no per-allocation headers

Resources:

- Memory Allocators 101 - Detailed allocator theory
- Intel Manual Volume 1 - Chapter 3.7: Memory Management

Lexer Implementation (`lexer.c`)

Finite State Machine Theory: Your lexer is a **Deterministic Finite Automaton (DFA)** that processes input character by character.

```c
// State transitions in LexerNextToken()
State_Current → Character_Input → State_Next

INITIAL      → [a-zA-Z_]     → IDENTIFIER
INITIAL      → [0-9]         → NUMBER
INITIAL      → '"'           → STRING
IDENTIFIER   → [a-zA-Z0-9_]  → IDENTIFIER
IDENTIFIER   → OTHER         → EMIT_ID_TOKEN
NUMBER       → [0-9]         → NUMBER
NUMBER       → OTHER         → EMIT_NUM_TOKEN
```

Memory Access Pattern:

```c
// CPU-level view of character reading
char LexerNextC(Lexer *lexer) {
   // Memory load: MOV AL, [lexer->src + lexer->curr]
   char c = lexer->src[lexer->curr++];

   // Branch prediction affects performance here
   if (c == '\n') {        // Conditional jump: JE
      ++(lexer->line);    // Memory store: INC QWORD PTR [lexer->line]
      lexer->column = 1;  // Memory store: MOV QWORD PTR [lexer->column], 1
   } else {
      ++(lexer->column);  // Memory store: INC QWORD PTR [lexer->column]
   }
   return c;              // Return in AL register
}
```

CPU Performance Implications:

- **Branch prediction** critical for `if` statements in character processing
- **L1 cache hits** for sequential string scanning
- **Register allocation** for frequently used variables (`c`, `lexer->curr`)

Resources:

- Crafting Interpreters - Scanning - Lexer theory
- CPU Branch Prediction - Performance implications

Parser Implementation (`parser.c`)

Recursive Descent Theory: Your parser implements a **recursive descent parser** - each grammar rule becomes a recursive function.

```c
// Grammar to Function Mapping
Grammar Rule: expression → term (('+' | '-') term)*
Function:     AST_Node *ParseTerm(Parser *parser)

// CPU call stack during parsing "1 + 2 * 3"
ParseExpression()
├── ParseTerm()
│   └── ParseFactor()
│       └── ParsePrimary() // "1"
├── ParseTerm()
│   ├── ParseFactor()
│   │   └── ParsePrimary() // "2"
│   └── ParseFactor()
│       └── ParsePrimary() // "3"
```

Stack Frame Layout:

```
Stack growth (towards lower addresses):
┌─────────────────────────────────────┐ ← RSP (Stack Pointer)
│ ParsePrimary() locals               │
├─────────────────────────────────────┤
│ ParseFactor() return address        │
├─────────────────────────────────────┤
│ ParseFactor() RBP (Frame Pointer)   │
├─────────────────────────────────────┤
│ ParseFactor() locals                │
├─────────────────────────────────────┤
│ ParseTerm() return address          │
└─────────────────────────────────────┘ ← RBP
```

Memory Allocation Flow:

```c
// AST node creation
AST_Node *ASTNodeCreate(Parser *parser, AST_Type type) {
   // This calls arena_alloc() internally
   AST_Node *node = arena_alloc(parser->arena, sizeof(AST_Node));

   // CPU instructions generated:
   // MOV RDI, parser->arena    ; First argument
   // MOV RSI, 64               ; sizeof(AST_Node)
   // CALL arena_alloc          ; Function call
   // MOV [result], RAX         ; Store return value

   return node;
}
```

Resources:

- Recursive Descent Parsing - Theory
- Dragon Book Chapter 4 - Parsing algorithms

Memory Management Deep Dive

Virtual Memory System

Linux Memory Layout for Your Compiler:

```
Virtual Address Space (64-bit):
┌─────────────────────────────────┐ 0x7FFFFFFFFFFF
│         Kernel Space            │
├─────────────────────────────────┤ 0x7FFF00000000
│          Stack                  │ ← RSP grows downward
│           ↓                     │
├─────────────────────────────────┤
│                                 │
│      Memory Mapping             │ ← mmap(), shared libraries
│                                 │
├─────────────────────────────────┤
│           ↑                     │
│          Heap                   │ ← malloc(), your arenas
├─────────────────────────────────┤
│         .bss                    │ ← Uninitialized globals
├─────────────────────────────────┤
│         .data                   │ ← Initialized globals
├─────────────────────────────────┤
│         .text                   │ ← Your compiler executable
└─────────────────────────────────┘ 0x400000
```

