

# Code Generation for Data Processing

## Lecture 1: Introduction and Interpretation

Alexis Engelke

Chair of Data Science and Engineering (I25)  
School of Computation, Information, and Technology  
Technical University of Munich

Winter 2025/26

# Module “Code Generation for Data Processing”

## Learning Goals

- ▶ Getting from an intermediate code representation to machine code
- ▶ Designing and implementing IRs and machine code generators
- ▶ Apply for: JIT compilation, query compilation, ISA emulation

## Prerequisites

- ▶ Computer Architecture, Assembly ERA, GRA/ASP
- ▶ Databases, Relational Algebra GDB
- ▶ Beneficial: Compiler Construction, Modern DBs

# Topic Overview

## Introduction

- ▶ Introduction and Interpretation
- ▶ Compiler Front-end

## Intermediate Representations

- ▶ IR Concepts and Design
- ▶ LLVM-IR
- ▶ Analyses and Optimizations

## Compiler Back-end

- ▶ Instruction Selection
- ▶ Register Allocation
- ▶ Linker, Loader, Debuginfo

## Applications

- ▶ JIT-compilation + Sandboxing
- ▶ Query Compilation
- ▶ Binary Translation

# Lecture Organization

- ▶ Lecturer: Dr. Alexis Engelke [engelke@in.tum.de](mailto:engelke@in.tum.de)
- ▶ Time slot: Thu 10-14, 02.11.018
- ▶ Material: <https://db.in.tum.de/teaching/ws2526/codegen/>

## Exam

- ▶ Written exam, 90 minutes, **no retake**, 2026-02-27 14:00
- ▶ (Might change to oral on very low registration count)

## Exercises

- ▶ Regular homework, often with programming exercise
- ▶ Submission via POST request (see assignments)
  - ▶ Grading with  $\{*, +, \sim, -\}$ , feedback on best effort
- ▶ Exercises integrated into lecture
- ▶ Hands-on programming or analysis of systems (needs laptop)
- ▶ Occasionally: present and discuss homework solutions

## Grade Bonus

- ▶ Requirement:  $N - 2$  “sufficiently working” homework submissions and one presentations of homework in class (depends on submission count)
- ▶ Bonus: grades in  $[1.3; 4.0]$  improved by  $0.3/0.4$

# Why study compilers?

- ▶ Critical component of every system, functionality and performance
  - ▶ Compiler mostly *alone* responsible for using hardware well
- ▶ Brings together many aspects of CS:
  - ▶ Theory, algorithms, systems, architecture, software engineering, (ML)
- ▶ New developments/requirements pose new challenges
  - ▶ New architectures, environments, language concepts, . . .
- ▶ High complexity!

# Compiler Lectures @ TUM

## Compiler Construction

IN2227, SS, THEO

Front-end, parsing, semantic analyses, types

## Program Optimization

IN2053, WS, THEO

Analyses, transformations, abstract interpretation

## Virtual Machines

IN2040, SS, THEO

Mapping programming paradigms to IR/bytocode

## Programming Languages

CIT3230000, WS

Implementation of advanced language features

## Code Generation

CIT3230001, WS

Back-end, machine code generation, JIT comp.

# Why study code generation?

- ▶ Frameworks (LLVM, ...) exist and are comparably good, but often not good enough (performance, features)
  - ▶ Many systems with code gen. have their own back-end
  - ▶ E.g.: V8, WebKit FTL, .NET RyuJIT, GHC, Zig, QEMU, Umbra, ...
- ▶ Machine code is not the only target: bytecode
  - ▶ Often used for code execution
  - ▶ E.g.: V8, Java, .NET MSIL, BEAM (Erlang), Python, MonetDB, eBPF, ...
  - ▶ Allows for flexible design
  - ▶ But: efficient execution needs machine code generation

## Proebsting's Law

“Compiler advances double computing power every 18 years.”

– Todd Proebsting, 1998<sup>1</sup>

- ▶ Still optimistic; depends on number of abstractions

<sup>1</sup><http://proebsting.cs.arizona.edu/law.html>

## Motivational Example: Brainfuck

- ▶ Turing-complete esoteric programming language, 8 operations
  - ▶ Input/output: . ,
  - ▶ Moving pointer over infinite array: < >
  - ▶ Increment/decrement: + -
  - ▶ Jump to matching bracket if (not) zero: [ ]

```
++++++[->+++++<]>.
```

- ▶ Execution with pen/paper? ☹

# Program Execution



## Programs

- ▶ High flexibility (possibly)
- ▶ Many abstractions (typically)
- ▶ Several paradigms

## Hardware/ISA

- ▶ Low-level interface
- ▶ Few operations, imperative
- ▶ “Not easy” to write

# Motivational Example: Brainfuck – Interpretation

- ▶ Write an interpreter!

```
unsigned char state[10000];
unsigned ptr = 0, pc = 0;
while (prog[pc])
    switch (prog[pc++]) {
        case '.': putchar(state[ptr]); break;
        case ',': state[ptr] = getchar(); break;
        case '>': ptr++; break;
        case '<': ptr--; break;
        case '+': state[ptr]++; break;
        case '-': state[ptr]--; break;
        case '[': state[ptr] || (pc = matchParen(pc, prog)); break;
        case ']': state[ptr] && (pc = matchParen(pc, prog)); break;
    }
```

# Program Execution

## Compiler



- ▶ Translate program to other lang.
- ▶ Might optimize/improve program
- ▶ C, C++, Rust → machine code
- ▶ Python, Java → bytecode

## Interpreter



- ▶ Directly execute program
- ▶ Computes program result
- ▶ Shell scripts, Python bytecode, machine code (conceptually)

Multiple compilation steps can precede the “final interpretation”

# Compilers

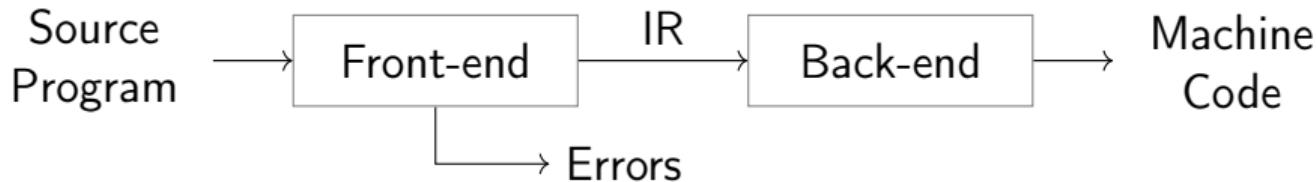
- ▶ Targets: machine code, bytecode, or other source language
- ▶ Typical goals: better language usability and performance
  - ▶ Make lang. usable at all, faster, use less resources, etc.
- ▶ Constraints: specs, resources (comp.-time, etc.), requirements (perf., etc.)
- ▶ Examples:
  - ▶ “Classic” compilers source → machine code
  - ▶ JIT compilation of JavaScript, WebAssembly, Java bytecode, ...
  - ▶ Database query compilation
  - ▶ ISA emulation/binary translation

## Compiler Structure: Monolithic



- ▶ Inflexible architecture, hard to retarget

# Compiler Structure: Two-phase architecture



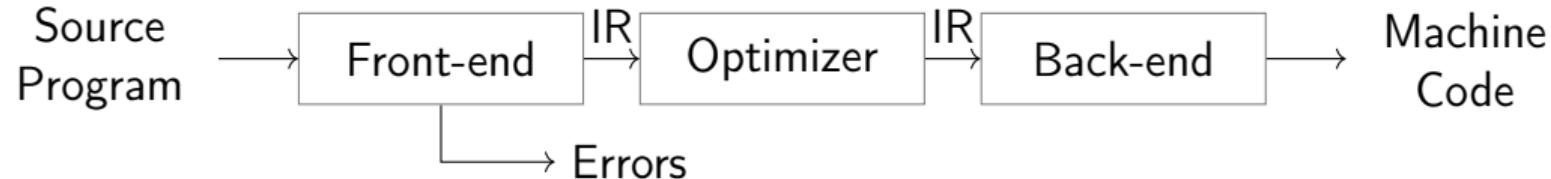
## Front-end

- ▶ Parses source code
- ▶ Detect syntax/semantical errors
- ▶ Emit *intermediate representation* encode semantics/knowledge
- ▶ Typically:  $\mathcal{O}(n)$  or  $\mathcal{O}(n \log n)$

## Back-end

- ▶ Translate IR to target architecture
- ▶ Can assume valid IR ( $\rightsquigarrow$  no errors)
- ▶ Possibly one back-end per arch.
- ▶ Contains  $\mathcal{NP}$ -complete problems

# Compiler Structure: Three-phase architecture



- ▶ Optimizer: analyze/transform/rewrite program inside IR

---

- ▶ Conceptual architecture: real compilers typically much more complex
  - ▶ Several IRs in front-end and back-end, optimizations on different IRs
  - ▶ Multiple front-ends for different languages
  - ▶ Multiple back-ends for different architectures

# Compiler Front-end

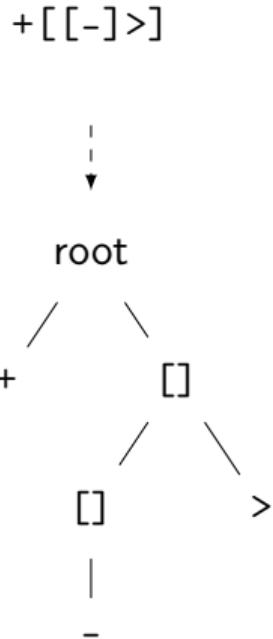
1. Tokenizer: recognize words, numbers, operators, etc.  $\mathcal{R}e$ 
  - ▶ Example:  $a+b*c \rightarrow \text{ID}(a) \text{ PLUS } \text{ID}(b) \text{ TIMES } \text{ID}(c)$
2. Parser: build (abstract) syntax tree, check for syntax errors  $\mathcal{CFG}$ 
  - ▶ Syntax Tree: describe grammatical structure of complete program  
Example: `expr("a", op("+"), expr("b", op("*"), expr("c")))`
  - ▶ Abstract Syntax Tree: only relevant information, more concise  
Example: `plus("a", times("b", "c"))`
3. Semantic Analysis: check types, variable existence, etc.
4. IR Generator: produce IR for next stage
  - ▶ This might be the AST itself

# Compiler Back-end

1. Instruction Selection: map IR operations to target instructions
  - ▶ Use target features: special insts., addressing modes, ...
  - ▶ Still using virtual/unlimited registers
2. Instruction Scheduling: optimize order for target arch.
  - ▶ Start memory/high-latency earlier, etc.
  - ▶ Requires knowledge about micro-architecture
3. Register Allocation: map values to fixed register set/stack
  - ▶ Use available registers effectively, minimize stack usage

## Motivational Example: Brainfuck – Front-end

- ▶ Need to skip comments
- ▶ Bracket searching is expensive/redundant
- ▶ Idea: “parse” program!
- ▶ Tokenizer: yield next operation, skipping comments
- ▶ Parser: find matching brackets, construct AST



# Motivational Example: Brainfuck – AST Interpretation

- ▶ AST can be interpreted recursively

```
struct node { char kind; unsigned cldCnt; struct node* cld; };
struct state { unsigned char* arr; size_t ptr; };
void donode(struct node* n, struct state* s) {
    switch (n->kind) {
        case '+': s->arr[s->ptr]++; break;
        // ...
        case '[': while (s->arr[s->ptr]) children(n, s); break;
        case 0: children(n, s); break; // root
    }
}
void children(struct node* n, struct state* s) {
    for (unsigned i = 0; i < n->cldCnt; i++) donode(n->cld + i, s);
}
```

## Motivational Example: Brainfuck – Optimization

- ▶ Inefficient sequences of `+/-</>` can be combined
  - ▶ Trivially done when generating IR
- ▶ Fold patterns into more high-level operations

Look at some Brainfuck programs. Which patterns are beneficial to fold?

## Motivational Example: Brainfuck – Optimization

- ▶ Fold offset into operation
  - ▶  $\text{right}(2) \text{ add}(1) = \text{addoff}(2, 1) \text{ right}(2)$
  - ▶ Also possible with loops
- ▶ Analysis: does loop move pointer?
  - ▶ Loops that keep position intact allow more optimizations
  - ▶ Maybe distinguish “regular loops” from arbitrary loops?
- ▶ Get rid of all “effect-less” pointer movements
- ▶ Combine arithmetic operations, disambiguate addresses, etc.

## Motivational Example: Brainfuck – Bytecode

- ▶ Tree is nice, but rather inefficient  $\rightsquigarrow$  flat and compact bytecode
- ▶ Avoid pointer dereferences/indirections; keep code size small
- ▶ Maybe dispatch two instructions at once?
  - ▶ `switch (ops[pc] | ops[pc+1] << 8)`
- ▶ Superinstructions: combine common sequences to one instruction

# Threaded Interpretation<sup>2</sup>

- ▶ Simple switch-case dispatch has lots of branch misses
- ▶ Threaded interpretation: at end of a handler, jump to next op

```
struct op { char op; char data; };
struct state { unsigned char* arr; size_t ptr; };
void threadedInterp(struct op* ops, struct state* s) {
    static const void* table[] = { &&CASE_ADD, &&CASE_RIGHT, };
#define DISPATCH do { goto *table[(++pc)->op]; } while (0)

    struct op* pc = ops;
    DISPATCH;

CASE_ADD: s->arr[s->ptr] += pc->data; DISPATCH;
CASE_RIGHT: s->arr += pc->data; DISPATCH;
}
```

<sup>2</sup>MA Ertl and D Gregg. "The structure and performance of efficient interpreters". In: *JILP* 5 (2003), pp. 1–25. 

# Threaded Interpretation with Tail Calls

- ▶ Threaded interpretation can also be implemented with tails calls

```
struct op { char op; char data; };
struct state { unsigned char* arr; size_t ptr; };
void tcInterp(struct op* pc, struct state* s);
static void fn_add(struct op* pc, struct state* s) {
    s->arr[s->ptr] += pc->data; tcInterp(pc + 1, s); }
static void fn_right(struct op* pc, struct state* s) {
    s->arr += pc->data; tcInterp(pc + 1, s); }
void tcInterp(struct op* pc, struct state* s) {
    typedef void (*Fn)(struct op* pc, struct state* s);
    static const Fn fns[] = { fn_add, fn_right };
    fns[pc->op](pc, s);
}
```

# Direct Threading<sup>5</sup>

- ▶ Use function pointer as operation to avoid indirection

```
struct op; struct state;
typedef void (* Fn)(struct op* pc, struct state* s);
struct op { Fn op; char data; };
struct state { unsigned char* arr; size_t ptr; };

void fn_add(struct op* pc, struct state* s) {
    s->arr[s->ptr] += pc->data; pc[1].op(pc + 1, s); }
void fn_right(struct op* pc, struct state* s) {
    s->arr += pc->data; pc[1].op(pc + 1, s); }
void dtInterp(struct op* ops, struct state* s) {
    ops[0].op(ops, s);
}
```

<sup>5</sup> JR Bell. "Threaded Code". In: CACM 16.6 (1973), pp. 370–372. 

# Threaded Interpretation — Comparison

What are benefits/drawbacks of the three threading approaches<sup>6</sup>?

- ▶ Indirect threading with computed goto
- ▶ Indirect threading with tail calls
- ▶ Direct threading with tail calls

- ▶ Some differences on code size, readability, and maintainability
- ▶ Performance: depends on hardware, context – always measure!

<sup>6</sup><https://godbolt.org/z/4rcEv4sqj>

# Fast Interpretation

- ▶ Key technique to “avoid” compilation to machine code
- ▶ Preprocess program into efficiently executable bytecode
  - ▶ Easily identifiable opcode, homogeneous structure
  - ▶ Can be linear (fast to execute), but trees also work
  - ▶ Match bytecode ops with needed operations  $\rightsquigarrow$  fewer instructions
  - ▶ Larger operations preferable, but not too many (prediction)
- ▶ Perhaps optimize – if it’s worth the benefit
  - ▶ Fold constants, combine instructions, ...
  - ▶ Consider superinstructions for common sequences
- ▶ For very cold code: avoid transformations at all

# Interpretation vs. Compilation

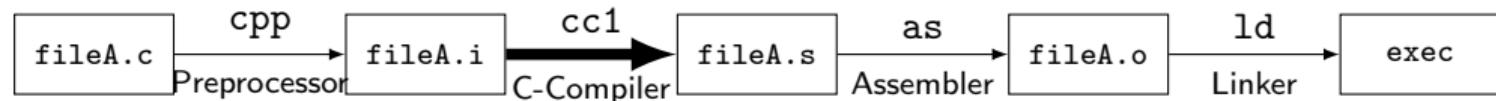
Fundamental benefits of compilation:

- ▶ Elimination of interpreter dispatch
  - ▶ Can be significant for bytecodes with many small operations
- ▶ Use of CPU registers for values across bytecode instructions
  - ▶ Interpreter: most values must be stored in memory
  - ▶ Register allocation can bring large improvements
- ▶ No fundamental benefit: optimizations
  - ▶ Many optimizations can also be applied on bytecode

Fundamental benefits of interpretation: simple, portable

# Compiler: Surrounding – Compile-time

- ▶ Typical environment for a C/C++ compiler:



- ▶ Calling Convention: interface with other objects/libraries
- ▶ Build systems, dependencies, debuggers, etc.
- ▶ Compilation target machine (hardware, VM, etc.)

## Compiler: Surrounding – Run-time

- ▶ OS interface (I/O, ...)
- ▶ Memory management (allocation, GC, ...)
- ▶ Parallelization, threads, ...
- ▶ VM for execution of virtual assembly (JVM, ...)
- ▶ Run-time type checking
- ▶ Error handling: exception unwinding, assertions, ...
- ▶ Reflection, RTTI

## Motivational Example: Brainfuck – Runtime Environment

- ▶ Needs I/O for . and ,
- ▶ Error handling: unmatched brackets
- ▶ Memory management: infinitely-sized array

How to efficiently emulate an infinitely sized array?

# Compilation point: AoT vs. JIT

## Ahead-of-Time (AoT)

- ▶ All code has to be compiled
- ▶ No dynamic optimizations
- ▶ Compilation-time secondary concern

## Just-in-Time (JIT)

- ▶ Compilation-time is critical
- ▶ Code can be compiled on-demand
  - ▶ Incremental optimization, too
- ▶ Handle cold code fast
- ▶ Dynamic specializations possible
- ▶ Allows for `eval()`

Various hybrid combinations possible

## Introduction and Interpretation – Summary

- ▶ Compilation vs. interpretation and combinations
- ▶ Compilers are key to usable/performant languages
- ▶ Target language typically machine code or bytecode
- ▶ Three-phase architecture widely used
- ▶ Interpretation techniques: bytecode, threaded interpretation, ...
- ▶ JIT compilation imposes different constraints

## Introduction and Interpretation – Questions

- ▶ What is typically compiled and what is interpreted? Why?
  - ▶ PostScript, C, JavaScript, HTML, SQL
- ▶ What are typical types of output languages of compilers?
- ▶ How does a compiler IR differ from the source input?
- ▶ What is the impact of the language paradigm on optimizations?
- ▶ What are important factors for an efficient interpreter?
- ▶ What are inherent benefits of compilation over interpretation?
- ▶ What are key differences between AoT and JIT compilation?

# Code Generation for Data Processing

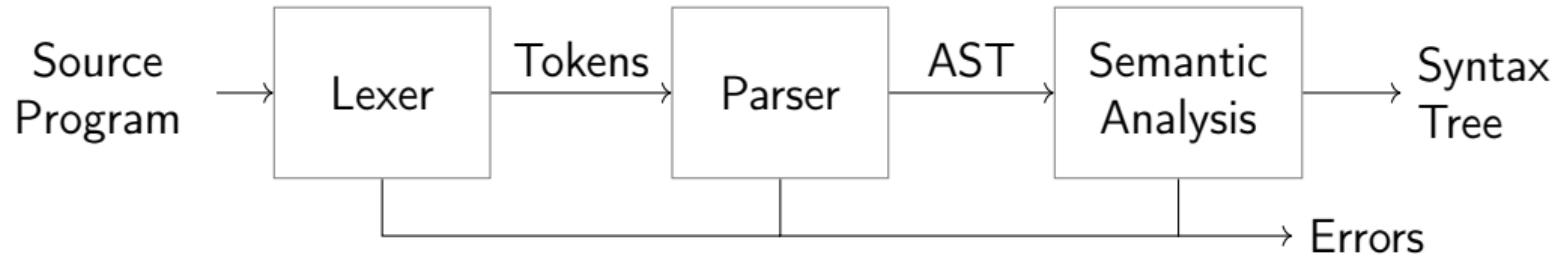
## Lecture 2: Compiler Front-end

Alexis Engelke

Chair of Data Science and Engineering (I25)  
School of Computation, Information, and Technology  
Technical University of Munich

Winter 2025/26

# Compiler Front-end



- ▶ Typical architecture: separate lexer, parser, and context analysis
  - ▶ Allows for more efficient lexical analysis
  - ▶ Smaller components, easier to understand, etc.
- ▶ Some languages: preprocessor and macro expansion

## Lexer

- ▶ Convert stream of chars to stream of words (*tokens*)
- ▶ Detect/classify identifiers, numbers, operators, ...
- ▶ Strip whitespace, comments, etc.

$a+b*c \rightarrow \text{ID}(a) \text{ PLUS } \text{ID}(b) \text{ TIMES } \text{ID}(c)$

- ▶ Typically representable as regular expressions

# Typical Token Kinds

- ▶ Punctuators ( ) [ ] { } ; = + += | ||
- ▶ Identifiers abc123 main
- ▶ Keywords void int \_\_asm\_\_
- ▶ Numeric constants 123 0xab1 5.7e3 0x1.8p1 09.1f
- ▶ Char constants 'a' u'æ'
- ▶ String literals "abc\x12\n"
- ▶ Internal EOF COMMENT UNKNOWN INDENT DEDENT
  - ▶ Comments might be useful for annotations, e.g. // fallthrough

# Lexer Implementation

```
struct Token { enum Kind { IDENT, EOF, PLUS, PLUSEQ, /*...*/ };  
    std::string_view v; Kind kind; };  
Token next(std::string_view v) {  
    if (v.empty()) return Token{v, Token::EOF};  
    if (v.starts_with("+=")) return Token{"+="sv, Token::PLUSEQ};  
    if (v.starts_with("+")) return Token{ "+"sv, Token::PLUS};  
    switch (v[0]) {  
        case ' ', '\n', '\t': return next(v.substr(1)); // skip whitespace  
        case 'a' ... 'z', 'A' ... 'Z', '_': {  
            Token t = // ... parse identifier, e.g. using regex  
            if (auto kind = isKeyword(t.v)) return Token{*kind, t.v};  
            return t;  
        }  
        case '0' ... '9': // ... parse number  
        default: return Token{v.substr(0, 1), Token::ERROR};  
    }  
}
```

## Lexing C??=

```
main() <%
    // yay, this is C99??
    puts("hi\u005Cworld!");
    puts("what's\u005Cup??!");
%>
```

Output: what's up|

- ▶ Trigraphs for systems with more limited encodings/char sets
- ▶ Digraphs to provide a more readable alternative...