Arena Backend Implementation Analysis

Linux mmap() Backend:

```c
#if ARENA_BACKEND == ARENA_BACKEND_LINUX_MMAP
Region *new_region(size_t capacity) {
    size_t size_bytes = sizeof(Region) + sizeof(uintptr_t) * capacity;

    // System call to kernel
    Region *r = mmap(
        NULL,                    // Let kernel choose address
        size_bytes,              // Size to allocate
        PROT_READ | PROT_WRITE,  // Memory protection
        MAP_ANONYMOUS | MAP_PRIVATE, // Anonymous private mapping
        -1,                      // No file descriptor
        0                        // No offset
    );

    // Kernel operations:
    // 1. Find free virtual address range
    // 2. Allocate physical pages
    // 3. Update page tables
    // 4. Return virtual address

    return r;
}
```

Page Table Walk (CPU Hardware):

```
Virtual Address: 0x7FFF12345678
┌────────┬────────┬────────┬────────┬────────┬────────────┐
│  63  │  48  │  39  │  30  │  21  │    12    │
│ Sign  │  PML4  │  PDPT  │   PD   │   PT   │   Offset   │
│ Extend │ Index  │ Index  │ Index  │ Index  │            │
└────────┴────────┴────────┴────────┴────────┴────────────┘

Hardware Translation:
1. CR3 register → PML4 table
2. PML4[Sign Extended bits] → PDPT table
3. PDPT[PDPT Index] → PD table
4. PD[PD Index] → PT table
5. PT[PT Index] → Physical page
6. Physical page + Offset → Final address
```

TLB (Translation Lookaside Buffer):

- **L1 TLB**: ~64 entries, 1 cycle access
- **L2 TLB**: ~1536 entries, ~7 cycles
- **Page table walk**: ~200+ cycles

Resources:

- Linux Memory Management - Complete mm theory
- Intel Manual Volume 3A - Chapter 4: Paging

Cache Hierarchy Impact

Your Arena's Cache Behavior:

```
L1 Cache (32KB, 8-way associative):
┌─────────────┬─────────────┬─────────────┐
│   Tag    │  Index  │ Offset  │
│ 51 bits │ 6 bits  │ 6 bits  │ (64-byte cache lines)
└─────────────┴─────────────┴─────────────┘

Arena allocation pattern:
Address 1: 0x7FFF12340000  │ Same cache line
Address 2: 0x7FFF12340008  │ (perfect locality)
Address 3: 0x7FFF12340016  │
Address 4: 0x7FFF12340024  │
```

Cache Miss Costs:

- **L1 hit**: ~1 cycle
- **L2 hit**: ~4 cycles
- **L3 hit**: ~40 cycles
- **DRAM**: ~200+ cycles

Your arena's advantage: Sequential allocation = maximum cache utilization

Resources:

- Gallery of Processor Cache Effects - Cache performance analysis
- What Every Programmer Should Know About Memory - Comprehensive memory hierarchy

Compilation Process Theory

Compiler Pipeline Theory

Your Current Pipeline:

```
Source Text → Lexical Analysis → Syntactic Analysis → Semantic Analysis → Code Generation

"x := 42;"
   ↓
[TOKEN_ID: "x"] [TOKEN_ASSIGN: ":="] [TOKEN_NUM: 42] [TOKEN_SEMICOLON: ";"]
   ↓
AST_ASSIGNMENT {
   name: "x",
   right: AST_NUM { value: 42 }
}
   ↓
(Semantic Analysis - Type checking, symbol resolution)
   ↓
Code Generation (FASM/x64)
```

Information Theory:

- **Source entropy**: Variable based on language complexity
- **Token compression**: Reduce character stream to semantic tokens
- **AST normalization**: Canonical representation for code generation

Symbol Table Implementation

Hash Table Theory:

```c
// If you implement symbol tables later
typedef struct Symbol {
    char *name;          // Key
    SymbolType type;     // Value part 1
    void *data;          // Value part 2
    struct Symbol *next; // Collision resolution (chaining)
} Symbol;

// Hash function (simplified)
uint32_t hash_symbol(const char *name) {
    uint32_t hash = 0;
    while (*name) {
        hash = hash * 31 + *name++;  // Prime multiplication
    }
    return hash;
}
```

Hash Collision Resolution:

- **Chaining**: Used above (linked list)
- **Open addressing**: Linear probing, quadratic probing
- **Robin Hood hashing**: Minimize worst-case access

Load Factor Impact:

- **α < 0.75**: Good performance
- **α > 0.9**: Degraded performance due to collisions

Resources:

- Hash Table Implementation - Theory and analysis
- Engineering a Compiler Chapter 5 - Symbol tables

Machine Code Generation

x86-64 Instruction Encoding

Instruction Format:

```
┌─────────┬─────────┬─────────┬─────────┬─────────┬─────────┐
│ Prefix  │ Opcode  │ModR/M   │   SIB   │  Disp   │   Imm   │
│ 0-4 B   │ 1-3 B   │ 0-1 B   │ 0-1 B   │0,1,2,4B │0,1,2,4,8│
└─────────┴─────────┴─────────┴─────────┴─────────┴─────────┘
```

Example: `mov rax, 42`

```asm
; Assembly: mov rax, 42
; Machine code: 48 C7 C0 2A 00 00 00

Breakdown:
48        = REX prefix (REX.W = 1, 64-bit operand)
C7        = Opcode (MOV r/m64, imm32)
C0        = ModR/M (11 000 000 = register RAX)
2A000000  = Immediate value 42 (little-endian)
```

REX Prefix Calculation:

```c
uint8_t rex_prefix = 0x40 | (w << 3) | (r << 2) | (x << 1) | b;
// w = 1 (64-bit operand)
// r = 0 (no extension to ModR/M reg field)
// x = 0 (no extension to SIB index field)
// b = 0 (no extension to ModR/M r/m field)
// Result: 0x48
```

ModR/M Byte:

```
┌────────┬────────┬────────┐
│  Mod   │  Reg   │  R/M   │
│ 2 bits │ 3 bits │ 3 bits │
└────────┴────────┴────────┘

For register RAX:
Mod = 11 (register direct)
Reg = 000 (part of opcode extension)
R/M = 000 (RAX register)
Result: 11000000 = 0xC0
```

Resources:

- Intel Manual Volume 2 - Complete instruction reference
- x86-64 Instruction Encoding - Practical encoding guide
- Online x86 Assembler - Test encoding

Register Allocation Theory

Graph Coloring Algorithm:

```
Interference Graph for expression "a = b + c * d":

Variables: a, b, c, d, temp1
Live ranges:
- b: [1, 3]
- c: [1, 4]
- d: [1, 4]
- temp1: [2, 4]
- a: [4, 5]

Interference edges:
b -- c    (both live at point 1-3)
b -- d    (both live at point 1-3)
c -- d    (both live at point 1-4)
c -- temp1 (both live at point 2-4)
d -- temp1 (both live at point 2-4)
```

Register Assignment:

- **Coloring**: Assign registers such that no adjacent nodes have same color
- **Spilling**: When colors exhausted, store some variables in memory
- **Coalescing**: Combine variables that don't interfere

Naive Allocation (Good for Fast Compilation):

- **Round-robin**: Assign registers in fixed order
- **Usage counting**: Track register usage, prefer less-used registers
- **Simple spilling**: Spill to stack when registers exhausted

Resources:

- Register Allocation via Graph Coloring - Classic algorithm
- Linear Scan Register Allocation - Faster alternative