# Lexer Implementation

- ▶ Essentially a DFA (for most languages)
  - ▶ Set of regexes → NFA → DFA
- ▶ Respect whitespace/separators for operators, e.g. + and +=
- ▶ Automatic tools (e.g., flex) exist; most compilers do their own
- ▶ Keywords typically parsed as identifiers first
  - ▶ Check identifier if it is a keyword; can use perfect hashing
- ▶ Other practical problems
  - ▶ UTF-8 homoglyphs; trigraphs; pre-processing directives

# Parsing

- ▶ Convert stream of tokens into (abstract) syntax tree
- ▶ Most programming languages are context-sensitive
  - ▶ Variable declarations, argument count, type match, etc.  
~~ separated into semantic analysis
- ▶ Syntactically valid: `void foo = doesntExist / "abc";`
- ▶ Grammar usually specified as CFG

# Context-Free Grammar (CFG)

- ▶ Terminals: basic symbols/tokens
- ▶ Non-terminals: syntactic variables
- ▶ Start symbol: non-terminal defining language
- ▶ Productions: non-terminal  $\rightarrow$  series of (non-)terminals

*stmt*  $\rightarrow$  *whileStmt* | *breakStmt* | *exprStmt*  
*whileStmt*  $\rightarrow$  **while** ( *expr* ) *stmt*  
*breakStmt*  $\rightarrow$  **break** ;  
*exprStmt*  $\rightarrow$  *expr* ;  
*expr*  $\rightarrow$  *expr* + *expr* | *expr* \* *expr* | *expr* = *expr* | ( *expr* ) | **number**

# Hand-written Parsing – First Try

- ▶ One function per non-terminal
- ▶ Check expected structure
- ▶ Return AST node
- ▶ Need look-ahead!

```
NodePtr parseBreakStmt() {  
    consume(Token::BREAK);  
    consume(Token::SEMICOLON);  
    return newNode(Node::BreakStmt);  
}  
  
NodePtr parseWhileStmt() {  
    consume(Token::WHILE);  
    consume(Token::LPAREN);  
    NodePtr expr = parseExpr();  
    consume(Token::RPAREN);  
    NodePtr body = parseStmt();  
    return newNode(Node::WhileStmt,  
                  {expr, body});  
}  
  
NodePtr parseStmt() {  
    // whoops!  
}
```

# Hand-written Parsing – Second Try

- ▶ Need look-ahead to distinguish production rules
- ▶ Consequences for grammar:
  - ▶ No left-recursion
  - ▶ First  $n$  terminals must allow distinguishing rules
  - ▶  $LL(n)$  grammar;  $n$  typically 1
- ⇒ Not all CFGs (easily) parseable (but most programming langs. are)
- ▶ Now... expressions

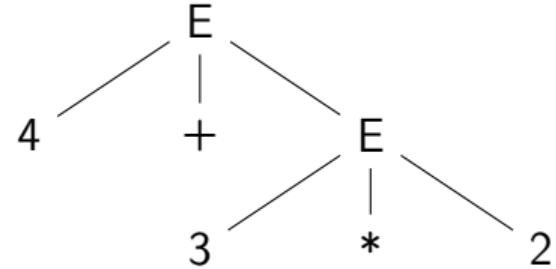
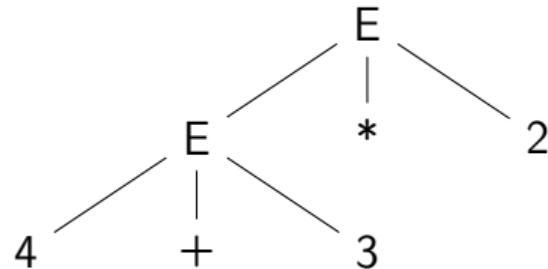
```
NodePtr parseBreakStmt() { /*...*/ }
NodePtr parseWhileStmt() { /*...*/ }

NodePtr parseStmt() {
    Token t = peekToken();
    if (t.kind == Token::BREAK)
        return parseBreakStmt();
    if (t.kind == Token::WHILE)
        return parseWhileStmt();
    // ...
    NodePtr expr = parseExpr();
    consume(Token::SEMICOLON);
    return newNode(Node::ExprStmt,
                  {expr});
}
```

# Ambiguity

$expr \rightarrow expr + expr \mid expr * expr \mid expr = expr \mid ( expr ) \mid number$

Input:  $4 + 3 * 2$

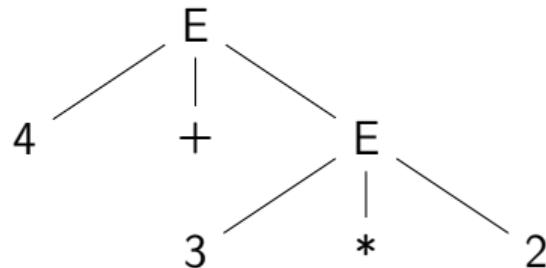


# Ambiguity – Rewrite Grammar?

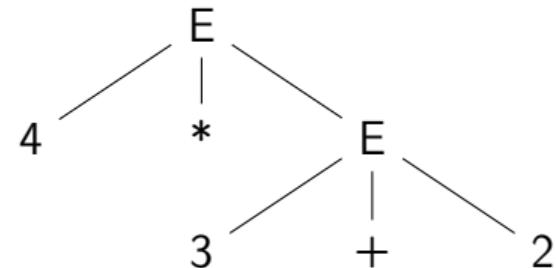
*primary*  $\rightarrow$  ( *expr* ) | *number*

*expr*  $\rightarrow$  *primary* + *expr* | *primary* \* *expr* | *primary* = *expr* | *primary*

Input:  $4 + 3 * 2$

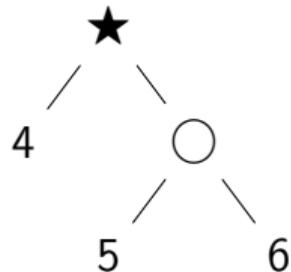


Input:  $4 * 3 + 2$

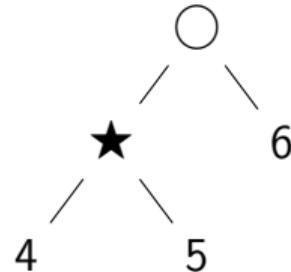


# Ambiguity – Precedence

Input: 4 ★ 5 ○ 6



If  $prec(\bigcirc) > prec(\star)$  or  
equal prec. and  $\star$  is right-assoc.



If  $prec(\bigcirc) < prec(\star)$  or  
equal prec. and  $\star$  is left-assoc.

# Hand-written Parsing – Expression Parsing

- ▶ Start with basic expr.:
- ▶ Number, variable, etc.
- ▶ Parenthesized expr.
  - ▶ Parse full expression
  - ▶ Next token must be )
- ▶ Unary expr: followed by expr. with higher prec.
  - ▶ - < unary - < [] / ->

```
NodePtr parseExpr(unsigned minPrec=0);  
NodePtr parsePrimaryExpr() {  
    switch (Token t = next(); t.kind) {  
        case Token::IDENT:  
            return makeNode(Node::IDENT, t.v);  
        case Token::NUMBER: // ...  
        case Token::MINUS:  
            // Only exprs with high precedence  
            return makeNode(Node::UMINUS,  
                {parseExpr(UNARY_PREC)});  
        case Token::LPAREN: // ...  
            // ...  
    }  
}
```

# Hand-written Parsing – Expression Parsing

- ▶ Only allow ops. with higher prec. on the right child
  - ▶ Right-assoc.: allow same
- ▶ Lower prec.: return + insert higher up in the tree

```
OpDesc OPS[] = { // {prec, rassoc}
    [Token::MUL] = {12, false},
    [Token::ADD] = {11, false},
    [Token::EQ] = {2, true},
    [Token::QUEST] = {3, true}, // ?: }
}
```

```
NodePtr parseExpr(unsigned minPrec=1) {
    auto lhs = parsePrimaryExpr();
    while (auto op = OPS[next().kind];
           op.prec >= minPrec) {
        // ... handle (, [, ?: ...
        auto newPrec = op.rassoc ?
            op.prec : op.prec + 1;
        auto rhs = parseExpr(newPrec);
        lhs = makeNode(op.nodeKind,
                      {lhs, rhs});
    }
    return lhs;
}
```

a = 3 \* 2 + 1;

a = b + c + d = 1;

a ? 1 : b ? 2 : 3;

# Top-down vs. Bottom-up Parsing

## Top-down Parsing

- ▶ Start with top rule
- ▶ Every step: choose expansion
- ▶ LL(1) parser
  - ▶ Left-to-right, Leftmost Derivation
- ▶ “Easily” writable by hand
- ▶ Error handling rather simple
- ▶ Covers many prog. languages

## Bottom-up Parsing

- ▶ Start with text
- ▶ Reduce to non-terminal
- ▶ LR(1) parser
  - ▶ Left-to-right, Rightmost Derivation
  - ▶ Strict super-set of LL(1)
- ▶ Often: uses parser generator
- ▶ Error handling more complex
- ▶ Covers nearly all prog. languages

# Parser Generators

- ▶ Writing parsers by hand can be large effort
- ▶ Parser generators can simplify parser writing a lot
  - ▶ Yacc/Bison, PLY, ANTLR, ...
- ▶ Automatic generation of parser/parsing tables from CFG
  - ▶ Finds ambiguities in the grammar
  - ▶ Lexer often written by hand
- ▶ Used heavily in practice, unless error handling is important

## Bison Example – part 1

```
%define api.pure full
%define api.value.type {ASTNode*}
%param { Lexer* lexer }
%code{
    static int yylex(ASTNode** lvalp, Lexer* lexer);
}

%token NUMBER
%token WHILE "while"
%token BREAK "break"

// precedence and associativity
%right '='
%left '+'
%left '*'
```

## Bison Example – part 2

```
%%
stmt : WHILE '(' expr ')' stmt { $$ = mkNode(WHILE, $1, $2); }
    | BREAK ';' { $$ = mkNode(BREAK, NULL, NULL); }
    | expr ';' { $$ = $1; }
    ;
expr : expr '+' expr { $$ = mkNode('+', $1, $2); }
    | expr '*' expr { $$ = mkNode('*', $1, $2); }
    | expr '=' expr { $$ = mkNode('=', $1, $2); }
    | '(' expr ')' { $$ = $1; }
    | NUMBER
    ;
%%
static int yylex(ASTNode** lvalp, Lexer* lexer) {
    /* return next token, or YYEOF/... */ }
```

# Parsing in Practice

- ▶ Some use parser generators, e.g. Python
- ▶ some use hand-written parsers, e.g. GCC, Clang, Swift, Go
- ▶ Optimization of grammar for performance
  - ▶ Rewrite rules to reduce states, etc.
- ▶ Useful error-handling: complex!
  - ▶ Try skipping to next separator, e.g. ; or ,
- ▶ Programming languages are not always context-free
  - ▶ C: foo\* bar;
  - ▶ May need to break separation between lexer and parser

<sup>7</sup><https://notes.eatonphil.com/parser-generators-vs-handwritten-parsers-survey-2021.html>

# Parsing C++

- ▶ C++ is not context-free (inherited from C): `T * a;`
- ▶ C++ is ambiguous: `Type (a), b;`
  - ▶ Can be a declaration or a comma expression
- ▶ C++ templates are Turing-complete<sup>8</sup>
- ▶ C++ *parsing* is hence *undecidable*<sup>9</sup>
  - ▶ Template instantiation combined with C `T * a` ambiguity

<sup>8</sup>TL Veldhuizen. *C++ templates are Turing complete*. 2003. .

<sup>9</sup>J Haberman. *Parsing C++ is literally undecidable*. 2013. .

# Semantic Analysis

- ▶ Syntactical correctness  $\not\Rightarrow$  correct program  
`void foo = doesntExist / ++"abc";`
- ▶ Needs context-sensitive analysis:
  - ▶ Variable existence, storage, accessibility, ...
  - ▶ Function existence, arguments, ...
  - ▶ Operator type compatibility
  - ▶ Attribute allowance
- ▶ Additional type complexity: inference, polymorphism, ...

# Semantic Analysis: Scope Checking with AST Walking

- ▶ Idea: walk through AST (in DFS-order) and validate on the way
- ▶ Keep track of scope with declared variables
  - ▶ Might need to keep track of defined types separately

## How to implement the scope data structure?

- ▶ For identifiers: check existence and get type
- ▶ For expressions: check types and derive result type
- ▶ For assignment: check lvalue-ness of left side
- ▶ *Might* be possible during AST creation
- ▶ Needs care with built-ins and other special constructs

# Semantic Analysis and Post-Parsing Transformations

- ▶ Check for error-prone code patterns
  - ▶ Completeness of switch, out-of-range constants, unused variables, ...
- ▶ Check method calls, parameter types
- ▶ Duplicate code for templates
- ▶ Make implicit value conversions explicit
- ▶ Handle attributes: visibility, warnings, etc.
- ▶ Mangle names, split functions (OpenMP), ABI-specific setup, ...
- ▶ Last step: generate IR code

# Parsing Performance

Is parsing/front-end performance important?

- ▶ Not necessarily: normal compilers
  - ▶ Some languages (e.g., Rust) need unbounded time *for parsing*
- ▶ Somewhat: JIT compilers
  - ▶ Start-up time is generally noticeable
- ▶ Somewhat more: Developer tools
  - ▶ Imagine: waiting for seconds just for updated syntax highlighting
  - ▶ Often uses tricks like incremental updates to parse tree

# Data Types

- ▶ Important part of programming languages
- ▶ Might have large variety and compatibility
  - ▶ Numbers, Strings, Arrays, Compound Types (struct/union), Enum, Templates, Functions, Pointers, ...
  - ▶ Class hierarchy, Interfaces, Abstract Classes, ...
  - ▶ Integer/float compatibility, promotion, ...
- ▶ Might have implicit conversions

## Data Types: Implementing Classes

- ▶ Simple class/struct: trivial, just bunch of fields
  - ▶ Methods take (pointer to) this as implicit parameter
- ▶ Single inheritance: also trivial – extend struct at end
- ▶ Virtual methods: store vtable in object representation
  - ▶ vtable = table of function pointers for virtual methods
  - ▶ Each sub-class has their own vtable
- ▶ Multiple inheritance is much more involved
- ▶ Dynamic casts: needs run-time type information (RTTI)

# Recommended Lectures

AD IN2227 “Compiler Constructions” covers parsing/analysis in depth

AD CIT3230000 “Programming Languages” covers dispatching/mixins/...

# Compiler Front-end – Summary

- ▶ Lexer splits input into tokens
  - ▶ Essentially Regex-Matching + Keywords; rather simple
- ▶ Parser constructs (abstract) syntax tree from tokens
  - ▶ Top-down vs. bottom-up parsing
  - ▶ Typical: top-down for control flow; bottom-up for expressions
  - ▶ Respect precedence and associativity for operators
- ▶ Semantic analysis ensures meaningful program
- ▶ Some data structures are complex to implement
- ▶ Some programming languages are more difficult to parse

## Compiler Front-end – Questions

- ▶ What are typical components of a compiler front-end?
- ▶ What output does the lexer produce?
- ▶ How does a parser disambiguate rules?
- ▶ What is the typical way to handle operator precedence?
- ▶ Why are not all programming languages describable using CFGs?
- ▶ How to implement classes with virtual functions?

# Code Generation for Data Processing

## Lecture 3: Intermediate Representations

Alexis Engelke

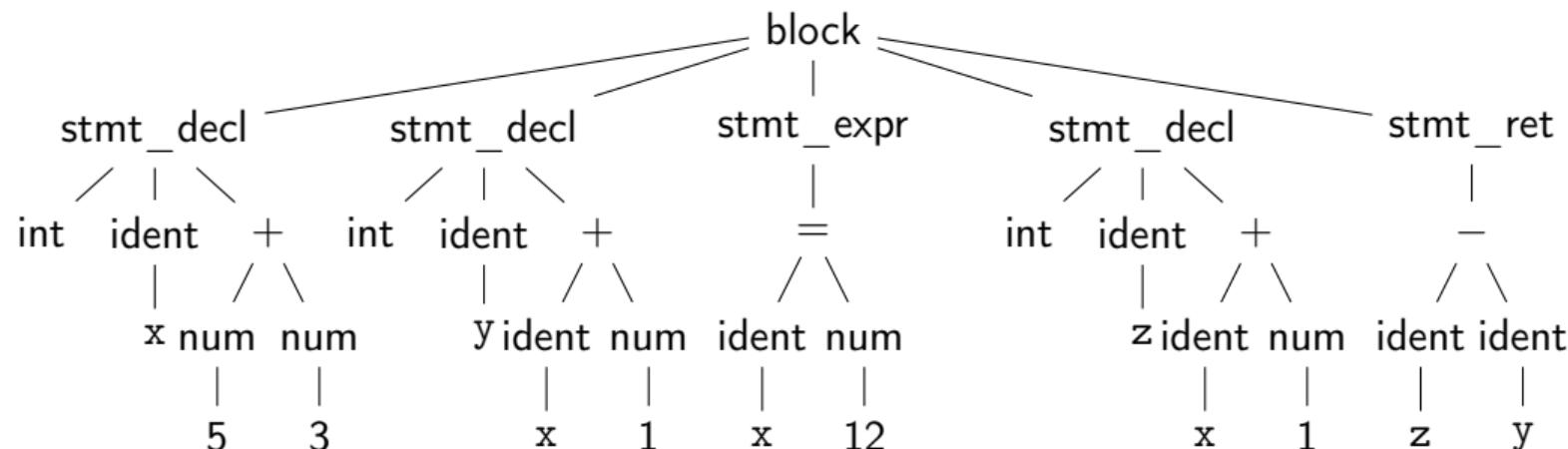
Chair of Data Science and Engineering (I25)  
School of Computation, Information, and Technology  
Technical University of Munich

Winter 2025/26

## Intermediate Representations: Motivation

- ▶ So far: program parsed into AST
- + Great for language-related checks
- + Easy to correlate with original source code (e.g., errors)
- Hard for analyses/optimizations due to high complexity
  - ▶ variable names, control flow constructs, etc.
  - ▶ Data and control flow implicit
- Highly language-specific

# Intermediate Representations: Motivation



Question: how to optimize? Is  $x+1$  redundant?  $\rightsquigarrow$  hard to tell 

## Intermediate Representations: Motivation

```
x1      ← 5  + 3
y1      ← x1 + 1
x2      ← 12
z1      ← x2 + 1
tmp1   ← z1 - y1
return    tmp1
```

Question: how to optimize? Is  $x+1$  redundant?  $\rightsquigarrow$  No! 

# Intermediate Representations

- ▶ Definitive program representation inside compiler
  - ▶ During compilation, only the (current) IR is considered
- ▶ Goal: simplify analyses/transformations
  - ▶ *Technically*, single-step compilation is possible for, e.g., C  
... but optimizations are hard without proper IRs
- ▶ Compilers *design* IRs to support frequent operations
  - ▶ IR design can vary strongly between compilers
- ▶ Typically based on **graphs** or **linear instructions** (or both)

# Compiler Design: Effect of Languages – Imperative

- ▶ Step-by-step execution of program modification of state
- ▶ Close to hardware execution model
- ▶ Direct influence of result
- ▶ Tracking of state is complex
- ▶ Dynamic typing: more complexity
- ▶ Limits optimization possibilities

```
void addvec(int* a, const int* b) {  
    for (unsigned i = 0; i < 4; i++)  
        a[i] += b[i]; // vectorizable?  
}
```

func:

```
    mov [rdi], rsi  
    mov [rdi+8], rdx  
    mov [rdi], 0 // redundant?  
    ret
```

# Compiler Design: Effect of Languages – Declarative

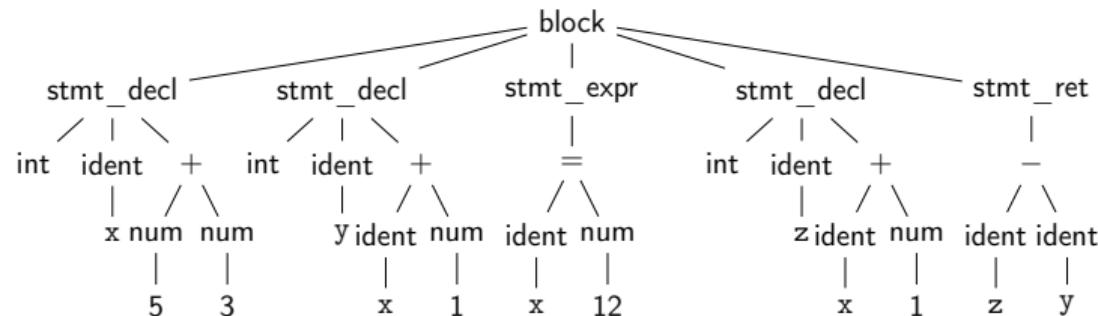
- ▶ Describes execution target
- ▶ Compiler has to derive good mapping to imperative hardware
- ▶ Allows for more optimizations
- ▶ Mapping to hardware non-trivial
  - ▶ Might need more stages
  - ▶ Preserve semantic info for opt!
- ▶ Programmer has less “control”

```
select s.name
from studenten s
where exists (select 1
               from hoeren h
               where h.matrno=s.matrno)
```

```
let rec fac = function
| 0 | 1 -> 1
| n -> n * fac (n - 1)
```

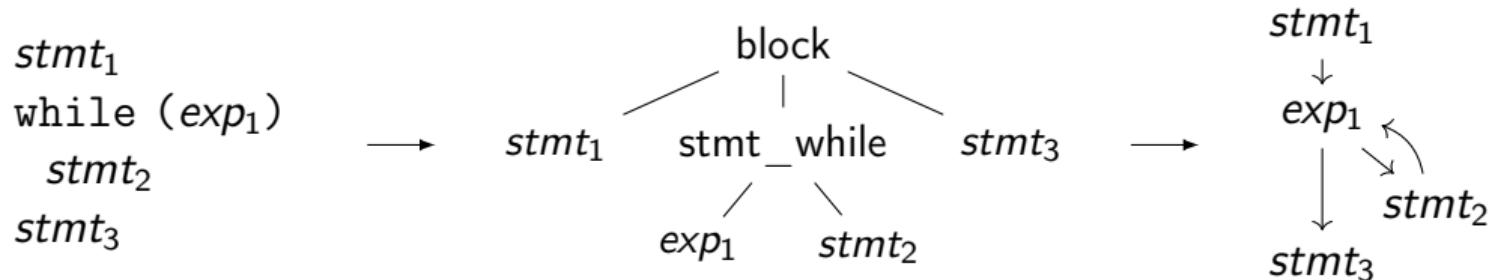
# Graph IRs: Abstract Syntax Tree (AST)

- ▶ Code representation close to the source
- ▶ Representation of types, constants, etc. might differ
- ▶ Storage might be problematic for large inputs



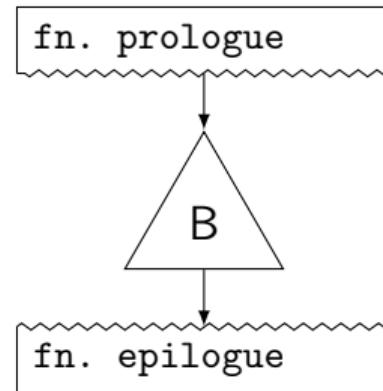
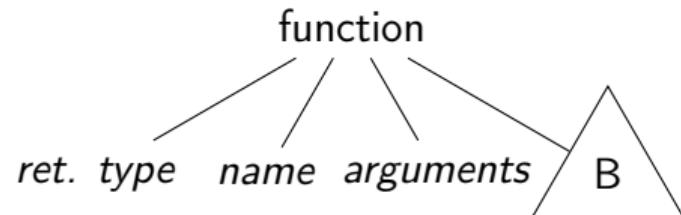
# Graph IRs: Control Flow Graph (CFG)

- ▶ Motivation: model control flow between different code sections
- ▶ Graph nodes represent **basic blocks**
  - ▶ Basic block: sequence of branch-free code (modulo exceptions)
  - ▶ Typically represented using a linear IR

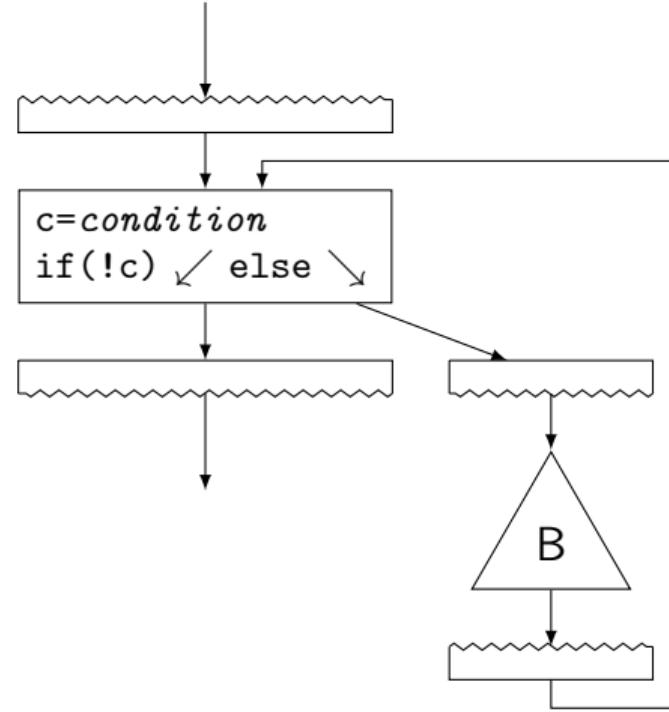
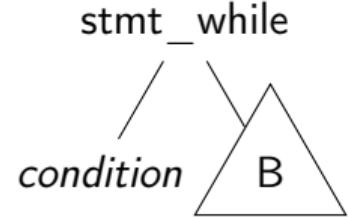


# Build CFG from AST – Function

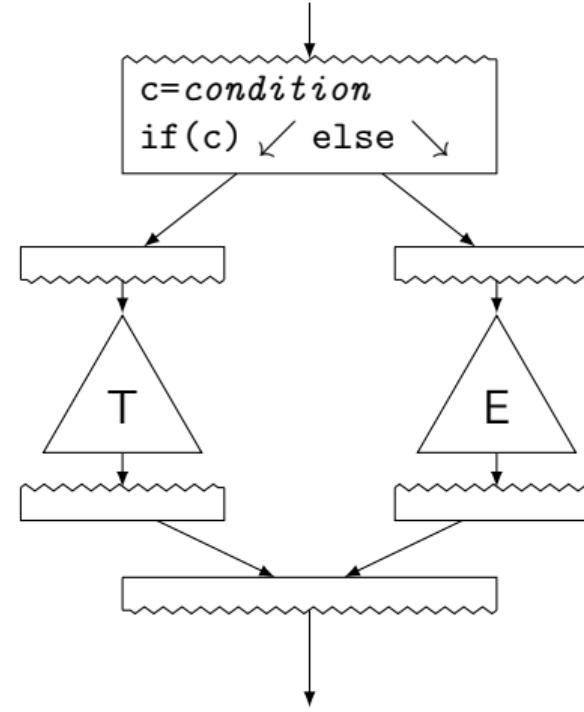
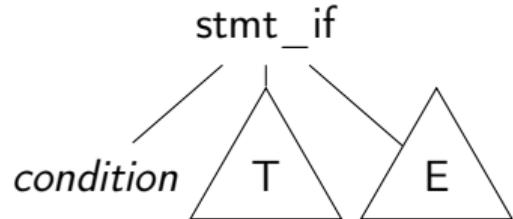
- ▶ Idea: Keep track of current insert block while walking through AST



# Build CFG from AST – While Loop



# Build CFG from AST – If Condition



# Build CFG from AST: Switch

## Linear search

```
t ← exp  
if t == 3: goto B3  
if t == 4: goto B4  
if t == 7: goto B7  
if t == 9: goto B9  
goto BD
```

+ Trivial

- Slow, lot of code

## Binary search

```
t ← exp  
if t == 7: goto B7  
elif t > 7:  
    if t == 9: goto B9  
else:  
    if t == 3: goto B3  
    if t == 4: goto B4  
goto BD
```

+ Good: sparse  
values

- Even more code

## Jump table

```
t ← exp  
if 0 ≤ t < 10:  
    goto table[t]  
goto BD
```

table = {  
 B<sub>D</sub>, B<sub>D</sub>, B<sub>D</sub>, B<sub>3</sub>,  
 B<sub>4</sub>, B<sub>D</sub>, ... }

+ Fastest

- Table can be large,  
needs ind. jump

## Build CFG from AST: Break, Continue, Goto, Computed Goto

- ▶ `break/continue`: trivial
  - ▶ Keep track of target block, insert branch
- ▶ `goto`: also trivial
  - ▶ Split block at target label, if needed
  - ▶ But: may lead to irreducible control flow graph (see later)
- ▶ `Computed goto`: trivial is bad
  - ▶ Split block at target label, if needed
  - ▶ Every computed goto is a branch to every address-taken label
  - ▶ CFG can grow extremely dense for dispatch code!

## CFG: Formal Definition

- ▶ **Flow graph:**  $G = (N, E, s)$  with a digraph  $(N, E)$  and entry  $s \in N$ 
  - ▶ Each node is a basic block,  $s$  is the entry block
  - ▶  $(n_1, n_2) \in E$  iff  $n_2$  might be executed immediately after  $n_1$
  - ▶ All  $n \in N$  shall be reachable from  $s$  (unreachable nodes can be discarded)
  - ▶ Nodes without successors are end points

# CFG from C – Example

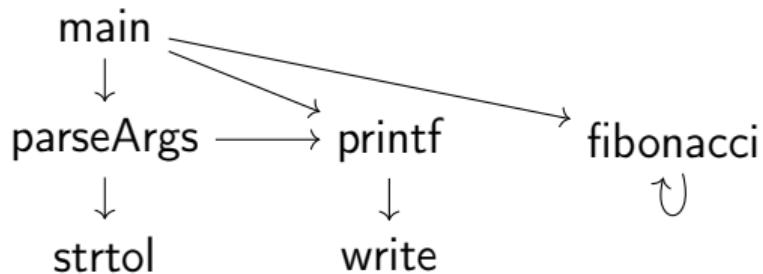
Derive the CFG for the these functions. Assume a switch instruction exists.

```
int fn1() {  
    if (a()) {  
        while (b()) {  
            c();  
            if (d())  
                continue;  
            e();  
        }  
    } else {  
        f();  
    }  
    return g();  
}
```

```
int fn2() {  
    a();  
    do switch (c()) {  
        case 1:  
            while (d()) {  
                e();  
            }  
        case 2:  
            f();  
        }  
        default:  
            g();  
    } while (h());  
    return b();  
}
```

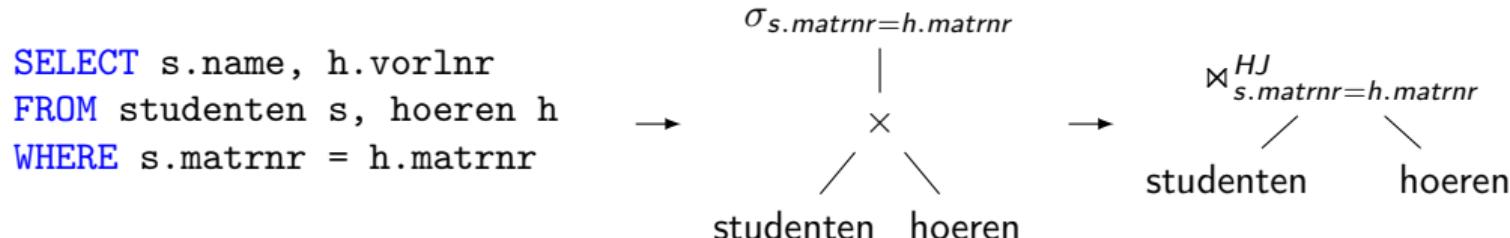
# Graph IRs: Call Graph

- ▶ Graph showing (possible) call relations between functions
- ▶ Useful for interprocedural optimizations
  - ▶ Function ordering
  - ▶ Stack depth estimation
  - ▶ ...



# Graph IRs: Relational Algebra

- ▶ Higher-level representation of query plans
  - ▶ Explicit data flow
- ▶ Allow for optimization and selection actual implementations
  - ▶ Elimination of common sub-trees
  - ▶ Joins: ordering, implementation, etc.



# Linear IRs: Stack Machines

- ▶ Operands stored on a stack
- ▶ Operations pop arguments from top and push result
- ▶ Typically accompanied with variable storage
- ▶ Generating IR from AST: trivial
- ▶ Often used for bytecode, e.g. Java, Python

+

- Compact code, easy to generate and implement
- Performance, hard to analyze

```
push 5
push 3
add
pop x
push x
push 1
add
pop y
push 12
pop x
push x
push 1
add
pop z
```

# Linear IRs: Register Machines

- ▶ Operands stored in registers
- ▶ Operations read and write registers
- ▶ Typically: infinite number of registers
- ▶ Typically: three-address form
  - ▶  $dst = src1 \ op \ src2$
- ▶ Generating IR from AST: trivial
- ▶ E.g., GIMPLE, eBPF, Assembly

```
x ← 5 + 3
y ← x + 1
x ← 12
z ← x + 1
tmp1 ← z - y
return tmp1
```

## Example: High GIMPLE

```
int foo(int n) {
    int res = 1;
    while (n) {
        res *= n * n;
        n -= 1;
    }
    return res;
}

int fac (int n)
gimple_bind < // <-- still has lexical scopes
    int D.1950;
    int res;

    gimple_assign <integer_cst, res, 1, NULL, NULL>
    gimple_goto <<D.1947>>
    gimple_label <<D.1948>>
    gimple_assign <mult_expr, _1, n, n, NULL>
    gimple_assign <mult_expr, res, res, _1, NULL>
    gimple_assign <plus_expr, n, n, -1, NULL>
    gimple_label <<D.1947>>
    gimple_cond <ne_expr, n, 0, <D.1948>, <D.1946>>
    gimple_label <<D.1946>>
    gimple_assign <var_decl, D.1950, res, NULL, NULL>
    gimple_return <D.1950>
>
```

## Example: Low GIMPLE

```
int fac (int n)
{
    int res;
    int D.1950;

int foo(int n) {
    int res = 1;
    while (n) {
        res *= n * n;
        n -= 1;
    }
    return res;
}

    gimple_assign <integer_cst, res, 1, NULL, NULL>
    gimple_goto <<D.1947>>
    gimple_label <<D.1948>>
    gimple_assign <mult_expr, _1, n, n, NULL>
    gimple_assign <mult_expr, res, res, _1, NULL>
    gimple_assign <plus_expr, n, n, -1, NULL>
    gimple_label <<D.1947>>
    gimple_cond <ne_expr, n, 0, <D.1948>, <D.1946>>
    gimple_label <<D.1946>>
    gimple_assign <var_decl, D.1950, res, NULL, NULL>
    gimple_goto <<D.1951>>
    gimple_label <<D.1951>>
    gimple_return <D.1950>
}
```

```
$ gcc -fdump-tree-lower-raw -c foo.c
```

## Example: Low GIMPLE with CFG

```
int fac (int n) {
    int res;
    int D.1950;
    <bb 2> :
    gimple_assign <integer_cst, res, 1, NULL, NULL>
    goto <bb 4>; [INV]
    <bb 3> :
    gimple_assign <mult_expr, _1, n, n, NULL>
    gimple_assign <mult_expr, res, res, _1, NULL>
    gimple_assign <plus_expr, n, n, -1, NULL>
    <bb 4> :
    gimple_cond <ne_expr, n, 0, NULL, NULL>
        goto <bb 3>; [INV]
    else
        goto <bb 5>; [INV]
    <bb 5> :
    gimple_assign <var_decl, D.1950, res, NULL, NULL>
    <bb 6> :
    gimple_label <<L3>>
    gimple_return <D.1950>
}

int foo(int n) {
    int res = 1;
    while (n) {
        res *= n * n;
        n -= 1;
    }
    return res;
}
```

```
$ gcc -fdump-tree-cfg-raw -c foo.c
```

# Linear IRs: Register Machines

- ▶ Problem: no clear def–use information

- ▶ Is  $x + 1$  the same?
- ▶ Hard to track actual values!

- ▶ How to optimize?

⇒ Disallow mutations of variables

```
x      ← 5      + 3
y      ← x      + 1
x      ← 12
z      ← x      + 1
tmp1 ← z      - y
return      tmp1
```

# Single Static Assignment: Introduction

- ▶ Idea: disallow mutations of variables, value set in declaration
- ▶ Instead: create new variable for updated value
- ▶ SSA form: every computed value has a unique definition
  - ▶ Equivalent formulation: each name describes result of one operation

|                          |   |                            |
|--------------------------|---|----------------------------|
| $x \leftarrow 5 + 3$     |   | $v_1 \leftarrow 5 + 3$     |
| $y \leftarrow x + 1$     |   | $v_2 \leftarrow v_1 + 1$   |
| $x \leftarrow 12$        | → | $v_3 \leftarrow 12$        |
| $z \leftarrow x + 1$     |   | $v_4 \leftarrow v_3 + 1$   |
| $tmp_1 \leftarrow z - y$ |   | $v_5 \leftarrow v_4 - v_2$ |
| return $tmp_1$           |   | return $v_5$               |

# Single Static Assignment: Control Flow

- ▶ How to handle diverging values in control flow?
- ▶ Solution:  $\Phi$ -nodes to merge values depending on predecessor
  - ▶ Value depends on edge used to enter the block
  - ▶ All  $\Phi$ -nodes of a block execute concurrently (ordering irrelevant)

```
entry : x ← ...
       if (x > 2) goto cont
then :  x ← x * 2
cont : return x
```



```
entry : v1 ← ...
       if (v1 > 2) goto cont
then :  v2 ← v1 * 2
cont : v3 ←  $\Phi$ (entry : v1, then : v2)
       return v3
```

## Example: GIMPLE in SSA form

```
int fac (int n) { int res, D.1950, _1, _6;
  <bb 2> :
  gimple_assign <integer_cst, res_4, 1, NULL, NULL>
  goto <bb 4>; [INV]
  <bb 3> :
  gimple_assign <mult_expr, _1, n_2, n_2, NULL>
  gimple_assign <mult_expr, res_8, res_3, _1, NULL>
  gimple_assign <plus_expr, n_9, n_2, -1, NULL>
  <bb 4> :
  # gimple_phi <n_2, n_5(D)(2), n_9(3)>
  # gimple_phi <res_3, res_4(2), res_8(3)>
  gimple_cond <ne_expr, n_2, 0, NULL, NULL>
    goto <bb 3>; [INV]
  else
    goto <bb 5>; [INV]
  <bb 5> :
  gimple_assign <ssa_name, _6, res_3, NULL, NULL>
  <bb 6> :
  gimple_label <<L3>>
  gimple_return <_6>
}

$ gcc -fdump-tree-ssa-raw -c foo.c
```

# SSA Construction – Local Value Numbering

- ▶ Simple case: inside block – keep mapping of variable to value

| Code                     | SSA IR                                | Variable Mapping        |
|--------------------------|---------------------------------------|-------------------------|
| $x \leftarrow 5 + 3$     | $v_1 \leftarrow \text{add } 5, 3$     | $x \rightarrow v_3$     |
| $y \leftarrow x + 1$     | $v_2 \leftarrow \text{add } v_1, 1$   | $y \rightarrow v_2$     |
| $x \leftarrow 12$        | $v_3 \leftarrow \text{const } 12$     | $z \rightarrow v_4$     |
| $z \leftarrow x + 1$     | $v_4 \leftarrow \text{add } v_3, 1$   | $tmp_1 \rightarrow v_5$ |
| $tmp_1 \leftarrow z - y$ | $v_5 \leftarrow \text{sub } v_4, v_2$ |                         |
| return $tmp_1$           | ret $v_5$                             |                         |

## SSA Construction – Across Blocks

- ▶ SSA construction with control flow is non-trivial
- ▶ Key problem: find value for variable in predecessor
- ▶ Naive approach:  $\Phi$ -nodes for all variables everywhere
  - ▶ Create empty  $\Phi$ -nodes for variables, populate variable mapping
  - ▶ Fill blocks (as on last slide)
  - ▶ Fill  $\Phi$ -nodes with last value of variable in predecessor
- ▶ Why is this a bad idea?  $\Rightarrow$  *don't do this!*
  - ▶ *Extremely inefficient, code size explosion, many dead  $\Phi$*

## SSA Construction – Across Blocks (“simple”<sup>10</sup>)

- ▶ Key problem: find value in predecessor
- ▶ Idea: seal block once all direct predecessors are known
  - ▶ For acyclic constructs: trivial
  - ▶ For loops: seal header once loop block is generated
- ▶ Current block not sealed: add  $\Phi$ -node, fill on sealing
- ▶ Single predecessor: recursively query that
- ▶ Multiple preds.: add  $\Phi$ -node, fill now

<sup>10</sup>M Braun et al. “Simple and efficient construction of static single assignment form”. In: CC. 2013, pp. 102–122. 

## SSA Construction – Example

```
func foo(v1)
entry: sealed; varmap: n→v1, res→v2
v2 ← 1

header: sealed; varmap: n→φ1, res→φ2
φ1 ← φ(entry: v1, body: v6)
φ2 ← φ(entry: v2, body: v5)
v3 ← equal φ1, 0
br v3, cont, body

int foo(int n) {
    int res = 1;
    while (n) {
        res *= n * n;
        n -= 1;
    }
    return res;
}
body: sealed; varmap: n→v6, res→v5
v4 ← mul φ1, φ1
v5 ← mul φ2, v4
v6 ← sub φ1, 1
br header

cont: sealed; varmap: res→φ2
ret φ2
```

# SSA Construction – Example

Construct an IR in SSA form for the following C functions.

```
int phis(int a, int b){  
    a = a * b;  
    if (a > b * b) {  
        int c = 1;  
        while (a > 0)  
            a = a - c;  
    } else {  
        a = b * b;  
    }  
    return a;  
}  
  
int swap(int a, int b, int c) {  
    while (c > 0) {  
        int tmp = a;  
        a = b;  
        b = tmp;  
        c = c - 1;  
    }  
    return a;  
}
```

# SSA Construction – Pruned/Minimal Form

- ▶ Resulting SSA is *pruned* – all  $\phi$  are used
- ▶ But not *minimal* –  $\phi$  nodes might have single, unique value
- ▶ When filling  $\phi$ , check that multiple real values exist
  - ▶ Otherwise: replace  $\phi$  with the single value
  - ▶ On replacement, update all  $\phi$  using this value, they might be trivial now, too
- ▶ Sufficient? Not for irreducible CFG
  - ▶ Needs more complex algorithms<sup>11</sup> or different construction method<sup>12</sup>

AD

IN2053 “Program Optimization” covers this more formally

<sup>11</sup> M Braun et al. “Simple and efficient construction of static single assignment form”. In: *CC*. 2013, pp. 102–122. .

<sup>12</sup> R Cytron et al. “Efficiently computing static single assignment form and the control dependence graph”. In: *TOPLAS* 13.4 (1991), pp. 451–490. .

## SSA: Implementation

- ▶ Value is often just a reference to the instruction
- ▶  $\phi$  nodes placed at beginning of block
  - ▶ They execute “concurrently” and on the edges, after all
- ▶ Variable number of operands required for  $\phi$  nodes
- ▶ Storage format for instructions and basic blocks
  - ▶ Consecutive in memory: hard to modify/traverse
  - ▶ Array of pointers:  $\mathcal{O}(n)$  for a single insertion...
  - ▶ Linked List: easy to insert, but pointer overhead

# Is SSA a graph IR?

Only if instructions have no side effects,  
consider load, store, call, ...

These *can* be solved using explicit dependencies as SSA values, e.g. for memory

# IR Design: High-level Considerations

- ▶ Define purpose!
- ▶ Structure: SSA vs. something else; control flow
  - ▶ Control flow: basic blocks/CFG vs. structured control flow
  - ▶ Remember: SSA can be considered as a DAG, too
  - ▶ SSA is easy to analyse, but non-trivial to construct/leave
- ▶ Broader integration: keep multiple stages in single IR?
  - ▶ Example: create IR with high-level operations, then incrementally lower
  - ▶ Model machine instructions in same IR?
  - ▶ Can avoid costly transformations, but adds complexity

# IR Design: Operations

- ▶ Data types
  - ▶ Simple type structure vs. complex/aggregate types?
  - ▶ Keep relation to high-level types vs. low-level only?
  - ▶ Virtual data types, e.g. for flags/memory?
- ▶ Instruction format
  - ▶ Single vs. multiple results?
  - ▶ Strongly typed vs. more generic result/operand types?
  - ▶ Operand number – fixed vs. dynamic?

# IR Design: Operations

- ▶ Allow instruction side effects?
  - ▶ E.g.: memory, floating-point arithmetic, implicit control flow
- ▶ Operation complexity and abstraction
  - ▶ E.g.: CheckBounds, GetStackPtr, HashInt128
  - ▶ E.g.: load vs. MOVQconstidx4
- ▶ Extensibility for new operations (e.g., new targets, high-level ops)

## IR Design: Desired Operations on IR

- ▶ Replacing all uses of an instruction with a different value
- ▶ Inserting an instruction or phi node/block argument
- ▶ Removing an instruction or phi node/block argument
- ▶ Changing the operand of an instruction
- ▶ Finding predecessors and successors of a basic block
- ▶ Finding all users of an instruction result
  
- ▶ For optimization-focused IRs, all these operations should be fast

# IR Design: Implementation

- ▶ Maintain user lists?
  - ▶ Simplifies optimizations, but adds considerable overhead
  - ▶ Replacement can use copy and lazy canonicalization
  - ▶ User *count* might be sufficient alternative
- ▶ Storage layout: operation size and locations
  - ▶ For performance: reduce heap allocations, small data structures
- ▶ Special handling for arguments vs. all-instructions?
- ▶ Metadata for source location, register allocation, etc.
- ▶ SSA:  $\phi$  nodes vs. block arguments?

# IR Example: Go SSA

- ▶ Strongly typed
  - ▶ Structured types decomposed
- ▶ Explicit memory side-effects
- ▶ Also High-level operations
  - ▶ IsInBounds, VarDef
- ▶ Only one type of value/instruction
  - ▶ Const64, Arg, Phi
- ▶ No user list, but user count
- ▶ Also used for arch-specific repr.

```
env GOSSAFUNC=fac go build test.go
```

```
b1:  
  v1 (?) = InitMem <mem>  
  v2 (?) = SP <uintptr>  
  v5 (?) = LocalAddr <*int> {~r1} v2 v1  
  v6 (7) = Arg <int> {n} (n[int])  
  v8 (?) = Const64 <int> [1] (res[int])  
  v9 (?) = Const64 <int> [2] (i[int])  
Plain -> b2 (+9)  
b2: <- b1 b4  
  v10 (9) = Phi <int> v9 v17 (i[int])  
  v23 (12) = Phi <int> v8 v15 (res[int])  
  v12 (+9) = Less64 <bool> v10 v6  
If v12 -> b4 b5 (likely) (9)  
b4: <- b2  
  v15 (+10) = Mul64 <int> v23 v10 (res[int])  
  v17 (+9) = Add64 <int> v10 v8 (i[int])  
Plain -> b2 (9)  
b5: <- b2  
  v20 (12) = VarDef <mem> {~r1} v1  
  v21 (+12) = Store <mem> {int} v5 v23 v20  
Ret v21 (+12)
```

## Intermediate Representations – Summary

- ▶ An IR is an internal representation of a program
- ▶ Main goal: simplify analyses and transformations
- ▶ IRs typically based on graphs or linear instructions
- ▶ Graph IRs: AST, Control Flow Graph, Relational Algebra
- ▶ Linear IRs: stack machines, register machines, SSA
- ▶ Single Static Assignment makes data flow explicit
- ▶ SSA is extremely popular, although non-trivial to construct
- ▶ IR design depends on purpose and integration constraints

## Intermediate Representations – Questions

- ▶ Who designs an IR? What are design criteria?
- ▶ Why is an AST not suited for program optimization?
- ▶ How to convert an AST to another IR?
- ▶ What are the benefits/drawbacks of stack/register machines?
- ▶ What benefits does SSA offer over a normal register machine?
- ▶ How do  $\phi$ -instructions differ from normal instructions?