ELF64 Binary Format

ELF Header Structure

Complete ELF64 Header:

```c
typedef struct {
    unsigned char e_ident[EI_NIDENT];  // Magic + metadata
    Elf64_Half    e_type;              // Object file type
    Elf64_Half    e_machine;           // Machine architecture
    Elf64_Word    e_version;           // Object file version
    Elf64_Addr    e_entry;             // Entry point address
    Elf64_Off     e_phoff;             // Program header offset
    Elf64_Off     e_shoff;             // Section header offset
    Elf64_Word    e_flags;             // Processor-specific flags
    Elf64_Half    e_ehsize;            // ELF header size
    Elf64_Half    e_phentsize;         // Program header entry size
    Elf64_Half    e_phnum;             // Program header entry count
    Elf64_Half    e_shentsize;         // Section header entry size
    Elf64_Half    e_shnum;             // Section header entry count
    Elf64_Half    e_shstrndx;          // Section header string table index
} Elf64_Ehdr;
```

Magic Number Analysis:

```c
e_ident[0] = 0x7F;     // DEL character
e_ident[1] = 'E';      // ASCII 'E'
e_ident[2] = 'L';      // ASCII 'L'
e_ident[3] = 'F';      // ASCII 'F'
e_ident[4] = ELFCLASS64;   // 64-bit format
e_ident[5] = ELFDATA2LSB;  // Little-endian
e_ident[6] = EV_CURRENT;   // Current version
e_ident[7] = ELFOSABI_SYSV; // System V ABI
```

Binary Representation:

```
Offset  Bytes                    Meaning
0x00    7F 45 4C 46                 Magic "\x7FELF"
0x04    02                       ELFCLASS64
0x05    01                       ELFDATA2LSB
0x06    01                       EV_CURRENT
0x07    00                       ELFOSABI_SYSV
0x08    00 00 00 00 00 00 00 00       Padding
```

Program Headers

Loadable Segment:

```c
typedef struct {
   Elf64_Word  p_type;    // Segment type (PT_LOAD, PT_INTERP, etc.)
   Elf64_Word  p_flags;   // Segment flags (PF_X, PF_W, PF_R)
   Elf64_Off   p_offset;  // Segment file offset
   Elf64_Addr  p_vaddr;   // Segment virtual address
   Elf64_Addr  p_paddr;   // Segment physical address
   Elf64_Xword p_filesz;  // Segment size in file
   Elf64_Xword p_memsz;   // Segment size in memory
   Elf64_Xword p_align;   // Segment alignment
} Elf64_Phdr;
```

Typical Executable Layout:

```
Program Header 0: PT_LOAD (Text segment)
├─ p_vaddr:  0x400000      (Virtual load address)
├─ p_flags:  PF_R | PF_X    (Read + Execute)
├─ p_filesz: 4096          (Size in file)
└─ p_memsz:  4096          (Size in memory)

Program Header 1: PT_LOAD (Data segment)
├─ p_vaddr:  0x600000      (Virtual load address)
├─ p_flags:  PF_R | PF_W    (Read + Write)
├─ p_filesz: 1024          (Size in file)
└─ p_memsz:  2048          (Size in memory - includes .bss)
```

Resources:

- ELF Specification - Complete format specification
- ELF Wikipedia - Overview
- readelf manpage - Analysis tools

CPU Execution Model

x86-64 Architecture

Register File:

```
General Purpose Registers (16 × 64-bit):
RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8-R15

Special Registers:
RIP  - Instruction Pointer (Program Counter)
RFLAGS - Processor Flags (Zero, Carry, etc.)

Segment Registers (Legacy, mostly unused in 64-bit):
CS, DS, ES, FS, GS, SS
```

Pipeline Architecture (Simplified Intel Core):

```
┌─────────┐ ┌─────────┐ ┌─────────┐ ┌─────────┐ ┌─────────┐
│ Fetch   │ │ Decode  │ │ Execute │ │ Memory  │ │Writeback│
│ (IF)    │ │ (ID)    │ │ (EX)    │ │ (MEM)   │ │ (WB)    │
└─────────┘ └─────────┘ └─────────┘ └─────────┘ └─────────┘
     ↑           ↑           ↑           ↑           ↓
Fetch from  Decode x64   ALU/FPU    L1 Cache    Register
 I-Cache    instruction  execution  access      file
```

Superscalar Execution:

- **Multiple execution units** can run in parallel
- **Out-of-order execution** for performance
- **Register renaming** to avoid false dependencies
- **Branch prediction** to avoid pipeline stalls