# Code Generation for Data Processing

## Lecture 4: LLVM-IR

Alexis Engelke

Chair of Data Science and Engineering (I25)  
School of Computation, Information, and Technology  
Technical University of Munich

Winter 2025/26

# LLVM<sup>13</sup>

## LLVM “Core” Library

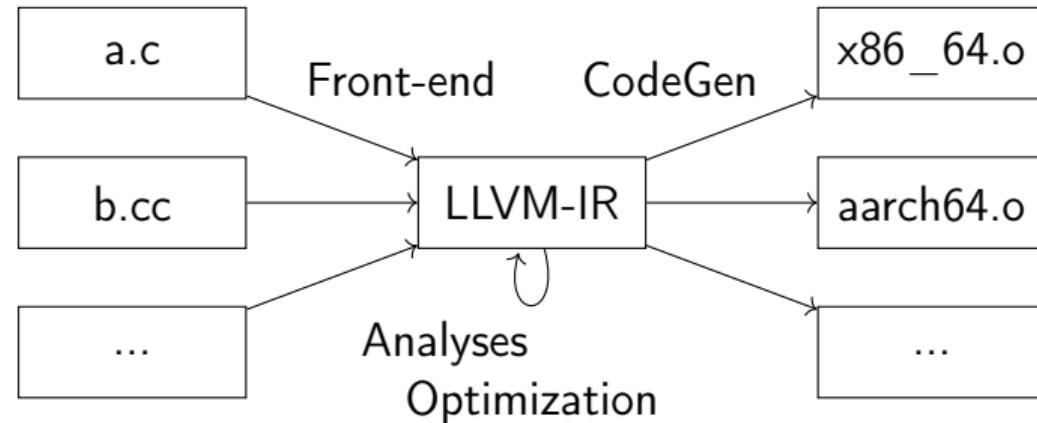
- ▶ Optimizer and compiler back-end
- ▶ “Set of compiler components”
  - ▶ IRs: LLVM-IR, SelDag, MIR
  - ▶ Analyses and Optimizations
  - ▶ Code generation back-ends
- ▶ Started from Chris Lattner’s master’s thesis
- ▶ Used for C, C++, Swift, D, Julia, Rust, Haskell, ...

## LLVM Project

- ▶ Umbrella for several projects related to compilers/toolchain
  - ▶ LLVM Core
  - ▶ Clang: C/C++ front-end for LLVM
  - ▶ libc++, compiler-rt: runtime support
  - ▶ LLDB: debugger
  - ▶ LLD: linker
  - ▶ MLIR: experimental IR framework

<sup>13</sup> C Lattner and V Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: CGO. 2004, pp. 75–86. 

# LLVM: Overview



- ▶ Independent front-end derives LLVM-IR, LLVM does opt. and code gen.
- ▶ LTO: dump LLVM-IR into object file, optimize at link-time

# LLVM-IR: Overview

- ▶ SSA-based IR, representations  
textual, bitcode, in-memory
- ▶ Hierarchical structure
  - ▶ Module
  - ▶ Functions, global variables
  - ▶ Basic blocks
  - ▶ Instructions
- ▶ Strongly/strictly typed

```
define dso_local i32 @foo(i32 %0) {  
    %2 = icmp eq i32 %0, 0  
    br i1 %2, label %10, label %3  
  
3: ; preds = %1, %3  
    %4 = phi i32 [ %7, %3 ], [ 1, %1 ]  
    %5 = phi i32 [ %8, %3 ], [ %0, %1 ]  
    %6 = mul nsw i32 %5, %5  
    %7 = mul nsw i32 %6, %4  
    %8 = add nsw i32 %5, -1  
    %9 = icmp eq i32 %8, 0  
    br i1 %9, label %10, label %3  
  
10: ; preds = %3, %1  
    %11 = phi i32 [ 1, %1 ], [ %7, %3 ]  
    ret i32 %11  
}
```

# LLVM-IR: Data types

- ▶ First class types:
  - ▶  $i\langle N \rangle$  – arbitrary bit width integer, e.g.  $i1$ ,  $i25$ ,  $i1942652$
  - ▶  $\text{ptr}/\text{ptr } \text{addrspace}(1)$  – pointer with optional address space
  - ▶  $\text{float}/\text{double}/\text{half}/\text{bfloat}/\text{fp128}/\dots$
  - ▶  $\langle N \times \text{ty} \rangle$  – vector type, e.g.  $\langle 4 \times i32 \rangle$
- ▶ Aggregate types:
  - ▶  $[N \times \text{ty}]$  – constant-size array type, e.g.  $[32 \times \text{float}]$
  - ▶  $\{ \text{ty}, \dots \}$  – struct (can be packed/opaque), e.g.  $\{i32, \text{float}\}$
- ▶ Other types:
  - ▶  $\text{ty}(\text{ty}, \dots)$  – function type, e.g.  $\{i32, i32\}(\text{ptr}, \dots)$
  - ▶  $\text{void}$
  - ▶ label/token/metadata

# LLVM-IR: Modules

- ▶ Top-level entity, one compilation unit – akin to C/C++
- ▶ Contains global values, specified with linkage type
- ▶ Global variable declarations/definitions

```
@externInt = external global i32, align 4
@globVar = global i32 4, align 4
@staticPtr = internal global ptr null, align 8
```

- ▶ Function declarations/definitions

```
declare i32 @readPtr(ptr)
define i32 @return1() {
    ret i32 1
}
```

- ▶ Global named metadata (discarded during compilation)

## LLVM-IR: Functions

- ▶ Functions definitions contain all code, not nestable
- ▶ Single return type (or `void`), multiple parameters, list of basic blocks
  - ▶ No basic blocks ⇒ function declaration
- ▶ Specifiers for `callconv`, section name, other attributes
  - ▶ E.g.: `noinline/alwaysinline`, `noreturn`, `readonly`
- ▶ Parameter and return can also have attributes
  - ▶ E.g.: `noalias`, `nonnull`, `sret(<ty>)`

## LLVM-IR: Basic Block

- ▶ Sequence of instructions
  - ▶  $\phi$  nodes come first
  - ▶ Regular instructions come next
  - ▶ Must end with a terminator
- ▶ First block in function is entry block  
Entry block cannot be branch target

## LLVM-IR: Instructions – Control Flow and Terminators

- ▶ Terminators end a block/modify control flow
- ▶ `ret <ty> <val>/ret void`
- ▶ `br label <dest>/br i1 <cond>, label <then>, label <else>`
- ▶ `switch/indirectbr`
- ▶ `unreachable`
- ▶ Few others for exception handling
- ▶ Not a terminator: `call`

# LLVM-IR: Instructions – Arithmetic-Logical

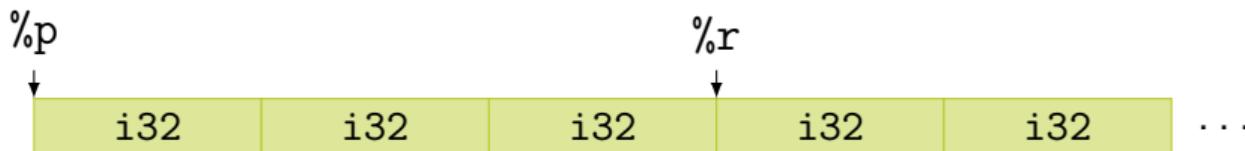
- ▶ add/sub/mul/udiv/sdiv/urem/srem
  - ▶ Arithmetic uses two's complement
  - ▶ Division corner cases are *undefined behavior*
- ▶ fneg/fadd/fsub/fmul/fdiv/frem
- ▶ shl/lshr/ashr/and/or/xor
  - ▶ Out-of-range shifts have an undefined result
- ▶ icmp <pred>/fcmp <pred>/select <cond>, <then>, <else>
- ▶ trunc/zext/sext/fptrunc/fpext/fptoui/fptosi/uitofp/sitofp
- ▶ bitcast
  - ▶ Cast between equi-sized datatypes by reinterpreting bits

## LLVM-IR: Instructions – Memory and Pointer

- ▶ `alloca <ty>` – allocate addressable stack slot
- ▶ `load <ty>, ptr <ptr>/store <ty> <val>, ptr <ptr>`
  - ▶ May be `volatile` (e.g., MMIO) and/or `atomic`
- ▶ `cmpxchg/atomicrmw` – similar to hardware operations
- ▶ `ptrtoint/inttoptr`
- ▶ `getelementptr` – address computation on `ptr/structs/arrays`

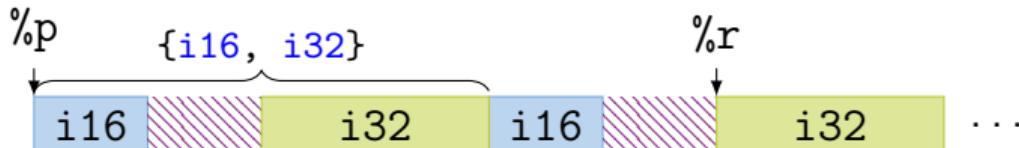
## LLVM-IR: getelementptr Examples

▶ `%r = getelementptr i32, ptr %p, i64 3`



Equivalent in C: `&((int*) p)[3]`

▶ `%r = getelementptr {i16, i32}, ptr %p, i64 1, i32 1`



Equivalent in C: `&((struct {short _0; int _1;}*) p)[1]._1`

▶ Also works with nested structs and arrays (and vectors)

▶ Pointer and array can be dynamic, struct indices must be constant

# LLVM-IR: undef and poison

- ▶ undef – unspecified value, compiler may choose any value
  - ▶ `%b = add i32 %a, i32 undef → i32 undef`
  - ▶ `%c = and i32 %a, i32 undef → i32 %a`
  - ▶ `%d = xor i32 %b, i32 %b → i32 undef`
  - ▶ `br i1 undef, label %p, label %q → undefined behavior`
- ▶ poison – result of erroneous operations
  - ▶ Delay *undefined behavior* on illegal operation until actually relevant
  - ▶ Allows to speculatively “execute” instructions in IR
  - ▶ `%d = shl i32 %b, i32 34 → i32 poison`

# LLVM-IR: Intrinsics

- ▶ Not all operations provided as instructions
- ▶ Intrinsic functions: special functions with defined semantics
  - ▶ Replaced during compilation, e.g., with instruction or lib call
- ▶ Benefit: no changes needed for parser/bitcode/... on addition
- ▶ Examples:
  - ▶ `declare iN @llvm.ctpop.iN(iN <src>)`
  - ▶ `declare {iN, i1} @llvm.sadd.with.overflow.iN(iN %a, iN %b)`
  - ▶ `memcpy, memset, sqrt, returnaddress, ...`

## LLVM-IR: Tools

- ▶ clang can emit LLVM-IR bitcode

```
clang -O -emit-llvm -c test.c -o test.bc
```

- ▶ llvm-dis disassembles bitcode to textual LLVM-IR

```
clang -O -emit-llvm -c test.c -o - | llvm-dis
```

- ▶ llc compiles LLVM-IR (textual or bitcode) to assembly

```
clang -O -emit-llvm -c test.c -o - | llc
```

```
clang -O -emit-llvm -c test.c -o - | llvm-dis | llc
```

Example Listings omitted – they would span several slides

# LLVM-IR: Example

```
define <4 x float> @foo2(<4 x float> %0, <4 x float> %1) {
  %3 = alloca <4 x float>, align 16
  %4 = alloca <4 x float>, align 16
  store <4 x float> %0, ptr %3, align 16
  store <4 x float> %1, ptr %4, align 16
  %5 = load <4 x float>, ptr %3, align 16
  %6 = load <4 x float>, ptr %4, align 16
  %7 = fadd <4 x float> %5, %6
  ret <4 x float> %7
}
```

# LLVM-IR: Example

```
define i32 @foo3(i32 %0, i32 %1) {
  %3 = tail call { i32, i1 } @llvm.smul.with.overflow.i32(i32 %0, i32 %1)
  %4 = extractvalue { i32, i1 } %3, 1
  %5 = extractvalue { i32, i1 } %3, 0
  %6 = select i1 %4, i32 -2147483648, i32 %5
  ret i32 %6
}
```

# LLVM-IR: Example

```
define i32 @sw(i32 %0) {
  switch i32 %0, label %4 [
    i32 4, label %5
    i32 5, label %2
    i32 8, label %3
    i32 100, label %5
  ]
  2: ; preds = %1
    br label %5
  3: ; preds = %1
    br label %5
  4: ; preds = %1
    br label %5
  5: ; preds = %1, %1, %4, %3, %2
    %6 = phi i32 [ %0, %4 ], [ 9, %3 ], [ 32, %2 ], [ 12, %1 ], [ 12, %1 ]
    ret i32 %6
}
```

# LLVM-IR: Example

```
@a = private unnamed_addr constant [7 x i32] [i32 12, i32 32, i32 12,
                                         i32 12, i32 9, i32 12, i32 12], align 4
define dso_local i32 @f(i32 %0) {
    %2 = add i32 %0, -4
    %3 = icmp ult i32 %2, 7
    br i1 %3, label %4, label %13
4: ; preds = %1
    %5 = trunc i32 %2 to i8
    %6 = lshr i8 83, %5
    %7 = and i8 %6, 1
    %8 = icmp eq i8 %7, 0
    br i1 %8, label %13, label %9
9: ; preds = %4
    %10 = sext i32 %2 to i64
    %11 = getelementptr inbounds [7 x i32], ptr @a, i64 0, i64 %10
    %12 = load i32, ptr %11, align 4
    br label %13
13: ; preds = %1, %4, %9
    %14 = phi i32 [ %12, %9 ], [ %0, %4 ], [ %0, %1 ]
    ret i32 %14
}
```

## LLVM-IR API

- ▶ LLVM offers two APIs: C++ and C
  - ▶ C++ is the full API, exposing nearly all internals
  - ▶ C API is more limited, but more stable
- ▶ Nearly all major versions have breaking changes
- ▶ Some support for multi-threading:
  - ▶ All modules/types/... associated with an LLVMContext
  - ▶ Different contexts may be used in different threads

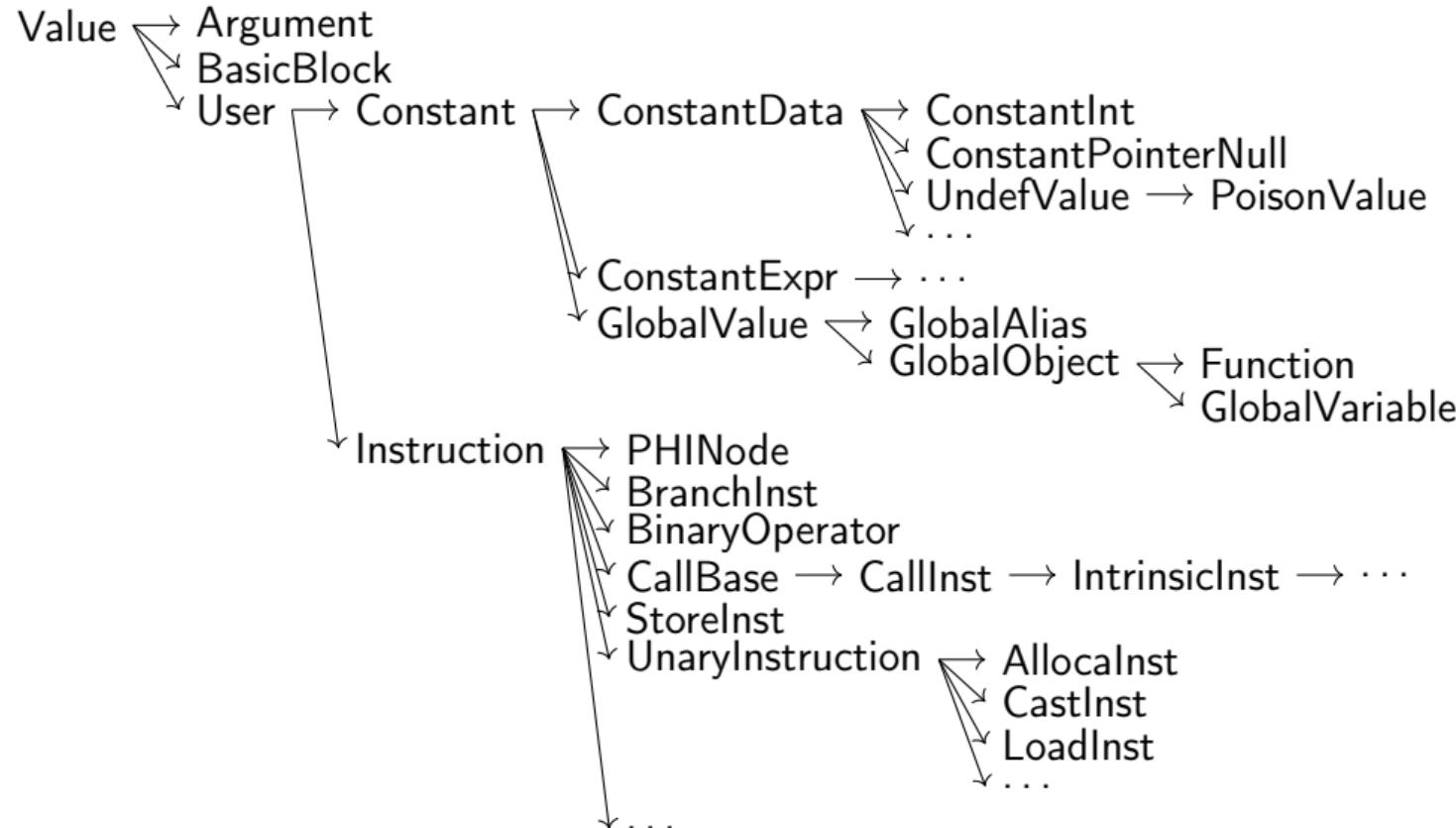
# LLVM-IR C++ API: Basic Example

```
#include <llvm/IR/IRBuilder.h>
int main(void) {
    llvm::LLVMContext ctx;
    auto modUP = std::make_unique<llvm::Module>("mod", ctx);

    llvm::Type* i64 = llvm::Type::getInt64Ty(ctx);
    llvm::FunctionType* fnTy = llvm::FunctionType::get(i64, {i64}, false);
    llvm::Function* fn = llvm::Function::Create(fnTy,
                                                llvm::GlobalValue::ExternalLinkage, "addOne", modUP.get());
    llvm::BasicBlock* entryBB = llvm::BasicBlock::Create(ctx, "entry", fn);

    llvm::IRBuilder<> irb(entryBB);
    llvm::Value* add = irb.CreateAdd(fn->getArg(0), irb.getInt64(1));
    irb.CreateRet(add);
    modUP->print(llvm::outs(), nullptr);
    return 0;
}
```

# LLVM-IR API: Almost Everything is a Value... (excerpt)



# LLVM-IR API: Programming Environment

- ▶ LLVM implements custom RTTI
  - ▶ `isa<>`, `cast<>`, `dyn_cast<>`
- ▶ LLVM implements a multitude of specialized data structures
  - ▶ E.g.: `SmallVector<T, N>` to keep  $N$  elements stack-allocated
  - ▶ Custom vectors, sets, maps; see manual<sup>15</sup>
- ▶ Preferably uses `ArrayRef`, `StringRef`, `Twine` for references
- ▶ LLVM implements custom streams instead of std streams
  - ▶ `outs()`, `errs()`, `dbgs()`

<sup>15</sup><https://www.llvm.org/docs/ProgrammersManual.html>

# LLVM and IR Design

- ▶ LLVM provides a decent general-purpose IR for compilers
- ▶ But: not ideal for all purposes
  - ▶ High-level optimizations difficult, e.g. due to lost semantics
  - ▶ Several low-level operations only exposed as intrinsics
  - ▶ IR rather complex, high code complexity
  - ▶ High compilation times, not very efficient data structures
- ▶ Thus: heavy trend towards custom IRs

## LLVM-IR – Summary

- ▶ LLVM is a modular compiler framework
- ▶ Extremely popular and high-quality compiler back-end
- ▶ Primarily provides optimizations and a code generator
- ▶ Main interface is the SSA-based LLVM-IR
  - ▶ Easy to generate, friendly for writing front-ends/optimizations

# LLVM-IR – Questions

- ▶ What is the structure of an LLVM-IR module/function?
- ▶ Which LLVM-IR data types exist?  
How do they relate to the target architecture?
- ▶ How do semantically invalid operations in LLVM-IR behave?
- ▶ What is special about intrinsic functions?
- ▶ How to derive LLVM-IR from C code using Clang?

# Code Generation for Data Processing

## Lecture 5: Analyses and Transformations

Alexis Engelke

Chair of Data Science and Engineering (I25)  
School of Computation, Information, and Technology  
Technical University of Munich

Winter 2025/26

# Program Transformation: Motivation

- ▶ “User code” is often not very efficient
- ▶ Also: no need to, compiler can (often?) optimize better
  - ▶ More knowledge: e.g., data layout, constants after inlining, etc.
- ▶ Allows for more pragmatic/simple code
- ▶ Generating “better” IR code on first attempt is expensive
  - ▶ What parts are actually used? How to find out?
- ▶ Transformation to “better” code must be done *somewhere*
- ▶ Optimization is a misnomer: we don’t know whether it improves code!
  - ▶ Many transformations are driven by heuristics
- ▶ Many types of optimizations are well-known<sup>16</sup>

<sup>16</sup> FE Allen and J Cocke. *A catalogue of optimizing transformations*. 1971. 

# Dead Block Elimination

- ▶ CFG not necessarily connected
- ▶ E.g., consequence of optimization
  - ▶ Conditional branch → unconditional branch
- ▶ Removing dead blocks is trivial
  1. DFS traversal of CFG from entry, mark visited blocks
  2. Remove unmarked blocks

# Optimization Example 1

```
define i32 @fac(i32 %0) {
  br label %for.header
for.header: ; preds = %for.body, %1
  %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]
  %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
  %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
  %cond = icmp sle i32 %i, %0
  br i1 %cond, label %for.body, label %exit
for.body: ; preds = %for.header
  %a.new = mul i32 %a, %i
  %b.new = add i32 %b, %i
  %i.new = add i32 %i, 1
  br label %for.header
exit: ; preds = %for.header
  %absum = add i32 %a, %b
  ret i32 %a
}
```

# Simple Dead Code Elimination (DCE)

- ▶ Look for trivially dead instructions
  - ▶ No users or side-effects
  - ▶ Calls *might* be removed

1. Add all instructions to work queue
2. While work queue not empty:
  - 2.1 Check for deadness (zero users, no side-effects)
  - 2.2 If dead, remove and add all operands to work queue

**Warning:** Don't implement it this naively, this is inefficient

# Applying Simple DCE

```
define i32 @fac(i32 %0) {
eff.: cf  br label %for.header
          for.header: ; preds = %for.body, %1
users: 3  %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]
users: 2  %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
users: 4  %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
users: 1  %cond = icmp sle i32 %i, %0
eff.: cf  br i1 %cond, label %for.body, label %exit
          for.body: ; preds = %for.header
users: 1  %a.new = mul i32 %a, %i
users: 1  %b.new = add i32 %b, %i
users: 1  %i.new = add i32 %i, 1
eff.: cf  br label %for.header
          exit: ; preds = %for.header
users: 0  %absum = add i32 %a, %b
eff.: cf  ret i32 %a
}
```

# Dead Code Elimination

- ▶ Problem: unused value cycles
- ▶ Idea: find “value sinks” and mark all needed values as live  
unmarked values can be removed
  - ▶ Sink: instruction with side effects (e.g., store, control flow)
  - ▶ Alternative formulation: assume instruction is dead unless proven otherwise

1. Only mark instrs. with side effects as live
2. Populate work list with newly added live instrs.
3. While work list not empty:
  - 3.1 Mark dead operand instructions as live and add to work list
4. Remove instructions not marked as live

# Applying Liveness-based DCE

```
define i32 @fac(i32 %0) {  
  live  br1 label %for.header  
  for.header: ; preds = %for.body, %1  
  live  %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]  
  
  live  %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]  
  live  %cond = icmp sle i32 %i, %0  
  live  br2 i1 %cond, label %for.body, label %exit  
  for.body: ; preds = %for.header  
  live  %a.new = mul i32 %a, %i  
  
  live  %i.new = add i32 %i, 1  
  live  br3 label %for.header  
  exit: ; preds = %for.header  
  
  live  ret i32 %a  
}
```

Work list (stack)

## Optimization Example 2

```
define i32 @trivial() {
entry:
  br label %for.cond
for.cond:
  %x = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp ugt i32 %x, 0
  br i1 %cmp, label %for.inc, label %ret
for.inc:
  %inc = add i32 %x, 1
  br label %for.cond
ret:
  ret i32 %x
}
```

# Sparse Conditional Constant Propagation<sup>17</sup>

- ▶ Constant folding and dead code elimination separately insufficient
- ▶ Idea: be optimistic about reachability and constantness
  - ▶ Assume block is dead/value is constant unless proven otherwise
  - ▶ Ignore edges that are not proven as executable
- ▶ For every value: store lattice element
  - ▶ Lattice:  $\top$  (any value)  $>$  *constant* (single constant value)  $>$   $\perp$  (dynamic)
  - ▶  $\top \wedge x = x, \perp \wedge x = \perp, x \wedge y = (x \text{ if } x = y \text{ else } \perp)$
- ▶ For every edge: store executability
  - ▶  $\phi$ -nodes: edges that are not (yet) marked executable get  $\top$

<sup>17</sup>MN Wegman and FK Zadeck. "Constant propagation with conditional branches". In: *TOPLAS* 13.2 (1991), pp. 181–210. 