Virtual Memory Translation

Page Table Entry Structure:

```
┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────────────┐
│  P  │ R/W │ U/S │ PWT │ PCD │  A  │  D  │ PAT │ Physical    │
│     │     │     │     │     │     │     │     │ Address     │
└─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────────────┘
  0     1     2     3     4     5     6     7    12        51

P   = Present (1 if page is in memory)
R/W = Read/Write (0 = read-only, 1 = read-write)
U/S = User/Supervisor (0 = kernel only, 1 = user accessible)
A   = Accessed (set by CPU on any access)
D   = Dirty (set by CPU on write access)
```

Address Translation Process:

1. **CPU** generates virtual address
2. **MMU** checks TLB for translation
3. If **TLB miss**, hardware page table walk
4. **Physical address** used for memory access
5. **TLB update** for future accesses

Resources:

- Intel Manual Volume 3A Chapter 4 - Paging
- AMD64 Architecture Manual Volume 2 - System programming

System Call Interface

Linux System Call Mechanism

System Call Entry:

```asm
; User space code
mov rax, 60        ; sys_exit system call number
mov rdi, 0         ; exit status
syscall            ; Transfer to kernel

; CPU hardware actions:
; 1. Save user RIP to RCX
; 2. Save RFLAGS to R11
; 3. Load kernel RIP from MSR_LSTAR
; 4. Switch to ring 0 (kernel mode)
; 5. Switch to kernel stack
```

Kernel Entry Point (`entry_SYSCALL_64`):

```asm
; arch/x86/entry/entry_64.S (Linux kernel)
ENTRY(entry_SYSCALL_64)
    /* Save user registers */
    pushq   %rax
    pushq   %rcx
    pushq   %rdx
    pushq   %rsi
    pushq   %rdi
    pushq   %r8
    pushq   %r9
    pushq   %r10
    pushq   %r11

    /* Call system call handler */
    call    do_syscall_64

    /* Restore and return to user space */
    sysretq
ENDPROC(entry_SYSCALL_64)
```

System Call Table:

```c
// arch/x86/entry/syscalls/syscall_64.tbl
0   common  read                  sys_read
1   common  write                 sys_write
2   common  open                  sys_open
3   common  close                 sys_close
...
60  common  exit                  sys_exit
```

sys_write Implementation:

```c
// fs/read_write.c
SYSCALL_DEFINE3(write, unsigned int, fd,
                const char __user *, buf, size_t, count)
{
    struct fd f = fdget_pos(fd);      // Get file descriptor
    if (f.file) {
        loff_t pos = file_pos_read(f.file);
        ret = vfs_write(f.file, buf, count, &pos);  // Virtual File System
        if (ret >= 0)
            file_pos_write(f.file, pos);
        fdput_pos(f);
    }
    return ret;
}
```

Performance Characteristics:

- **syscall instruction**: ~150-300 cycles (fast path)
- **Context switch overhead**: Save/restore registers
- **TLB flush cost**: If kernel uses different page tables
- **vsyscall/vDSO**: Kernel code mapped in user space for fast calls

Resources:

- Linux System Call Interface - Complete list
- Understanding the Linux Kernel Chapter 10 - System calls
- Linux Kernel Source - Entry points

Complete Execution Pipeline

From Source to Process

1. Compilation Phase:

```
your_program.lang
      ↓ (Your C frontend)
   Lexical Analysis → Tokens
      ↓
   Syntactic Analysis → AST
      ↓
   Semantic Analysis → Annotated AST
      ↓ (FASM backend)
   Code Generation → Assembly
      ↓ (FASM assembler)
   Machine Code → ELF64 binary
```

2. Loading Phase:

```bash
$ ./your_program

# Kernel actions:
# 1. execve() system call
# 2. ELF loader reads program headers
# 3. mmap() creates virtual memory mappings
# 4. Copy program segments to memory
# 5. Set up initial stack
# 6. Jump to entry point (e_entry)
```