# SCCP Algorithm

State:

- ▶ EdgeWorklist
- ▶ InstrWorklist
- ▶ EdgeExecutable
  - ▶ Initialize false
- ▶ InstrLattice
  - ▶ Initialize  $T$

- ▶ EdgeWorklist  $\leftarrow$  edge to entry block
- ▶ Get item from any work list; stop when empty
- ▶ Edge – if not yet visited, mark visited and:
  - ▶ Visited block first time: *visit* all instrs (incl.  $\phi$ )
  - ▶ Otherwise: *visit* just  $\phi$ -nodes
- ▶ Instr – if block is reachable: *visit*
- ▶ At end: replace instrs with const lattice values

# SCCP Algorithm II

## *visit*

- ▶  $\phi$ -node: update lattice, meet of visited edge values
- ▶ Other instructions: try constant fold
- ▶ If lattice value changed:
  - ▶ Add all users to instruction worklist
  - ▶ Branch with constant/no condition: add selected edge to worklist
  - ▶ Branch with  $\perp$ : add all edges to worklist

```
define i32 @fn() {
entry:
  br label %loop
loop:
  %j2 = phi i32 [ 1, %entry ], [ %j4, %ifmerge ]
  %k2 = phi i32 [ 0, %entry ], [ %k4, %ifmerge ]
  %kcond = icmp slt i32 %k2, 100
  br i1 %kcond, label %loopbody, label %ret
loopbody:
  %jcond = icmp slt i32 %j2, 20
  br i1 %jcond, label %then, label %else
then:
  %k3 = add i32 %k2, 1
  br label %ifmerge
else:
  %k5 = add i32 %k2, 1
  br label %ifmerge
ifmerge:
  %j4 = phi i32 [ 1, %then ], [ %k2, %else ]
  %k4 = phi i32 [ %k3, %then ], [ %k5, %else ]
  br label %loop
ret:
  ret i32 %j2
}
```

Apply SCCP algorithm  
and fold constants.  
Then, remove dead  
instructions and blocks.

## SCCP: Misc

- ▶ Strictly more powerful than separate constant folding + DCE
- ▶ Runtime:  $\mathcal{O}(\# \text{CFG-edges} + \# \text{instr.-operands})$ 
  - ▶ Every CFG edge is processed at most once
  - ▶ Every def-use relation is processed at most twice  
lattice can lower at most twice ( $\top \rightarrow \text{const}, \text{const} \rightarrow \perp$ )
- ▶ Can also be used as inter-procedural optimization  
reaching into module-internal functions

## Optimization Example 3

```
define i32 @foo(i32 %0, ptr %1, ptr %2) {
%4 = zext i32 %0 to i64
%5 = getelementptr i32, ptr %1, i64 %4
%6 = load i32, ptr %5, align 4
%7 = zext i32 %0 to i64
%8 = getelementptr i32, ptr %2, i64 %7
%9 = load i32, ptr %8, align 4
%10 = add nsw i32 %6, %9
ret i32 %10
}
```

## Common Subexpression Elimination (CSE) – Attempt 1

- ▶ Idea: find/eliminate redundant computation of same value
- ▶ Keep track of previously seen values in hash map
- ▶ Iterate over all instructions
  - ▶ If found in map, remove and replace references
  - ▶ Otherwise add to map
- ▶ Easy, right?

# CSE Attempt 1 – Example 1

```
define i32 @foo(i32 %0, ptr %1, ptr %2) {  
→ ht    %4 = zext i32 %0 to i64  
→ ht    %5 = getelementptr i32, ptr %1, i64 %4  
→ ht    %6 = load i32, ptr %5, align 4  
dup %4  %7 = zext i32 %0 to i64  
→ ht    %8 = getelementptr i32, ptr %2, i64 %7%4  
→ ht    %9 = load i32, ptr %8, align 4  
→ ht    %10 = add nsw i32 %6, %9  
→ ht    ret i32 %10  
}
```

- ▶ Obsolete instr. can be killed immediately, or in a later DCE

# CSE Attempt 1 – Example 2

```
define i32 @square(i32 %a, i32 %b) {
entry:
→ ht    %cmp = icmp slt i32 %a, %b
→ ht    br i1 %cmp, label %if.then, label %if.end
if.then: ; preds = %entry
→ ht    %add1 = add i32 %a, %b
→ ht    br label %if.end
if.end: ; preds = %if.then, %entry
→ ht    %condvar = phi i32 [ %add1, %if.then ], [ %a, %entry ]
dup %add1  %add2 = add i32 %a, %b
→ ht    %res = add i32 %condvar, %add2%add1
→ ht    ret i32 %res
}
```

---

Instruction does not dominate all uses!  
error: input module is broken!

# Domination

- ▶ Remember: CFG  $G = (N, E, s)$  with digraph  $(N, E)$  and entry  $s \in N$
- ▶ Dominate:  $d \text{ dom } n$  iff every path from  $s$  to  $n$  contains  $d$ 
  - ▶ Dominators of  $n$ :  $DOM(n) = \{d | d \text{ dom } n\}$
- ▶ Strictly dominate:  $d \text{ sdom } n \Leftrightarrow d \text{ dom } n \wedge d \neq n$
- ▶ Immediate dominator:  
$$\text{idom}(n) = d : d \text{ sdom } n \wedge \nexists d'. d \text{ sdom } d' \wedge d' \text{ sdom } n$$

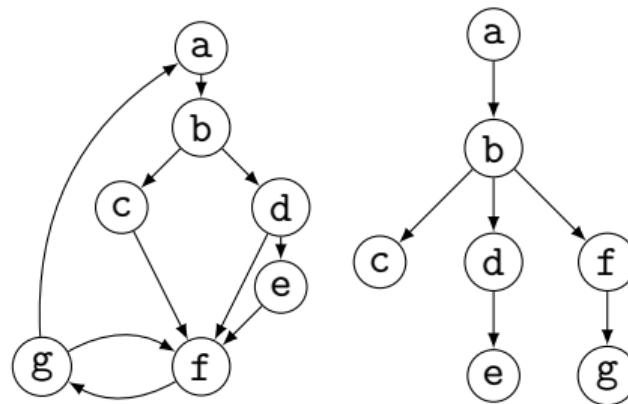
  

- ⇒ All strict dominators are always executed before the block
- ⇒ All values from dominators available/usable
- ⇒ All values not from dominators **not** usable

# Dominator Tree

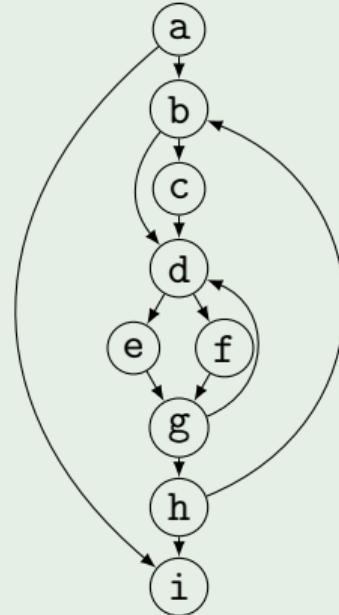
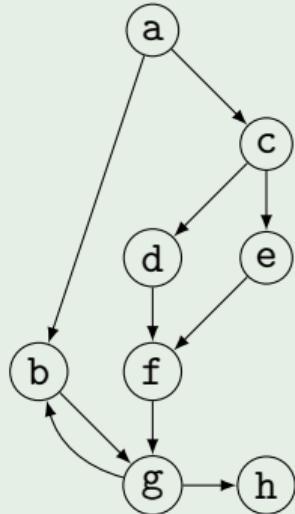
- ▶ Tree of immediate dominators
- ▶ Allows to iterate over blocks in pre-order/post-order
- ▶ Answer  $a \text{ sdom } b$  quickly

Control Flow Graph      Dominator Tree



## Dominator Tree – Example

Construct the dominator tree for the following CFGs (entry at a):



# Dominator Tree: Construction

- ▶ Naive: inefficient (but reasonably simple)<sup>18</sup>
  - ▶ For each block: find a path from the root – superset of dominators
  - ▶ Remove last block on path and check for alternative path
  - ▶ If no alternative path exists, last block is idom
- ▶ Lengauer–Tarjan: more efficient methods<sup>19</sup>
  - ▶ Simple method in  $\mathcal{O}(m \log n)$ ; sophisticated method in  $\mathcal{O}(m \cdot \alpha(m, n))$   
( $\alpha(m, n)$  is the inverse Ackermann function, grows *extremely* slowly)
  - ▶ Used in some compilers<sup>20</sup>
- ▶ Semi-NCA:  $\mathcal{O}(n^2)$ , but lower constant factors<sup>21</sup>

<sup>18</sup>ES Lowry and CW Medlock. "Object code optimization". In: *CACM* 12.1 (1969), pp. 13–22. 

<sup>19</sup>T Lengauer and RE Tarjan. "A fast algorithm for finding dominators in a flowgraph". In: *TOPLAS* 1.1 (1979), pp. 121–141. 

<sup>20</sup>Example: <https://github.com/WebKit/WebKit/blob/aabfacb/Source/WTF/wtf/Dominators.h>

<sup>21</sup>L Georgiadis. "Linear-Time Algorithms for Dominators and Related Problems". PhD thesis. Princeton University, Nov. 2005. 

## Dominator Tree: Implementation

- ▶ Per node store:  $idom$ , idom-children, DFS pre-order/post-order number
- ▶ Get immediate dominator: ...lookup  $idom$
- ▶ Iterate over all dominators/dominated by: ...trivial
- ▶ Check whether a sdom  $b^{23}$ 
  - ▶  $a.preNum < b.preNum \wedge a.postNum > b.postNum$
  - ▶ After updates, numbers might be invalid: recompute or walk tree
- ▶ Problem: dominance of unreachable blocks ill-defined  $\rightsquigarrow$  special handling

<sup>23</sup>PF Dietz. "Maintaining order in a linked list". In: *STOC*. 1982, pp. 122–127. 

# CSE Attempt 2

- ▶ Option 1:
  - ▶ For identical instructions, store all
  - ▶ Add dominance check before replacing
  - ▶ Visit nodes in reverse post-order (i.e., topological order)
- ▶ Option 2 (Global Value Numbering):<sup>24</sup>
  - ▶ Do a DFS over dominator tree
  - ▶ Use scoped hashmap to track available values

Does this work? Yes.

<sup>24</sup>P Briggs, KD Cooper, and LT Simpson. *Value numbering*. Tech. rep. CRPC-TR94517-S. Rice University, 1997. 

## CSE: Hashing an Instruction (and Beyond)

- ▶ Needs hash function *and* “relaxed” equality
- ▶ Idea: combine opcode and operands/constants into hash value
  - ▶ E.g. assign number to values (hence “value numbering”)
  - ▶ Alternatively: use pointer or index for instruction result operands
- ▶ Canonicalize commutative operations
  - ▶ Order operands deterministically, e.g., by number/address
- ▶ Identities:  $a + (b + c)$  vs.  $(a + b) + c$

# More Complex Global Value Numbering

- ▶ Hash-based approach only catches trivially removable duplicates
- ▶ Alternative: partition values into *congruence classes*
  - ▶ Congruent values are guaranteed to always have the same value
- ▶ Optimistic approach: values are congruent unless proven otherwise
- ▶ Pessimistic approach: values are not congruent unless proven
- ▶ Combinable with: reassociation, DCE, constant folding
- ▶ Rather complex, but can be highly beneficial<sup>25</sup>

<sup>25</sup>K Gargi. "A sparse algorithm for predicated global value numbering". In: *PLDI*. 2002, pp. 45–56.

# Inlining

- ▶ Inlining: copy function body in place of the call
  - ▶ Historically also referred to as “procedure integration”
- ▶ Benefits:
  - ▶ More optimization opportunities: constant propagation, dead code elimination, better register allocation, etc.
  - ▶ Avoids call overhead: stack frame setup, register saving, etc.
- ▶ Inlining is crucial for performance in many programming languages
  - ▶ “Zero-cost” abstractions typically rely on small functions
  - ▶ Critical in e.g. C++/Rust; not too important for C

## Inlining: Decision

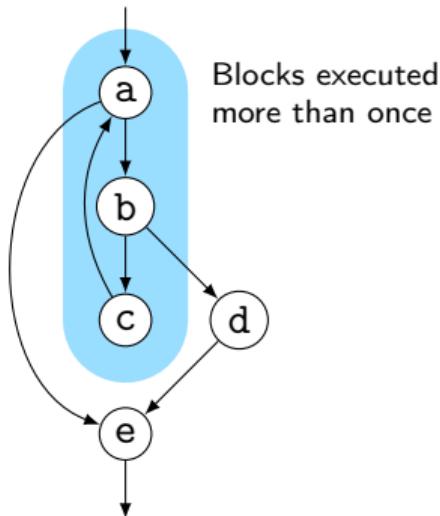
- ▶ First: determine whether function is inlinable
- ▶ Easy way: delegate to programmer (`__attribute__((always_inline))`)
- ▶ Otherwise: very difficult problem
- ▶ Problem 1: ordering
  - ▶ Inlining big `a()` into `b()` might make `b()` too big to inline but inlining `b()` into `c()` might provide more opportunities
- ▶ Problem 2: cost model and thresholds
  - ▶ Estimate cost of inlining, e.g. additional code size, more branches
  - ▶ Estimate optimization gains and hotness
  - ▶ Some attempts to use machine learning here

# Inlining Transformation

- ▶ Copy original function in place of the call
  - ▶ Split basic block containing function call
  - ▶ Needs map of old to new value
  - ▶ Constant fold/simplify instructions along the way
  - ▶ Replace constant conditional branches  $\rightsquigarrow$  fewer copying
- ▶ Replace returns with branches and  $\phi$ -node to/at continuation point
- ▶ Move static alloca to beginning
- ▶ Dynamic alloca: save/restore stack pointer
  - ▶ Prevent unbounded stack growth in loops
  - ▶ LLVM provides `stacksave`/`stackrestore` intrinsics
- ▶ Exceptions may need special treatment

# What is a Loop?

```
void func() {  
    while (a()) {  
        if (b()) {  
            d();  
            break;  
        }  
        c();  
    }  
    e();  
}
```

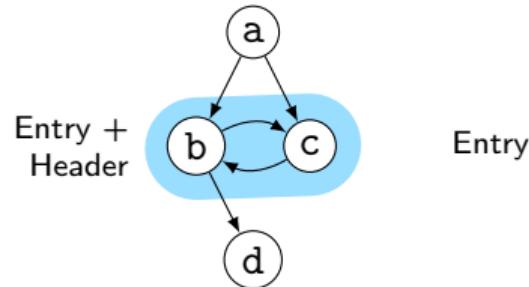
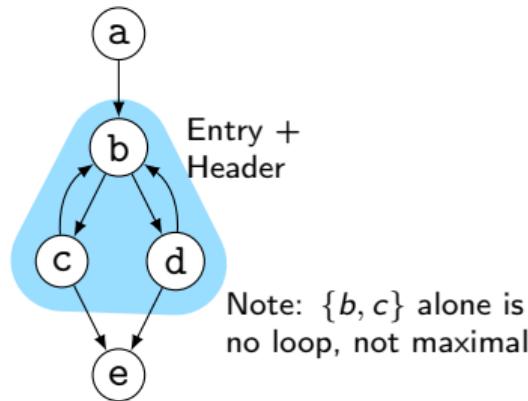
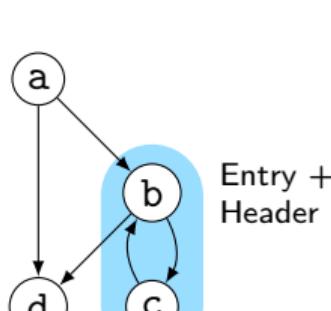


- ▶ Loops in source code  $\neq$  loops in CFG
- ▶ d is *not* part of loop: executed at most once

↝ Need algorithm to find loops in CFG

# Loops

- ▶ Loop: maximal SCC  $L$  with at least one internal edge<sup>27</sup>  
(strongly connected component (SCC): all blocks reachable from each other)
  - ▶ Entry: block with an edge from outside of  $L$
  - ▶ Header  $h$ : first entry found (might be ambiguous)
- ▶ Loop nested in  $L$ : loop in subgraph  $L \setminus \{h\}$



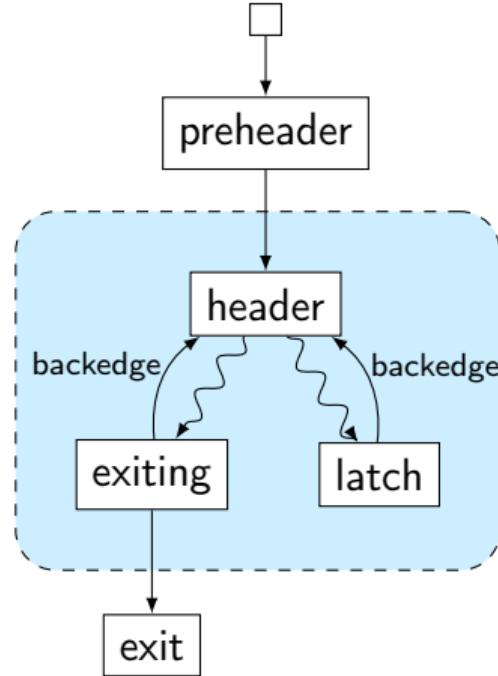
<sup>27</sup> P Havlak. "Nesting of reducible and irreducible loops". In: *TOPLAS* 19.4 (1997), pp. 557–567.

# Natural Loops

- ▶ Natural Loop: loop with single entry
  - ⇒ Header is unique
  - ⇒ Header dominates all block
  - ⇒ Loop is reducible
- ▶ Backedge: edge from block to header
- ▶ Predecessor: block with edge into loop
- ▶ Preheader: unique predecessor

## Formal Definition

Loop  $L$  is reducible iff  $\exists h \in L . \forall n \in L . h \text{ dom } n$   
CFG is reducible iff all loops are reducible



# Finding Natural Loops

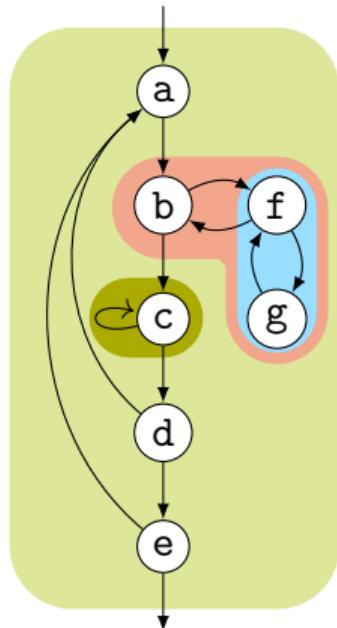
- ▶ Modified version<sup>28</sup> of Tarjan's algorithm<sup>29</sup>
- ▶ Iterate over dominator tree in post order
- ▶ Each block: find predecessors dominated by the block
  - ▶ None  $\rightsquigarrow$  no loop header, continue
  - ▶ Any  $\rightsquigarrow$  loop header, these edges *must* be backedges
- ▶ Walk through predecessors until reaching header again
  - ▶ All blocks on the way must be part of the loop body
  - ▶ Might encounter nested loops, update loop parent

<sup>28</sup> G Ramalingam. "Identifying loops in almost linear time". In: *TOPLAS* 21.2 (1999), pp. 175–188. 

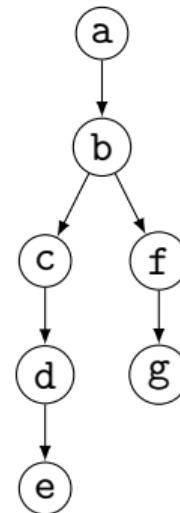
<sup>29</sup> R Tarjan. "Testing flow graph reducibility". In: *STOC*. 1973, pp. 96–107. 

# Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



Loop Info

Loop A: {c}

header: c; parent: D

Loop B: {f, g}

header: f; parent: C

Loop C: {b, f, g}

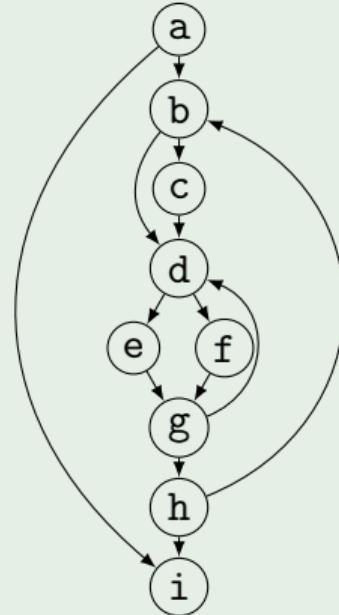
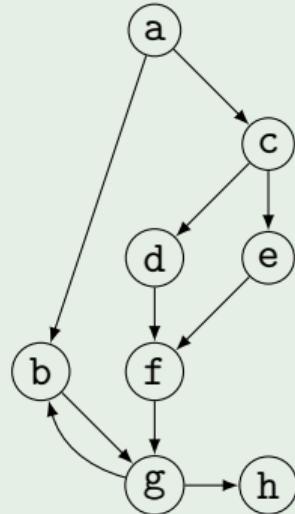
header: b; parent: D

Loop D: {a, b, c, d, e, f, g}

header: a; parent: NULL

## Loop Analysis – Example

Apply the previous algorithm to find loops in the following CFGs (entry at a):



# Loop Invariant Code Motion (LICM)

- ▶ Analyze loops, iterate over loop tree in post-order
  - ▶ I.e., visit inner loops first
- ↑ Hoist:<sup>30</sup> iterate over blocks of loop in reverse post-order
  - ▶ For each movable inst., check for loop-defined operands
  - ▶ If not, move to preheader (create one, if not existent)
  - ▶ Otherwise, add inst. to set of values defined inside loop
- ↓ Sink: Iterate over blocks of loop in post-order
  - ▶ For each movable inst., check for users inside loop
  - ▶ If none, move to unique exit (if existent)

<sup>30</sup><https://github.com/bytecakealliance/wasmtime/blob/bd6fe11/cranelift/codegen/src/licm.rs>

# Loop Unswitch

- ▶ Hoist loop-invariant conditionals outside loop

```
while (...)                                if (loopInvariantCond)
    if (loopInvariantCond)                while (...)

        A                                A
    else                                C
        B                                else
    C                                while (...)

                                B
                                C
```

→

- ▶ Might duplicate parts of the loop ↪ needs heuristics
- ▶ Might lead to exponential code size increase

# Loop Rotation

- ▶ Rotate loop condition to bottom
  - ▶ Essentially, convert `while{}` to `if{do{}}while{}`

```
define void @src() {  
    ; ...  
loopheader:  
    %iv = phi ...  
    %cond = ...  
    br i1 %cond, %body, %exit  
body:  
    ; ...  
    br %loopheader  
exit: ; ...  
}
```

→

```
define void @tgt() {  
    ; ...  
    %cond1 = ...  
    br i1 %cond1, %body, %exit  
body:  
    %iv = phi ...  
    ; ...  
    %cond2 = ...  
    br i1 %cond2, %body, %exit  
exit: ; ...  
}
```

# Loop Strength Reduction

- ▶ Loop induction variables (e.g., loop counters) are often multiplied
  - ▶ E.g., for array indexing
- ▶ Strength reduction: replace “strong” multiply with “weak” addition
- ▶ Needs: analysis of loop induction variables and their recurrence
- ▶ Rewrite through replaced/extra loop induction variable
  - ▶ E.g., replace scaled index operation with pointer addition
- ▶ Information about ISA addressing modes beneficial

## Analyses and Transformations – Summary

- ▶ Program Transformation critical for performance improvement
- ▶ Code not necessarily better
- ▶ Analyses are important to drive transformations
  - ▶ Dominator tree, loop detection, value liveness
- ▶ Important optimizations
  - ▶ Dead code elimination, sparse conditional constant propagation, common sub-expression elimination, loop-invariant code motion, loop unswitching, loop rotation, loop strength reduction

## Analyses and Transformations – Questions

- ▶ Why is “optimization” a misleading name for a transformation?
- ▶ How to find unused code sections in a function’s CFG?
- ▶ Why is a liveness-based DCE better than a simple, user-based DCE?
- ▶ Why is SCCP more powerful than separate DCE/constant prop.?
- ▶ What is a dominator tree useful for?
- ▶ What is the difference between an irreducible and a natural loop?
- ▶ How to find natural loops in a CFG?
- ▶ How does the algorithm handle irreducible loops?
- ▶ Why is sinking a loop-invariant inst. harder than hoisting?

# Code Generation for Data Processing

## Lecture 6: Optimizations in LLVM

Alexis Engelke

Chair of Data Science and Engineering (I25)  
School of Computation, Information, and Technology  
Technical University of Munich

Winter 2025/26

# Optimizations in LLVM

- ▶ Analysis: collect information about IR
  - ▶ Examples: loop info, alias analysis, branch probabilities
  - ▶ Results are cached, re-run when outdated
- ▶ Pass: perform transformation on module/CGSCC/function/loop
  - ▶ Examples: instruction combine, simplify CFG, global value numbering, scalar replacement of aggregates (SROA; promotes alloca to SSA)
  - ▶ Input and output IR must be valid
  - ▶ Can use results of analyses; invalidates (some) analyses on changes
- ▶ Pass Manager: execute series of passes of same granularity
  - ▶ Otherwise, use adaptor: `createFunctionToLoopPassAdaptor`

# LLVM Optimization Pipeline (Simplified)<sup>31</sup>

Module Pipeline (`buildPerModuleDefaultPipeline`): two parts

Module Simplification Pipeline (`buildModuleSimplificationPipeline`)

- ▶ Early function simplification (SimplifyCFG, SROA, EarlyCSE)
- ▶ CGSCC (post-order) (repeated after devirtualization):
  - ▶ Inliner
  - ▶ Function Simplification Pipeline (`buildFunctionSimplificationPipeline`)
    - ▶ SROA, SimplifyCFG, InstCombine, Reassociate, GVN, DCE, LICM
    - ▶ Most passes are here

Module Optimization Pipeline (`buildModuleOptimizationPipeline`)

- ▶ Primarily hard-to-reverse loop optimizations that are not simplifications
- ▶ E.g. vectorization, loop unrolling, loop distribution

<sup>31</sup>See `llvm/lib/Passes/PassBuilderPipelines.cpp` for full details.

## SSA Construction: Mem2Reg and SROA

- ▶ Front-ends typically emit `allocas` for variables
  - ▶ Easier; also simplifies debugging and debug information
- ▶ Mem2reg: promote `alloca` to SSA values/phis
  - ▶ Condition: only `load/store`, no address taken
  - ▶ Essentially just SSA construction
- ▶ SROA: scalar replacement of aggregate
  - ▶ Separate structure fields into separate variables
  - ▶ Also promote them to SSA, subsumes mem2reg

# Simplification

- ▶ Many operations can be expressed in various ways
  - ▶ E.g.: `sub i32 %x, 1, add i32 %x, -1, add i32 -1, %x`
- ▶ Problem: always need to consider all equivalent cases

⇒ Canonicalize IR: one single preferred form

- ▶ Single instructions: e.g. operand order
- ▶ Instruction sequences: e.g. series of additions, useless PHIs
- ▶ CFG: e.g. branches, loop layout

- ▶ E.g.: constants go to the right, constant sub is add, loops with fixed trip count turned into for `(i = 0; i != 25; ++i), ...`

# InstCombine

- ▶ Main instruction canonicalization and simplification pass
- ▶ Includes many arithmetic-logical patterns, folding of PHIs, ...
- ▶ Implementation:
  - ▶ Add instructions to work list
    - ▶ Trivially dead/constant instructions eliminated immediately
    - ▶ Blocks traversed in reverse post-order
  - ▶ Patterns applied to instructions; on match, replace all uses
  - ▶ Changed instruction re-enqueues all users to work list
- ▶ Historically: fixed-point iteration, now just one iter. per run
- ▶ Covers many patterns, many patterns missing, >50kLOC C++

# IR Pattern Matching

- ▶ IR patterns are typically DAG matches on def-use graph
- ▶ Typically implemented via helper functions<sup>32</sup>

```
// Excerpt from InstCombinerImpl::visitAdd

// (add (xor A, B) (and A, B)) --> (or A, B)
// (add (and A, B) (xor A, B)) --> (or A, B)
if (match(&I, m_c_BinOp(m_Xor(m_Value(A), m_Value(B)),
                         m_c_And(m_Deferred(A), m_Deferred(B)))))

    return BinaryOperator::CreateOr(A, B);
```

<sup>32</sup>llvm/IR/PatternMatch.h

# LLVM: Use Tracking

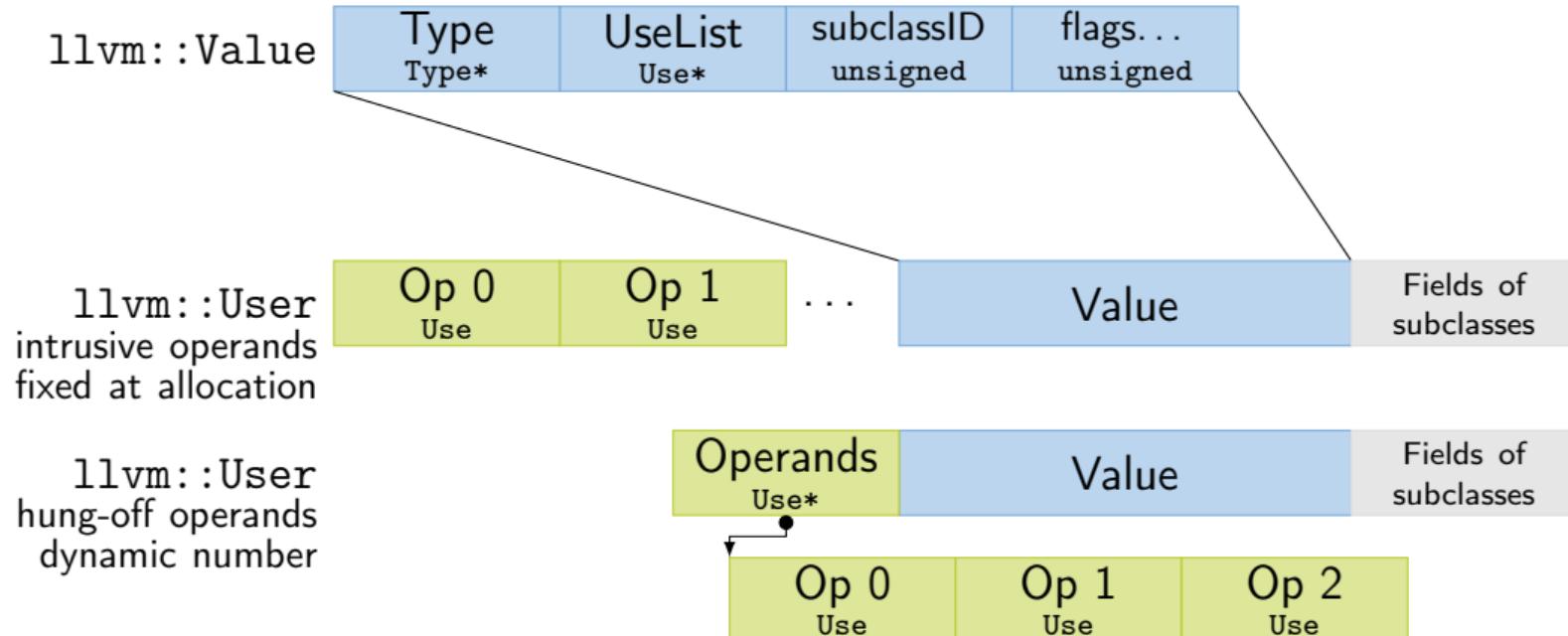
- ▶ Values track their users

```
llvm::Value* v = /* ... */;
for (llvm::User* u : v->users())
    if (auto i = llvm::dyn_cast<llvm::Instruction>(u))
        // ...
```

- ▶ Allows for easy replacement:

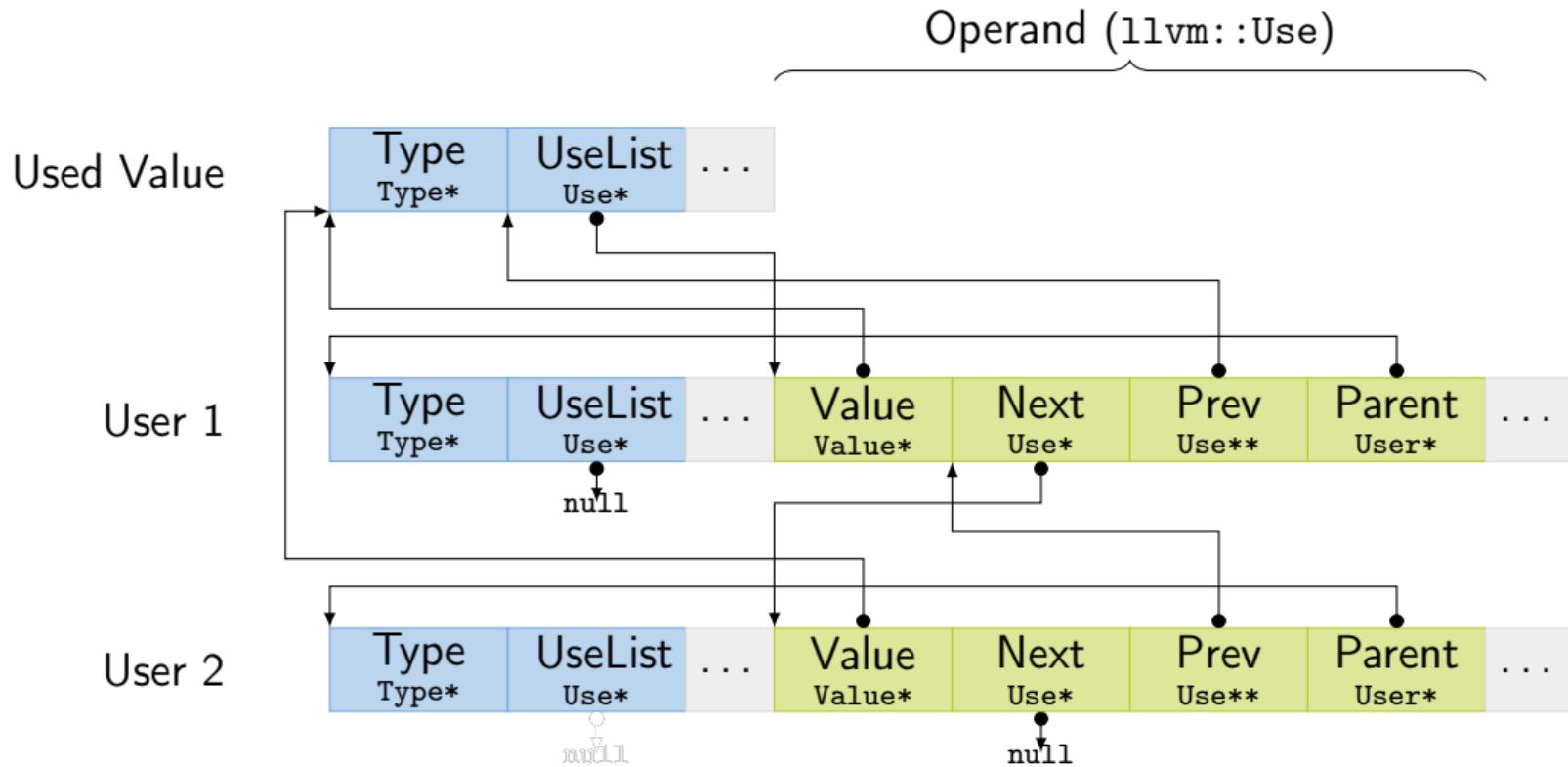
- ▶ `inst->replaceAllUsesWith(replVal);`

# LLVM IR Implementation: Value/User



PHINode additionally stores  $n$  `BasicBlock*` after the operands, but aren't users of blocks.

# LLVM IR Implementation: Use



## LLVM IR Implementation: Instructions/Blocks

- ▶ Instruction and BasicBlock have pointers to parent and next/prev
  - ▶ Linked list updated on changes and used for iteration
  - ▶ Instructions have cached *order* (integer) for fast “comes before”
- ▶ BasicBlock successors: blocks used by terminator
- ▶ BasicBlock predecessors:
  - ▶ Iterate over users of block – these are terminators (and `blockaddress`)
  - ▶ Ignore non-terminators, parent of using terminator is predecessor
  - ▶ Same predecessor might be duplicated ( $\rightsquigarrow$  `getUniquePredecessor()`)
- ▶ Finding first non- $\phi$  requires iterating over  $\phi$ -nodes

# Valid Transforms?

Which of these transformations  $\text{src} \rightarrow \text{tgt}$  are valid?

```
define float @src1(float %x) {  
  %a = fadd float %x, 0.0  
  ret float %a  
}
```

```
define i1 @src2(i8 %x) {  
  %xm1 = add i8 %x, -1  
  %y = xor i8 %x, %xm1  
  %r = icmp ult i8 %x, %y  
  ret i1 %r  
}
```

```
define i8 @src3(i8 %x) {  
  %div = sdiv i8 %x, 4  
  ret i8 %div  
}
```

```
define float @tgt1(float %x) {  
  ret float %x  
}
```

```
define i1 @tgt2(i8 %x) {  
  %ctpop = call i8 @llvm.ctpop(i8 %x)  
  %r = icmp ult i8 %ctpop, 2  
  ret i1 %r  
}
```

```
define i8 @tgt3(i8 %x) {  
  %div = ashr i8 %x, 2  
  ret i8 %div  
}
```

# Alive2

- ▶ Seemingly correct transformations might be wrong in edge cases
- ▶ Unexpected transformations can be correct
- ⇒ Prove correctness of transformations
- ▶ Alive2<sup>33</sup>: translation validation tool for LLVM-IR
- ▶ Convert LLVM-IR into SMT formulas
- ▶ Proves that target is a refinement of source function
- ▶ InstCombine transformations require Alive2 proofs nowadays<sup>34</sup>

<sup>33</sup>NP Lopes et al. "Alive2: bounded translation validation for LLVM". In: *PLDI*. 2021, pp. 65–79. .

<sup>34</sup>Random Example: [https://alive2.llvm.org/ce/z/xe\\_vb2](https://alive2.llvm.org/ce/z/xe_vb2)

# Value Tracking

- ▶ Many transformations benefit from value ranges and known bits
  - ▶ E.g., fold `trunc+shift` into smaller shift if shift amount is known small
- ▶ `computeKnownBits`: recursively collect information from `instrs`
  - ▶ Arithmetic-logical `instrs`, casts, several intrinsics, etc.
- ▶ Bitwise instructions easier to reason about  $\rightsquigarrow$  preferred canonicalization
  - ▶ E.g.  $(1 \ll X) - 1 \rightarrow (-1 \ll X) \wedge -1$
  - ▶ E.g. `sext(X)  $\rightarrow$  zext(X)` if `X` is known non-negative
- ▶ Also use knowledge from front-end (but how encoded?)

# Attribute, Flags, Assume

- ▶ Parameter/return attributes: constrain value ranges
  - ▶ range(from, to), nonnull, noundef, ...
  - ▶ Values outside of the range are poison
- ▶ Instruction flags: encode additional knowledge about operation
  - ▶ add/sub/mul/trunc nsw/nuw: no signed/unsigned wrap
  - ▶ or disjoint: combination of disjoint bits
  - ▶ udiv/sdiv/lshr/ashr exact: divisor multiple of dividend
  - ▶ getelementptr nuw/nusw/inbounds: no wrap/stays inside allocation
  - ▶ zext/uitofp nneg: operand is non-negative
  - ▶ icmp same sign: operands have the same sign
  - ▶ Violation is poison

## Assume

- ▶ Intrinsic `llvm.assume(i1)`
- ▶ Conditional immediate undefined behavior if argument is false
- ▶ Requires extra instruction for condition  $\rightsquigarrow$  might be problematic
  - ▶ Instructions use other values, etc.  $\rightsquigarrow$  prevent optimizations
- ▶ Also supports operand bundles for `nonnull`, `align`, `separate_storage`

# Valid Transforms?

Which of these transformations  $\text{src} \rightarrow \text{tgt}$  are valid?

```
define i8 @src1(i8 %x) {  
  %div = udiv exact i8 1, %x  
  ret i8 %div  
}  
  
define i1 @src2(i8 %x, i8 %y) {  
  %cmp = icmp sge i8 %x, %y  
  %cmpeq = icmp samesign eq i8 %x, 127  
  %r = select i1 %cmp, i1 %cmpeq, i1 false  
  ret i1 %r  
}  
  
define i8 @src3(i8 %x, i8 %y, i8 %z) {  
  %xy = add nsw i8 %x, %y  
  %xyz = add nsw i8 %xy, %z  
  ret i8 %xyz  
}
```

```
define i8 @tgt1(i8 %x) {  
  ret i8 1  
}  
  
define i1 @tgt2(i8 %x, i8 %y) {  
  %cmpeq = icmp samesign eq i8 %x, 127  
  ret i1 %cmpeq  
}  
  
define i8 @tgt3(i8 %x, i8 %y, i8 %z) {  
  %xz = add nsw i8 %x, %z  
  %xyz = add nsw i8 %xz, %y  
  ret i8 %xyz  
}
```

# Fast-Math Flags

- ▶ Floating-point arithmetic typically follows IEEE 754
  - ▶ Except for NaN, which is more relaxed
- ▶ Implication: almost impossible to optimize
  - ▶ non-associative, NaNs, infinity, negative zero, no reciprocals, no contraction
  - ▶ Transformations can result in vastly different results!
- ▶ Additionally: floating-point exceptions
  - ▶ Typically ignored, but some users might be interested
- ▶ Value/operation flags to permit transformations locally
- ▶ `strictfp` function attribute allows more control through constrained floating-point intrinsics

# Branch Weights

- ▶ Annotation of branch with expected probabilities
  - ▶ Sources: programmer annotations, execution profiles
- ▶ Added as metadata to branch instructions
- ▶ Can compute block execution frequencies

```
define i32 @f(i32 %x) {  
    ; ...  
    %cmp = icmp eq i32 %x, 0  
    br i1 %cmp, label %then, label %else, !prof !5  
    ; ...  
}  
  
!5 = !{!"branch_weights", !"expected", i32 1, i32 2000}
```

# Function Argument Optimizations

- ▶ Inferring argument attributes
- ▶ Eliminate dead arguments
  - ▶ Optimistic liveness analysis of arguments (assume dead)
  - ▶ Create new function with fewer arguments
- ▶ Promote pointer arguments to value arguments
  - ▶ Esp. relevant for C++ pass-by-reference
  - ▶ Only possible when both caller/callee CPU features match

# Other Inter-Procedural Optimizations

- ▶ Inlining / Outlining
- ▶ Function Specialization
- ▶ Optimization of global variables
  - ▶ Marking as constant, delete dead variables, attributes, etc.
- ▶ Inter-procedural SCCP
- ▶ Devirtualization / call-target speculation
- ▶ Hot-Cold Splitting
- ▶ ...

# Optimization Remarks

- ▶ Understanding optimization decisions can be difficult
  - ▶ Complex code base, abstraction, heuristics, ...
- ▶ Optimization remarks: passes can report information
  - ▶ Inlining decisions: provide cost values/thresholds for decisions
  - ▶ Others: GVN, LICM, FastISel
- ▶ Output also provided as YAML or a binary format for analysis
- ▶ Lots of verbose output, hard to identify important points
- ▶ Clang: -Rpass=<pass> -Rpass-analysis=<pass>  
-Rpass-missed=<pass>

## Other Aspects not Covered Here

- ▶ Analyses: alias analysis, MemorySSA, Scalar Evolution (SCEV), target-specific analyses/heuristics, ...
- ▶ Optimizations: various loop transformations (distribute, fuse, flatten, polyhedral optimizations), value propagation, vectorization, reassociation, ...
- ▶ Loop-Closed SSA canonical form
  - ▶ Values defined inside loop only used in loop or in  $\phi$ -node in exit
  - ▶ Simplifies tracking uses of values defined in loops

# Using LLVM (New) Pass Manager

```
void optimize(llvm::Function* fn) {
    llvm::PassBuilder pb;
    llvm::LoopAnalysisManager lam{};
    llvm::FunctionAnalysisManager fam{};
    llvm::CGSCCArchitectureManager cgam{};
    llvm::ModuleAnalysisManager mam{};
    pb.registerModuleAnalyses(mam);
    pb.registerCGSCCArchitecture(cgam);
    pb.registerFunctionAnalyses(fam);
    pb.registerLoopAnalyses(lam);
    pb.crossRegisterProxies(lam, fam, cgam, mam);

    llvm::FunctionPassManager fpm{};
    fpm.addPass(llvm::DCEPass());
    fpm.addPass(llvm::createFunctionToLoopPassAdaptor(llvm::LoopRotatePass()));
    fpm.run(*fn, fam);
}
```

# Writing a Pass for LLVM's New PM – Part 1

```
#include "llvm/IR/PassManager.h"
#include "llvm/Passes/PassBuilder.h"
#include "llvm/Passes/PassPlugin.h"

class TestPass : public llvm::PassInfoMixin<TestPass> {
public:
    llvm::PreservedAnalyses run(llvm::Function &F,
                               llvm::FunctionAnalysisManager &AM) {
        // Do some magic
        llvm::DominatorTree *DT = &AM.getResult<llvm::DominatorTreeAnalysis>(F);
        // ...
        llvm::errs() << F.getName() << "\n";
        return llvm::PreservedAnalyses::all();
    }
};

// ...
```

## Writing a Pass for LLVM's New PM – Part 2

```
extern "C" ::llvm::PassPluginLibraryInfo LLVM_ATTRIBUTE_WEAK
llvmGetPassPluginInfo() {
    return { LLVM_PLUGIN_API_VERSION, "TestPass", "v1",
        [] (llvm::PassBuilder &PB) {
            PB.registerPipelineParsingCallback(
                [] (llvm::StringRef Name, llvm::FunctionPassManager &FPM,
                    llvm::ArrayRef<llvm::PassBuilder::PipelineElement>) {
                    if (Name == "testpass") {
                        FPM.addPass(TestPass());
                        return true;
                    }
                    return false;
                });
        } );
    } };
}

c++ -shared -o testpass.so testpass.cc -fPIC
opt -S -load-pass-plugin=$PWD/testpass.so -passes=testpass input.ll
```

# Optimizations in LLVM – Summary

- ▶ LLVM provides a wide range of analyses and optimizations
- ▶ Optimizations organized in flexibly composeable passes
- ▶ Simplification centered around canonicalization
- ▶ Much knowledge inferred from assumptions (and poison/UB)
- ▶ SSA simplifies many transforms through explicit def–use chains
- ▶ Only few non-simplifying transformation passes

## Optimizations in LLVM – Questions

- ▶ Why are some passes executed multiple times during optimization?
- ▶ How does the simplification pipeline relate to inlining?
- ▶ What is the key benefit of canonicalizing the IR?
- ▶ How does LLVM's replaceAllUsesWith work?
- ▶ What is the benefit of instr. flags like nsw?
- ▶ Why are floating-point optimizations problematic?
- ▶ How is C++ `[[likely]]` represented in LLVM-IR?