3. Runtime Execution:

```
Process Virtual Address Space:
┌─────────────────────────────────────┐ 0x7FFFFFFFFFFF
│   Kernel Space (Ring 0)          │
├─────────────────────────────────────┤ 0x7FFF00000000
│   Stack (grows down)             │ ← RSP
│   - Return addresses             │
│   - Local variables              │
│   - Function parameters          │
├─────────────────────────────────────┤
│   Heap (your arenas)             │ ← malloc region
├─────────────────────────────────────┤
│   .bss (uninitialized data)      │ ← Zero-filled
├─────────────────────────────────────┤
│   .data (initialized data)       │ ← Global variables
├─────────────────────────────────────┤
│   .text (your generated code)    │ ← Machine instructions
└─────────────────────────────────────┘ 0x400000
```

4. Instruction Execution Cycle:

```
1. Fetch: CPU reads instruction from memory at RIP
2. Decode: Instruction decoder determines operation
3. Execute: ALU/FPU performs computation
4. Memory: Load/store data if needed
5. Writeback: Update registers with results
6. Update RIP: Point to next instruction
```

5. System Interaction:

```
Your Program              Linux Kernel
     │                         │
     │ ─── syscall ──────────→ │
     │                         │ (Handle system call)
     │                         │ (Access hardware/filesystem)
     │ ←───── return ───────── │
     │                         │
```

Performance Analysis Tools

CPU Performance Counters:

```bash
# Measure your compiler's performance
perf stat -e cpu-cycles,instructions,L1-dcache-misses,branch-misses ./your_compiler

# Results show:
# - Instructions per cycle (IPC)
# - Cache miss rates
# - Branch prediction accuracy
# - Memory bandwidth utilization
```

Cache Analysis:

```bash
# Cache simulation
perf stat -e L1-dcache-loads,L1-dcache-load-misses,LLC-loads,LLC-load-misses ./your_compiler

# Expected for arena allocator:
# - High L1 hit rate (sequential allocation)
# - Low LLC miss rate (good locality)
# - High cache efficiency
```

Memory Analysis:

```bash
# Virtual memory usage
pmap -x $(pgrep your_compiler)

# Shows:
# - Virtual memory layout
# - RSS (Resident Set Size)
# - Shared vs private pages
# - Memory-mapped regions
```

Theoretical Performance Limits

Your Arena Allocator:

- **Allocation speed**: Limited by memory bandwidth (~100 GB/s DDR4)
- **Theoretical limit**: ~800M allocations/second (8-byte objects)
- **Practical limit**: ~200M allocations/second (CPU overhead)

Your Lexer:

- **Scanning speed**: Limited by branch prediction and cache misses
- **Theoretical limit**: ~2 billion characters/second (L1 cache speed)
- **Practical limit**: ~500M characters/second (branch overhead)

Your Parser:

- **Parse speed**: Limited by function call overhead and memory allocation
- **Recursive descent**: ~10-50 cycles per AST node
- **Practical limit**: ~100M AST nodes/second

Code Generation:

- **FASM compilation**: ~1M lines/second assembly processing
- **Machine code emission**: Memory bandwidth limited
- **ELF creation**: Dominated by I/O (fsync, file system)

Resources:

- Software Optimization Guide - Comprehensive performance analysis
- Linux perf Tutorial - Performance measurement
- Intel VTune Profiler - Advanced profiling

Integration Points

How Your Components Interact

Memory Flow:

```
Source File (mmap'd)
    ↓
Lexer (sequential scan, arena-allocated tokens)
    ↓
Parser (recursive calls, arena-allocated AST)
    ↓
Code Generator (FASM macro expansion)
    ↓
ELF64 Binary (kernel loader)
    ↓
Running Process (virtual memory, system calls)
```

Performance Bottlenecks:

1. **I/O bound**: Reading source files (solved by buffering)
2. **Memory bound**: Large ASTs (solved by arena allocation)
3. **CPU bound**: Complex parsing (solved by recursive descent)
4. **System bound**: File creation (solved by in-memory generation)

Optimization Opportunities:

1. **Parallel lexing**: Split input into chunks
2. **Incremental parsing**: Only reparse changed functions
3. **Template instantiation**: Pre-compile common patterns
4. **JIT compilation**: Compile frequently-used code first

This analysis shows that your architecture is theoretically sound and should achieve the target performance of **1M+ lines/second compilation** through optimal memory management, efficient algorithms, and direct machine code generation.