# Code Generation for Data Processing

## Lecture 7: Vectorization

Alexis Engelke

Chair of Data Science and Engineering (I25)  
School of Computation, Information, and Technology  
Technical University of Munich

Winter 2025/26

# Parallel Data Processing

- ▶ Sequential execution has inherently limited performance
  - ▶ Clock rate, energy consumption/cooling, data path lengths, speed of light, ...
- ▶ Parallelism is the key to substantial and scalable perf. improvements
- ▶ Modern systems have many levels of parallelism:
  - ▶ Multiple nodes/systems, connected via network
  - ▶ Different compute units (CPU, GPU, etc.), connected via PCIe
  - ▶ Multiple CPU sockets, connected via QPI (Intel) or HyperTransport (AMD)
  - ▶ Multiple CPU cores
  - ▶ Multiple threads per core
  - ▶ Instruction-level parallelism (superscalar out-of-order execution)
  - ▶ Data parallelism (SIMD)

# Single Instruction, Multiple Data (SIMD)

- ▶ Idea: perform same operations on multiple data in parallel
- ▶ First computer with SIMD operations: MIT Lincoln Labs TX-2, 1957<sup>35</sup>
- ▶ Wider use in HPC in 1970s with vector processors (Cray et al.)
  - ▶ Ultimately replaced by much more scalable distributed machines
- ▶ SIMD-extensions for multimedia processing from 1990s onwards
  - ▶ Often include very special instructions for image/video/audio processing
- ▶ Shift towards HPC and data processing around 2010
- ▶ Extensions for machine learning/AI in late 2010s

<sup>35</sup>W Clark et al. *The Lincoln TX-2 Computer*. Apr. 1957. 

# SIMD: Idea

- ▶ Multiple data elements are stored in *vectors*
  - ▶ Size of data may differ, vector size is typically constant
  - ▶ Single elements in vector referred to as *lane*
- ▶ (Vertical) Operations apply the same operation to all lanes

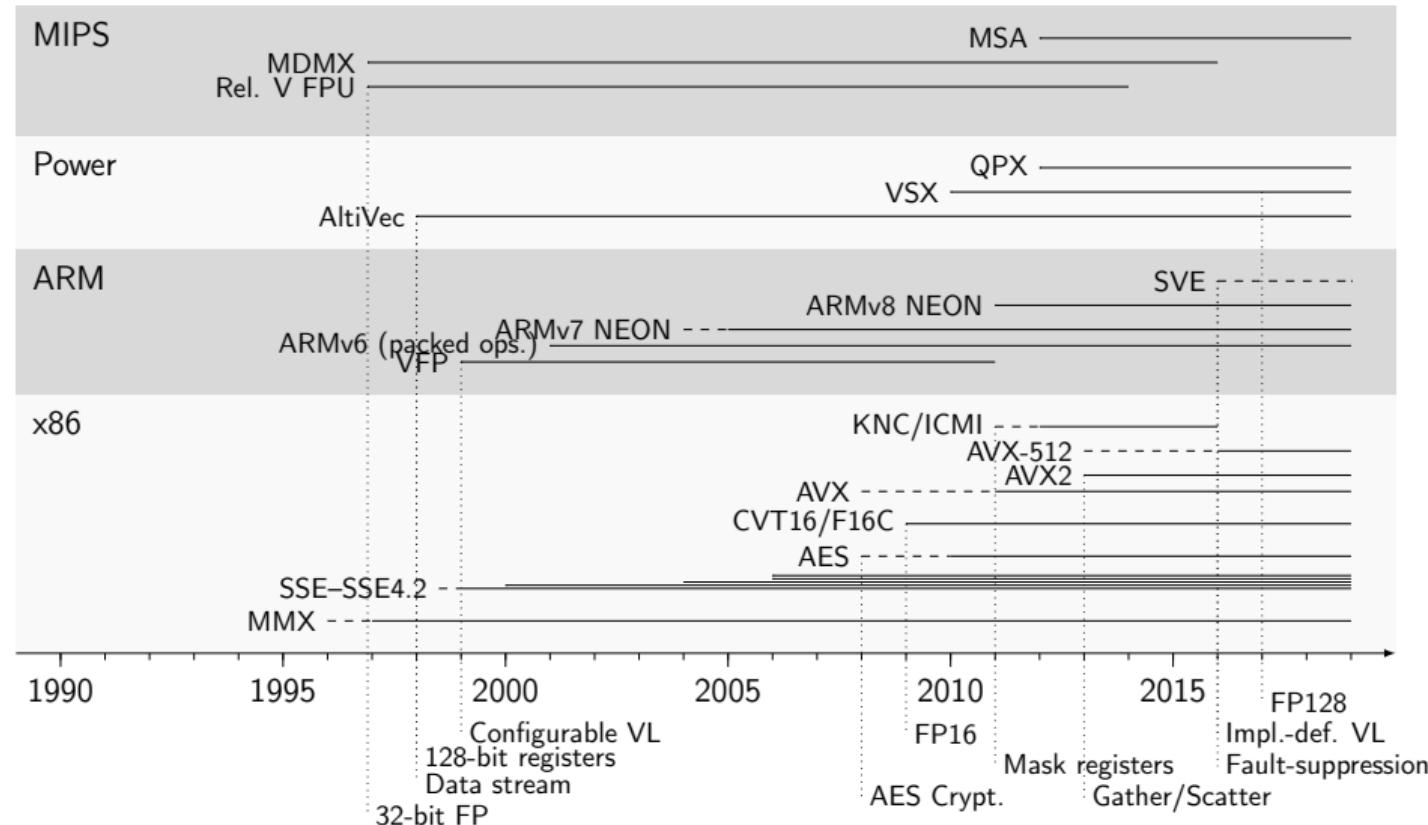
|        | lane 3 | lane 2 | lane 1 | lane 0 |
|--------|--------|--------|--------|--------|
| src 1  | 1      | 2      | 3      | 4      |
|        | +      | +      | +      | +      |
| src 2  | 1      | 2      | 3      | 4      |
|        | ↓      | ↓      | ↓      | ↓      |
| result | 2      | 4      | 6      | 8      |

- ▶ Horizontal operations work on neighbored elements

# SIMD ISAs: Design

- ▶ Vectors are often implemented as fixed-size wide registers
  - ▶ Examples: ARM NEON  $32 \times 128$ -bit, Power QPX  $32 \times 256$ -bit
  - ▶ Data types and element count is defined by instruction
- ▶ Some ISAs have dynamic vector sizes: ARM VFP, ARM SVE, RISC-V V
  - ▶ Problematic for compilers: variable spill size, less constant folding
- ▶ Data types vary, e.g. i8/i16/i32/i64/f16/bf16/f32/f64/f128
  - ▶ Sometimes only conversion, sometime with saturating arithmetic
- ▶ Masking allows to suppress operations for certain lanes
  - ▶ Dedicated mask registers (AVX-512, SVE, RVV) allow for hardware masking
  - ▶ Can also apply for memory operations, optionally suppressing faults
  - ▶ Otherwise: software masking with another vector register

# Historical Development of SIMD Extensions



## SIMD: Use Cases

- ▶ Dense linear algebra: vector/matrix operations
  - ▶ Implementations: Intel MKL, OpenBLAS, ATLAS, ...
- ▶ Sparse linear algebra
  - ▶ Needs gather/scatter instructions
- ▶ Image and video processing, manipulation, encoding
- ▶ String operations
  - ▶ Implemented, e.g., in glibc, simdjson
- ▶ Cryptography

# SIMD ISAs: Usage Considerations

- ▶ Very easy to implement in hardware
  - ▶ Simple replication of functional units and larger vector registers
  - ▶ Too large vectors, however, also cause problems (AVX-512)
- ▶ Offer significant speedups for certain applications
  - ▶ With 4x parallelism, speed-ups of  $\sim 3x$  are achievable
  - ▶ Amdahl's Law applies, unfortunately
- ▶ Caveat: non-trivial to program
  - ▶ Optimized routines provided by libraries
  - ▶ Compilers try to auto-vectorize, but often need guidance

# SIMD Programming: (Inline) Assembly

- ▶ Idea: SIMD is too complicated, let programmer handle this
- ▶ Programmer specifies exact code (instrs, control flow, and registers)
- ▶ Inline assembly allows for integration into existing code
  - ▶ Specification of register constraints and clobbers needed
- ▶ “Popular” for optimized libraries

**+** Allows for best performance

**-** Very tedious to write, manual register allocation, non-portable

**-** No optimization across boundaries

# SIMD Programming: Intrinsics

- ▶ Idea: deriving a SIMD schema is complicated, delegate to programmer
- ▶ Intrinsic functions correspond to hardware instructions
  - ▶ `__m128i _mm_add_epi32 (__m128i a, __m128i b)`
- ▶ Programmer explicitly specifies vector data processing instructions  
compiler supplements registers, control flow, and scalar processing

- + Allows for very good performance, still exposes all operations
- ~ Compiler can to some degree optimize intrinsics
  - ▶ GCC does not; Clang/LLVM does – intrinsics often lowered to LLVM-IR vectors (which also has some problems)
- Tedious to write, non-portable

# SIMD Programming: Intrinsics – Example

```
float sdot(size_t n, const float x[n], const float y[n]) {
    size_t i = 0;
    __m128 sum = _mm_set_ps1(0);
    for (i = 0; i < (n & ~(size_t)3ul); i += 4) {
        __m128 xl = _mm_loadu_ps(&x[i]);
        __m128 yl = _mm_loadu_ps(&y[i]);
        sum = _mm_add_ps(sum, _mm_mul_ps(xl, yl));
    }
    // ... take care of tail (i..<n) ...
}
```

## Intrinsics for Unknown Vector Size

- ▶ Size not known at compile-time, but can be queried at runtime
  - ▶ SVE: instruction `incd` adds number of vector lanes to register
- ▶ In C: behave like an incomplete type, except for parameters/returns
- ▶ Flexible code often slower than with assumed constant vector size
- ▶ Consequences:
  - ▶ Cannot put such types in structures, arrays, `sizeof`
  - ▶ Stack spilling implies variably-sized stack
- ▶ Instructions to set mask depending on bounds: `whilelt`, ...
  - ▶ No loop peeling for tail required

# SIMD Programming: Target-independent Vector Extensions

- ▶ Idea: vectorization still complicated, but compiler can choose instrs.
  - ▶ Programmer still specifies exact operations, but in target-independent way
  - ▶ Often mixable with target-specific intrinsics
- ▶ Compiler maps operations to actual target instructions
- ▶ If no matching target instruction exists, use replacement code
  - ▶ Inherent danger: might be less efficient than scalar code
- ▶ Often relies on explicit vector size

# GCC Vector Extensions

Compile<sup>36</sup> the following operations and observe how the output changes:

- ▶ Add 16-byte vectors of element type `uint32_t`
- ▶ Multiply 8-byte vectors of element type `uint32_t/uint8_t`
- ▶ Divide 64-byte vectors of element type `uint32_t/long double`

```
// compile with: clang -O3 -S --target=x86_64 file.c -o -
// also try --target=aarch64
#include <stdint.h>
typedef uint32_t vecty __attribute__((vector_size(16)));
vecty op(vecty a, vecty b) {
    return a + b;
}
```

<sup>36</sup><https://godbolt.org/z/43v98rdno>

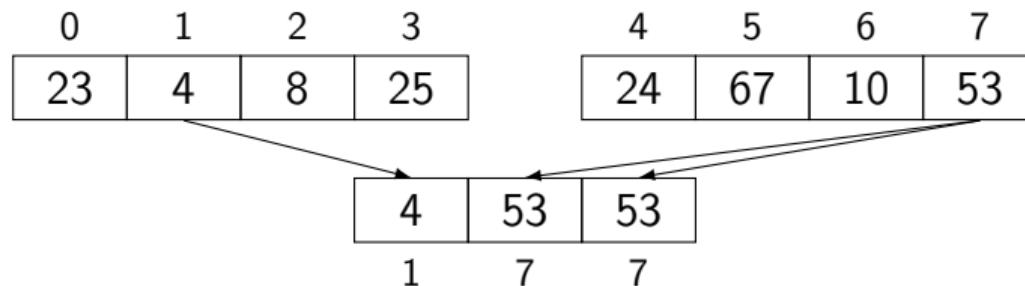
# LLVM-IR: Vectors

- ▶  $\langle N \times \text{ty} \rangle$  – fixed-size vector type, e.g.  $\langle 4 \times \text{i32} \rangle$ 
  - ▶ Valid element type: integer, floating-point, pointers
  - ▶ Memory layout: densely packed (i.e.,  $\langle 8 \times \text{i2} \rangle \approx \text{i16}$ )
- ▶  $\langle \text{vscale} \times N \times \text{ty} \rangle$  – scalable vector, e.g.  $\langle \text{vscale} \times 4 \times \text{i32} \rangle$ 
  - ▶ Vector with a multiple of N elements
  - ▶ Intrinsic `@llvm.vscale.i32()` – get runtime value of `vscale`
- ▶ Most arithmetic operations can also operate on vectors
- ▶ `insertelement/extractelement`: modify single element
  - ▶ Example: `%4 = insertelement <4 x float> %0, float %1, i32 %2`
  - ▶ Index can be non-constant value

## LLVM-IR: shufflevector

- ▶ Instruction to reorder values and resize vectors
- ▶ `shufflevector <n x ty> %x, <n x ty> %y, <m x i32> %mask`
  - ▶ `%x, %y` – values to shuffle, must have same size
  - ▶ `%mask` – element indices for result (0.. $<\text{n}$  refer to `%x`,  $\text{n}..<2\text{n}$  to `%y`)
  - ▶ Result is of type `<m x ty>`

`shufflevector <4 x i32> %x, <4 x i32> %y, <3 x i32> <i32 1, i32 7, i32 7>`



## shufflevector: Examples

What do these instructions do and what is the result type?

1. `%a = insertelement <4 x i32> poison, i32 %x, i64 0`  
`%r = shufflevector <4 x i32> %a, <4 x i32> poison,`  
`<4 x i32> zeroinitializer`
2. `%r = shufflevector <4 x i32> %a, <4 x i32> %b,`  
`<4 x i32> <i32 0, i32 5, i32 2, i32 7>`
3. `%r = shufflevector <4 x i16> %a, <4 x i16> %b,`  
`<8 x i32> <i32 0, i32 1, i32 2, i32 3, i32 4, i32 5, i32 6, i32 7>`

► Transform this LLVM-IR function into scalar, idiomatic C code

```
define void @foo(ptr %0, ptr %1) {
  br label %3
3: %4 = phi i64 [ 0, %2 ], [ %12, %3 ]
  %5 = phi <4 x i64> [ <i64 0, i64 1, i64 2, i64 3>, %2 ], [ %13, %3 ]
  %6 = getelementptr inbounds i64, ptr %1, i64 %4
  %7 = load <4 x i64>, ptr %6, align 8
  %8 = icmp slt <4 x i64> %7, %5
  %9 = add nsw <4 x i64> %7, %5
  %10 = select <4 x i1> %8, <4 x i64> %9, <4 x i64> zeroinitializer
  %11 = getelementptr inbounds i64, ptr %0, i64 %4
  store <4 x i64> %10, ptr %11, align 8
  %12 = add nuw i64 %4, 4
  %13 = add <4 x i64> %5, <i64 4, i64 4, i64 4, i64 4>
  %14 = icmp eq i64 %12, 2048
  br i1 %14, label %15, label %3
15: ret void
}
```

# LLVM-IR: Lowering Intrinsics

- ▶ Intrinsics translated to native LLVM-IR if possible
- ✚ Allows optimizations
- Intent of programmer might get lost

```
#include <immintrin.h>
__m128 func(__m128 a, __m128 b) {
    __m128 rev = _mm_shuffle_epi32(a + b, 0x1b);
    return _mm_round_ps(rev, _MM_FROUND_TO_NEG_INF);
}

define <4 x float> @func(<4 x float> %0, <4 x float> %1) {
    %3 = fadd <4 x float> %0, %1
    %4 = shufflevector <4 x float> %3, <4 x float> poison, <4 x i32> <i32 3, i32 2, i32 1, i32 0>
    %5 = tail call <4 x float> @llvm.x86.sse41.round.ps(<4 x float> %4, i32 1)
    ret <4 x float> %5
}
declare <4 x float> @llvm.x86.sse41.round.ps(<4 x float>, i32 immarg)
```

# SIMD Programming: Single Program, Multiple Data (SPMD)

- ▶ So far: manual vectorization
- ▶ Observation: same code is executed on multiple elements
- ▶ Idea: tell compiler to vectorize handling of single element
  - ▶ Splice code for element into separate function
  - ▶ Tell compiler to generate vectorized version of this function
  - ▶ Function called in vector-parallel loop
- ▶ Needs annotation of variables
  - ▶ Varying: variables that differ between lanes
  - ▶ Uniform: variables that are guaranteed to be the same  
(basically: scalar values that are broadcasted if necessary)

# SPMD: Example (OpenMP)

```
#pragma omp declare simd
int foo(int x, int y) {
    return x + y;
}
```

- ▶ Compiler generates version that operates on vector

```
foo:
    add edi, esi
    mov eax, edi
    ret
```

```
_ZGVxN4vv_foo:
    paddd xmm0, xmm1
    ret
```

# SPMD: Example (OpenMP)

```
#pragma omp declare simd uniform(y)
int foo(int x, int y) {
    return x + y;
}
```

- ▶ Uniform: always same value

```
foo:
    add edi, esi
    mov eax, edi
    ret

_ZGVxN4vu_foo:
    movd xmm1, eax
    pshufd xmm2, xmm1, 0
    paddd xmm0, xmm2
    ret
```

## SPMD: Example (OpenMP) – if/else

```
#pragma omp declare simd
int foo(int x, int y) {
    int res;
    if (x > y) res = x;
    else res = y - x;
    return res;
}
```

- ▶ Diverging control flow:  
all paths are executed

```
foo:
    mov eax, esi
    sub eax, edi
    cmp edi, esi
    cmovg eax, edi
    ret

_ZGVxN4vv_foo:
    movdqa xmm2, xmm0
    pcmppgtd xmm0, xmm1
    psubd xmm1, xmm2
    pblendvb xmm1, xmm2, xmm0
    movdqa xmm0, xmm1
    ret
```

## SPMD to SIMD: Handling if/else

- ▶ Control flow solely depending on uniforms: nothing different
- ▶ Otherwise: control flow may diverge
  - ▶ Different lanes may choose different execution paths
  - ▶ But: CPU has only one control flow, so all paths must execute
- ▶ Condition becomes mask, mask determines result
- ▶ After insertion of masks, linearize control flow
  - ▶ Relevant control flow now encoded in data through masks
- ▶ Problem: side-effects prevent vectorization

# SPMD to SIMD: Handling Loops

- ▶ Uniform loops: nothing different
- ▶ Otherwise: need to retain loop structure
  - ▶ “active” mask added to all loop iterations
  - ▶ Loop only terminates once all lanes terminate (active is zero)
  - ▶ Lanes that terminated early need their values retained
- ▶ Approach also works for nested loops/conditions
- ▶ Irreducible loops need special handling<sup>37</sup>

<sup>37</sup> R Karrenberg and S Hack. “Whole-function vectorization”. In: CGO. 2011, pp. 141–150.

# SPMD Implementations on CPUs

- ▶ OpenMP SIMD functions
  - ▶ Need to be combined with `#pragma omp simd` loops
- ▶ Intel ispc<sup>38</sup> (Implicit SPMD Program Compiler)
  - ▶ Extension of C with keywords `uniform`, `varying`
  - ▶ Still active and interesting history<sup>39</sup>
- ▶ OpenCL on CPU
  - ▶ Very similar programming model
  - ▶ But: higher complexity for communicating with rest of application

<sup>38</sup> M Pharr and WR Mark. “ispc: A SPMD compiler for high-performance CPU programming”. In: *InPar.* 2012, pp. 1–13.

<sup>39</sup> <https://pharr.org/matt/blog/2018/04/30/ispc-all>

# SIMD Programming: SPMD on CPUs

- ▶ Semi-explicit vectorization
- ▶ Programmer chooses level of vectorization
  - ▶ E.g., inner vs. outer loop
- ▶ Compiler does actual work

- + Allows simple formulation of complex control flow
- Compilers often fail at handling complex control flow well
  - ▶ Loops are particularly problematic

# SIMD Programming: Auto-vectorization

- ▶ Idea: programmer is too incompetent/busy, let compiler do vectorization
- ▶ Inherently difficult and problematic, after decades of research
  - ▶ Recognizing and matching lots of patterns
  - ▶ Instruction selection becomes more difficult
  - ▶ Compiler lacks domain knowledge about permissible transformations
- ▶ Executive summary of the state of the art:
  - ▶ Auto-vectorization works well for very simple cases
  - ▶ For “medium complexity”, code is often suboptimal
  - ▶ In many cases, auto-vectorization fails on unmodified code

# Auto-vectorization Strategies

- ▶ Loop Vectorization
  - ▶ Try to transform loop body into vectors with  $n$  lanes
  - ▶ Often needs tail loop for remainder that doesn't fill a vector
  - ▶ Extremely common
- ▶ Superword-level Parallelism (SLP)
  - ▶ Vectorize constructs outside of loops
  - ▶ Detect neighbored stores, try to fold operations into vectors

# Loop Vectorization: Strategy

- ▶ Only consider innermost loop (at first)

1. Check legality: is vectorization possible at all?
  - ▶ Only vectorizable data types and operations used
  - ▶ No loop-carried dependencies, overlapping memory regions, etc.
2. Check profitability: is vectorization beneficial?
  - ▶ Consider: runtime checks, gather/scatter, masked operations, etc.
  - ▶ Needs information about target architecture
3. Perform transformation

## Outer Loop Vectorization

- ▶ Vectorizing the innermost loop not always beneficial
  - ▶ Example 1: inner loop has only few iterations
  - ▶ Example 2: inner loop has loop-carried dependencies
- ▶ Thus: need to consider outer loops as well
  - ▶ Also: vectorization on multiple levels might be beneficial
- ▶ Very limited support in compilers, if any

## Auto-vectorization is Hard

- ▶ Biggest problem: data dependencies
  - ▶ Resolving loop-carried dependencies is difficult
- ▶ Memory aliasing
  - ▶ Overlapping arrays, or – worse – loop counter
- ▶ Which loop level to vectorize? Multiple?
- ▶ Loop body *might* impact loop count
- ▶ Function calls, e.g. for math functions
- ▶ Strided memory access (e.g., only every n-th element)
- ▶ Choosing vectorization level (outer loop *might* be better)
- ▶ Is vectorization profitable *at all*?
- ▶ Often black box to programmer, preventing fine-grained tuning

# Auto-Vectorization: Examples

Compile<sup>40</sup> the functions from ex07.txt with vectorization remarks.

```
clang -S -emit-llvm -O3 -Rpass=loop-vectorize  
-Rpass-analysis=loop-vectorize -Rpass-missed=loop-vectorize
```

- ▶ Does vectorization occur?
- ▶ What additional output is provided in the optimization remarks?
- ▶ If so: what is vectorized? How?
- ▶ Does the result match your expectation?

<sup>40</sup><https://godbolt.org/z/aqbb937cb>

## Vectorization – Summary

- ▶ SIMD is an easy way to improve performance numbers of CPUs
- ▶ Most general-purpose ISAs have one or more SIMD extensions
- ▶ Recent trend: variably-length vectors
- ▶ Inline Assembly: easiest for compiler, but extremely tedious
- ▶ Intrinsics: best trade-off towards performance and usability
- ▶ Target-independent operations: slightly increase portability
- ▶ SPMD: strategy dominant for GPU programming
- ▶ Auto-vectorization: very hard, unsuited for complex code

## Vectorization – Questions

- ▶ Why do modern CPUs provide SIMD extensions?
- ▶ Why do variable-length SIMD extensions have higher runtime costs?
- ▶ How are SIMD intrinsics lowered to LLVM-IR?
- ▶ What is the downside of target-independent vector operations?
- ▶ How can if/else/for constructs be vectorized?
- ▶ What is the difference between a uniform and a varying variable?
- ▶ Why is auto-vectorization often sub-par to manual optimization?

# Code Generation for Data Processing

## Lecture 8: Instruction Selection

Alexis Engelke

Chair of Data Science and Engineering (I25)  
School of Computation, Information, and Technology  
Technical University of Munich

Winter 2025/26

# Code Generation – Overview

- ▶ Instruction Selection
  - ▶ Map IR to assembly
  - ▶ Keep code shape and storage; change operations
- ▶ Instruction Scheduling
  - ▶ Optimize order to hide latencies and reduce register pressure
  - ▶ Keep operations unmodified
- ▶ Register Allocation
  - ▶ Map virtual to architectural registers and stack
  - ▶ Adds operations (spilling), changes storage

# Instruction Selection (ISel) – Overview

- ▶ Find machine instructions to implement abstract IR
- ▶ Typically separated from scheduling and register allocation
- ▶ Input: IR code with abstract instructions
- ▶ Output: lower-level IR code with target machine instructions

```
i64 %10 = add %8, %9
i8 %11 = trunc %10
i64 %12 = const 24
i64 %13 = add %7, %12
store %11, %13
```

```
i64 %10 = ADD %8, %9
STRB %10, [%7+24]
```

## ISel – Typical Constraints

- ▶ Target offers multiple ways to implement operations
  - ▶ `imul x, 2, add x, x, shl x, 1, lea x, [x+x]`
- ▶ Target operations have more complex semantics
  - ▶ E.g., combine truncation and offset computation into store
  - ▶ Can have multiple outputs, e.g. value+flags, quotient+remainder
- ▶ Target has multiple register sets, e.g. GP and FP/SIMD
  - ▶ Important to consider even before register allocation
- ▶ Target requires specific instruction sequences
  - ▶ E.g., for macro fusion
  - ▶ Often represented as pseudo-instructions until assembly writing

# Optimal ISel

- ▶ Find *most performant* instruction sequence with same semantics (?)
  - ▶ I.e., no program with better “performance” exists
  - ▶ Performance  $\approx$  instructions associated with specific costs
- ▶ Problem: optimal code generation is **undecidable**
- ▶ Alternative: optimal *tiling* of IR with machine code instructions
  - ▶ IR as dataflow graph, instr. tiles to optimally cover graph
  - ▶  $\mathcal{NP}$ -complete<sup>41</sup>
  - ▶ Additional complication: many different ways to express same computation

<sup>41</sup>DR Koes and SC Goldstein. “Near-optimal instruction selection on DAGs”. In: CCGO. 2008, pp. 45–54. 

# Avoiding ISel Altogether

Use an interpreter

- + Fast “compilation time”, easy to implement
- Slow execution time
- ▶ Best if code is executed once

# Macro Expansion

- ▶ Expand each IR operation with corresponding machine instrs

|                                |   |  |
|--------------------------------|---|--|
| $\%5 = \text{add } \%1, 12345$ | → | $\%5a = \text{movz } 12345$                            |
| $\%5 = \text{add } \%1, \%5a$  |   |  |
| $\%6 = \text{and } \%2, 7$     | → | $\%6 = \text{and } \%2, 7$                             |
|                                |   | $\%7a = \text{lsl } \%5, \%6$                          |
| $\%7 = \text{shl } \%5, \%6$   | → | $\%7b = \text{cmp } \%6, 64$                           |
|                                |   | $\%7 = \text{csel } \%7a, \text{xzr}, \%7b, \text{lo}$ |

## Macro Expansion

- ▶ Oldest approach, historically also does register allocation
  - ▶ Also possible by walking AST
- + Very fast, linear time, simple to implement, easy to port
- Inefficient and large output code
- ▶ Used by, e.g., LLVM FastISel, Go, GCC

# Peephole Optimization

- ▶ Plain macro expansion leads to suboptimal results
- ▶ Idea: replace inefficient instruction sequences<sup>42</sup>
- ▶ Originally: physical window over assembly code
  - ▶ Replace with more efficient instructions having same effects
  - ▶ Possibly with allocated registers
- ▶ Extension: do expansion before register allocation<sup>43</sup>
  - ▶ Expand IR into Register Transfer Lists (RTL) with temporary registers
  - ▶ While *combining*, ensure that each RTL can be implemented as single instr.

<sup>42</sup>WM McKeeman. "Peephole optimization". In: *CACM* 8.7 (1965), pp. 443–444. .

<sup>43</sup>JW Davidson and CW Fraser. "Code selection through object code optimization". In: *TOPLAS* 6.4 (1984), pp. 505–526. .

# Peephole Optimization

- ▶ Originally covered only adjacent instructions
- ▶ Can also use logical window of data dependencies
  - ▶ Problem: instructions with multiple uses
  - ▶ Needs more sophisticated matching schemes for data deps.  
⇒ Tree-pattern matching
- + Fast, also allows for target-specific sequences
- Pattern set grows large, limited potential
- ▶ Widely used today at different points during compilation

## ISel as Graph Covering – High-level Intuition

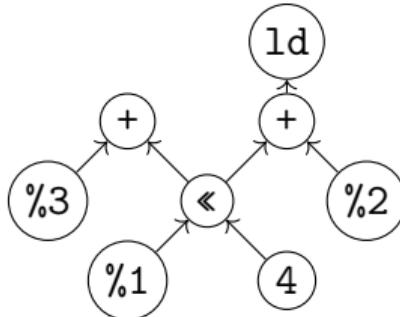
- ▶ Idea: represent program as data flow graph
- ▶ Tree: expression, comb. of single-use SSA instructions (local ISel)
- ▶ DAG: data flow in basic block, e.g. SSA block (local ISel)
- ▶ Graph: data flow of entire function, e.g. SSA function (global ISel)
- ▶ ISA “defines” *pattern set* of trees/DAGs/graphs for instrs.
- ▶ Cover data flow tree/DAG/graph with least-cost combination of patterns
  - ▶ Patterns in data flow graph may overlap
  - ▶ For non-global ISel: values used outside of block must be generated

# Tree Covering: Converting SSA into Trees

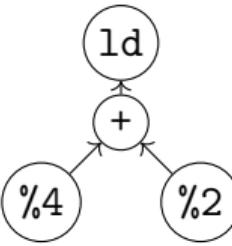
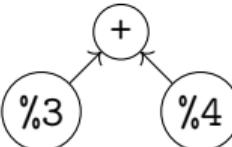
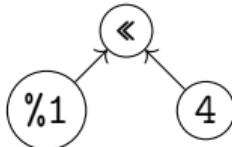
► SSA form:

```
%4 = shl %1, 4
%5 = add %2, %4
%6 = add %3, %4
%7 = load %5
live-out: %6, %7
```

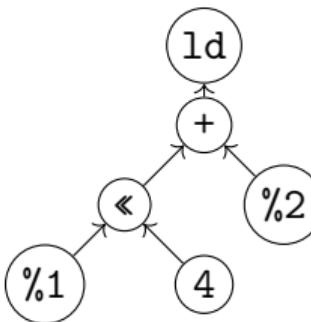
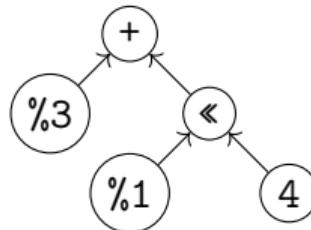
► Data flow graph:



► Method 1:  
Edge Splitting



► Method 2:  
Node Duplication



# Tree Covering: Patterns

| Pattern  |   | Cost | Instruction                            |
|----------|---|------|--|
| $P_0$    | $GP_{R1} \rightarrow \ll(GP_{R2}, K_1)$                       | 1    | lsl $R_1, R_2, \#K_1$                  |
| $P_1$    | $GP_{R1} \rightarrow +(GP_{R2}, GP_{R3})$                     | 1    | add $R_1, R_2, R_3$                    |
| $P_2$    | $GP_{R1} \rightarrow +(GP_{R2}, \ll(GP_{R3}, K_1))$           | 2    | add $R_1, R_2, R_3, \text{lsl } \#K_1$ |
| $P_3$    | $GP_{R1} \rightarrow +( \ll(GP_{R2}, K_1), GP_{R2})$          | 2    | add $R_1, R_3, R_2, \text{lsl } \#K_1$ |
| $P_4$    | $GP_{R1} \rightarrow \text{ld}(GP_{R2})$                      | 2    | ldr $R_1, [R_2]$                       |
| $P_5$    | $GP_{R1} \rightarrow \text{ld}(+(GP_{R2}, GP_{R3}))$          | 2    | ldr $R_1, [R_2, R_3]$                  |
| $P_6$    | $GP_{R1} \rightarrow \text{ld}(+(GP_{R2}, \ll(GP_{R3}, 3)))$  | 3    | ldr $R_1, [R_2, R_3, \text{lsl } \#3]$ |
| $P_7$    | $GP_{R1} \rightarrow \text{ld}(+( \ll(GP_{R2}, 3), GP_{R3}))$ | 3    | ldr $R_1, [R_3, R_2, \text{lsl } \#3]$ |
| $P_8$    | $GP_{R1} \rightarrow *(GP_{R2}, GP_{R3})$                     | 3    | madd $R_1, R_2, R_3, \text{xzr}$       |
| $P_9$    | $GP_{R1} \rightarrow +(*(GP_{R2}, GP_{R3}), GP_{R4})$         | 3    | madd $R_1, R_2, R_3, R_4$              |
| $P_{10}$ | $GP_{R1} \rightarrow K_1$                                     | 1    | mov $R_1, K_1$                         |
| :        | :   | :    | :                                      |

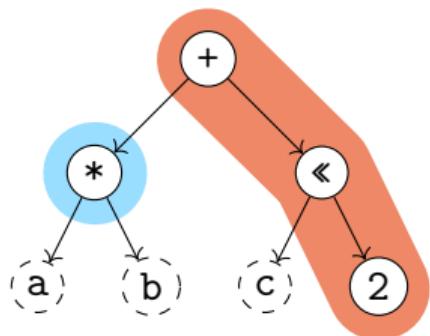
## Tree Covering: Greedy/Maximal Munch

- ▶ Top-down always take largest pattern
- ▶ Repeat for sub-trees, until everything is covered

- + Easy to implement, fast
- Result might be non-optimum

# Tree Covering: Greedy/Maximal Munch – Example



Matching Patterns:

- ▶ +:  $P_1$  – cost 1 – covered nodes: 1
- ▶ +:  $P_2$  – cost 2 – covered nodes: 3
- ▶ +:  $P_9$  – cost 3 – covered nodes: 2
- ▶ \*:  $P_8$  – cost 3 – covered nodes: 1 – best

Total cost: 5

```
madd %1, %a, %b, xzr
add %2, %1, %c, lsl #2
```

# Tree Covering: with LR-Parsing

- ▶ Can we use (LR-)parsing for instruction selection? Yes!<sup>44</sup>
  - ▶ Pattern set = grammar; IR (in prefix notation) = input

## Advantages

- ▶ Possible in linear time
- ▶ Can be formally verified
- ▶ Implementation can be generated automatically

## Disadvantages

- ▶ Constraints must map to non-terminals
  - ▶ Constant ranges, reg types, ...
- ▶ CISC: handle all operand combinations
  - ▶ Large grammar (impractical)
  - ▶ Refactoring into non-terminals
- ▶ Ambiguity hard to handle optimally

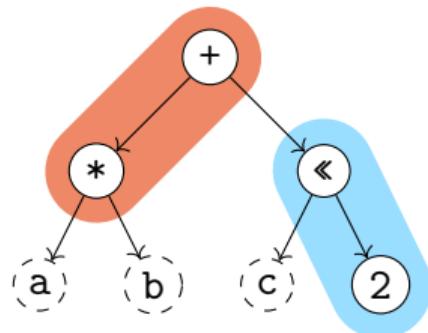
<sup>44</sup> RS Glanville and SL Graham. "A new method for compiler code generation". In: *POPL*. 1978, pp. 231–254. .

# Tree Covering: Dynamic Programming<sup>45</sup>

- ▶ Step 1: compute cost matrix, bottom-up for all nodes
  - ▶ Matrix: tree node  $\times$  register bank  
(different patterns might yield the same result in different register banks)
  - ▶ Cost is sum of pattern and sum of children costs
  - ▶ Always store cheapest rule and cost
- ▶ Step 2: walk tree top-down using rules in matrix
  - ▶ Start with goal, follow rules in matrix
- ▶ Time linear w.r.t. tree size

<sup>45</sup> AV Aho, M Ganapathi, and SWK Tjiang. "Code generation using tree matching and dynamic programming". In: *TOPLAS* 11.4 (1989), pp. 491–516. 

## Tree Covering: Dynamic Programming – Example



Node: +

Pattern:  $P_9: GP \rightarrow +(*(GP, GP), GP)$

Pat. Cost: 3

Cost Sum: 4

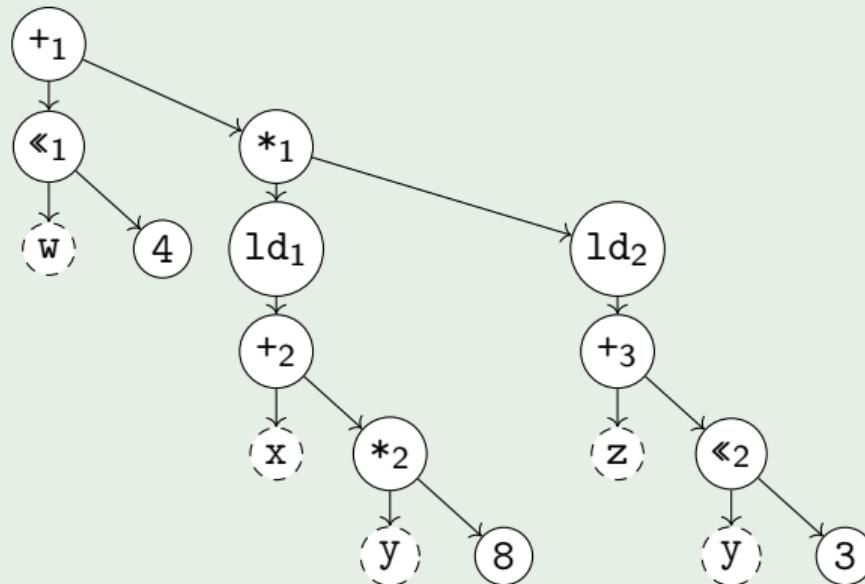
---

|    | Node    | +     | *     | <<    | 2        |
|----|---------|-------|-------|-------|----------|
| GP | Cost    | 4     | 3     | 1     | 1        |
|    | Pattern | $P_9$ | $P_8$ | $P_0$ | $P_{10}$ |

---

# Tree Covering: Dynamic Programming – Example

Find a tree covering using the dynamic programming algorithm.



# Tree Covering: Dynamic Programming – Off-line Analysis

- ▶ Cost analysis can actually be *precomputed*<sup>46</sup>
- ▶ Idea: annotate each node with a state based on child states
- ▶ Lookup node label from precomputed table (one per register bank)
- ▶ Significantly improves compilation time
- ▶ But: Tables can be large, need to cover all possible (sub-)trees
- ▶ Variation: dynamically compute and cache state tables<sup>47</sup>

<sup>46</sup> A Balachandran, DM Dhamdhere, and S Biswas. "Efficient retargetable code generation using bottom-up tree pattern matching". In: *Computer Languages* 15.3 (1990), pp. 127–140.

<sup>47</sup> MA Ertl, K Casey, and D Gregg. "Fast and flexible instruction selection with on-demand tree-parsing automata". In: *PLDI 41.6* (2006), pp. 52–60.

# Tree Covering

- + Efficient: linear time to find local optimum
- + Better code than pure macro expansion
- + Applicable to many ISAs
- Common sub-expressions cannot be represented
  - ▶ Need either edge split (prevents using complex instructions) or node duplication (redundant computation  $\Rightarrow$  inefficient code)
- Cannot make use of multi-output instructions (e.g., divmod)

# DAG Covering

- ▶ Idea: lift restriction of trees, operate on data flow DAG
  - ▶ Reminder: an SSA basic block already forms a DAG
- ▶ Trivial approach: split into trees 😞
- ▶ Least-cost covering is  $\mathcal{NP}$ -complete<sup>48</sup>

<sup>48</sup>DR Koes and SC Goldstein. "Near-optimal instruction selection on DAGs". In: CCGO. 2008, pp. 45–54. 

# DAG Covering: Adapting Dynamic Programming |<sup>49</sup>

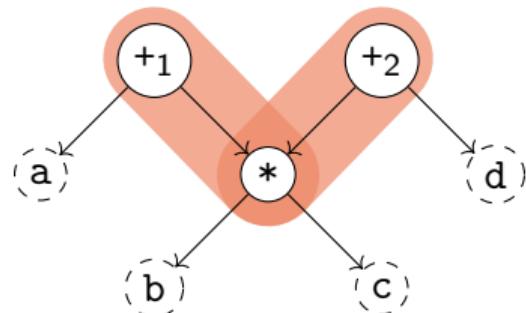
- ▶ Step 1: compute cost matrix, bottom-up for all nodes
  - ▶ As before; make sure to visit each node once
- ▶ Step 2: iterate over DAG top-down
  - ▶ Respect that multiple roots exist: start from all roots
  - ▶ Mark visited node/regbank combinations: avoid redundant emit

⊕ Linear time

⊖ Generally not optimal, only for specific grammars

<sup>49</sup>MA Ertl. "Optimal code selection in DAGs". In: *POPL*. 1999, pp. 242–249. .

# DAG Covering: Adapting Dynamic Programming I – Example



Total cost: 6

`madd %1, %b, %c, %a`

`madd %2, %b, %c, %d`

Optimal cost: 5  $\rightsquigarrow$  non-optimal result

|    | Node    | $+2$  | $+1$  | $*$   |
|----|---------|-------|-------|-------|
| GP | Cost    | 3     | 3     | 3     |
|    | Pattern | $P_9$ | $P_9$ | $P_8$ |

# DAG Covering: Adapting Dynamic Programming II<sup>50</sup>

- ▶ Step 1: compute cost matrix, bottom-up (as before)
- ▶ Step 2: iterate over DAG top-down (as before)
- ▶ Step 3: identify overlaps and check whether split is beneficial
  - ▶ Mark nodes which should not be duplicated as *fixed*
- ▶ Step 4: as step 1, but skip patterns that *include* fixed nodes
- ▶ Step 5: as step 2

- + Probably fast? “Near-optimal”?
- Generally not optimal, superlinear time

<sup>50</sup>DR Koes and SC Goldstein. “Near-optimal instruction selection on DAGs”. In: CCGO. 2008, pp. 45–54. 

## DAG Covering: ILP<sup>51</sup>

- ▶ Idea: model ISel as integer linear programming (ILP) problem
- ▶  $P$  is set of patterns with cost and edges,  $V$  are DAG nodes
- ▶ Variables:  $M_{p,v}$  is 1 iff a pattern  $p$  is rooted at  $v$

$$\begin{aligned} & \text{minimize} && \sum_{p,v} p.\text{cost} \cdot M_{p,v} \\ & \text{subject to} && \forall r \in \text{roots}. \sum_p M_{p,r} \geq 1 \\ & && \forall p, v, e \in p.\text{edges}(v). M_{p,v} - \sum_{p'} M_{p',e} \leq 0 \\ & && M_{p,v} \in \{0, 1\} \end{aligned}$$

Minimize cost for all matched patterns s.t. every root has a match and every input of a match has a match.

- + Optimal result
- Practicability beyond small programs questionable (at best)

<sup>51</sup>DR Koes and SC Goldstein. "Near-optimal instruction selection on DAGs". In: CCGO. 2008, pp. 45–54. 

## DAG Covering: Greedy/Maximal Munch

- ▶ Top-down, start at roots, always take largest pattern
- ▶ Repeat for remaining roots until whole graph is covered
- +
- Easy to implement, reasonably fast
- Result often non-optimal
- ▶ Used by: LLVM SelectionDAG

# Graph Covering

- ▶ Idea: lift limitation of DAGs, cover entire function graphs
- ▶ Better handling of predication and VLIW bundling
  - ▶ E.g., hoisting instructions from a conditional block
- ▶ Allows to handle instructions that expand to multiple blocks
  - ▶ `switch`, `select`, etc.
- ▶ May need new IR to model control flow in addition to data flow
- ▶ In practice: only used by adapting methods showed for DAGs
- ▶ Used by: Java HotSpot Server, LLVM GlobalISel (all tree-covering)

# Flawed Assumptions

- ▶ Cost model is fundamentally flawed
  - ⇒ “Optimal” ISel doesn’t really mean anything
- ▶ Out-of-order execution: costs are not linear
  - ▶ Instructions executed in parallel, might execute for free
  - ▶ Possible contention of functional units
- ▶ Register allocator will modify instructions
- ▶ “Bad” instructions boundaries increase register requirements
  - ▶ More stack spilling  $\rightsquigarrow$  much slower code!

# Instruction Selection in Practice

- ▶ Most compilers use some form of greedy tree/DAG pattern matching
- ▶ Later stages use peephole optimizations
  - ▶ Basically also tree/DAG matching on machine operations
- ▶ Distinction between tree/DAG/graph matching somewhat artificial<sup>52</sup>

Problem in practice: implementing the huge amount of required patterns

- ▶ LLVM X86 back-end has 73k lines C++ for lowering + auto-generated patterns
- ▶ Needs lots of handling for corner cases, e.g. immediates
- ▶ Coming up with the patterns is often non-trivial

<sup>52</sup>My personal opinion.

## LLVM Back-end: Overview

- ▶ LLVM-IR → Machine IR: instruction selection + scheduling
  - ▶ MIR is SSA-representation of target instructions
  - ▶ Selectors: SelectionDAG, FastISel, GlobalISel
  - ▶ Also selects register bank (GP/FP/...) – required for instruction
  - ▶ Annotates registers: calling convention, encoding restrictions, etc.
- ▶ MIR: minor (peephole) optimizations
- ▶ MIR: register allocation
- ▶ MIR: prolog/epilog insertion (stack frame, callee-saved regs, etc.)
- ▶ MIR → MC: translation to machine code

# LLVM MIR Example

```
define i64 @fn(i64 %a, i64 %b, i64 %c) {
    %shl = shl i64 %c, 2
    %mul = mul i64 %a, %b
    %add = add i64 %mul, %shl
    ret i64 %add
}
```

```
# YAML with name, registers, frame info
body: |
    bb.0 (%ir-block.0):
        liveins: $x0, $x1, $x2
        %2:gpr64 = COPY $x2
        %1:gpr64 = COPY $x1
        %0:gpr64 = COPY $x0
        %3:gpr64 = MADDXrrr %0, %1, $xzx
        %4:gpr64 = ADDXrs killed %3, %2, 2
        $x0 = COPY %4
    RET_ReallyLR implicit $x0
```

```
llc -march=aarch64 -stop-after=finalize-isel
```

## LLVM MIR Example

Analyze the Machine IR of the following code<sup>55</sup>. (Also consult the reference<sup>56</sup>.)

- ▶ What is the difference between physical and virtual registers?
- ▶ What do killed and implicit-def mean?
- ▶ How do branches and calls differ from LLVM-IR?

```
// clang --target=aarch64 -c -mllvm -stop-after=finalize-isel -O1 -o -
// Also try -O0, -O2, -g, and --target=x86_64.
[[clang::noinline]] int foo(int n) {
    int r = 1;
    while (n) { r *= n << n; n--; }
    return r;
}
int bar(int n) { return foo(n) + n; }
```

<sup>55</sup><https://godbolt.org/z/zj9WM84Wh>

<sup>56</sup> <https://llvm.org/docs/MIRLangRef.html>

# LLVM: Instruction Selectors

## FastISel

- ▶ Uses macro expansion
- ▶ Low compile-time
- ▶ Code quality poor
- ▶ Only common cases
- ▶ Otherwise: fallback to SelectionDAG
- ▶ Default for -00

## SelectionDAG

- ▶ Converts each block into separate DAGs
- ▶ Greedy tree matching
- ▶ Slow, but good code
- ▶ Handles all cases
- ▶ No cross-block opt. (done in DAG building)
- ▶ Default

## GlobalISel

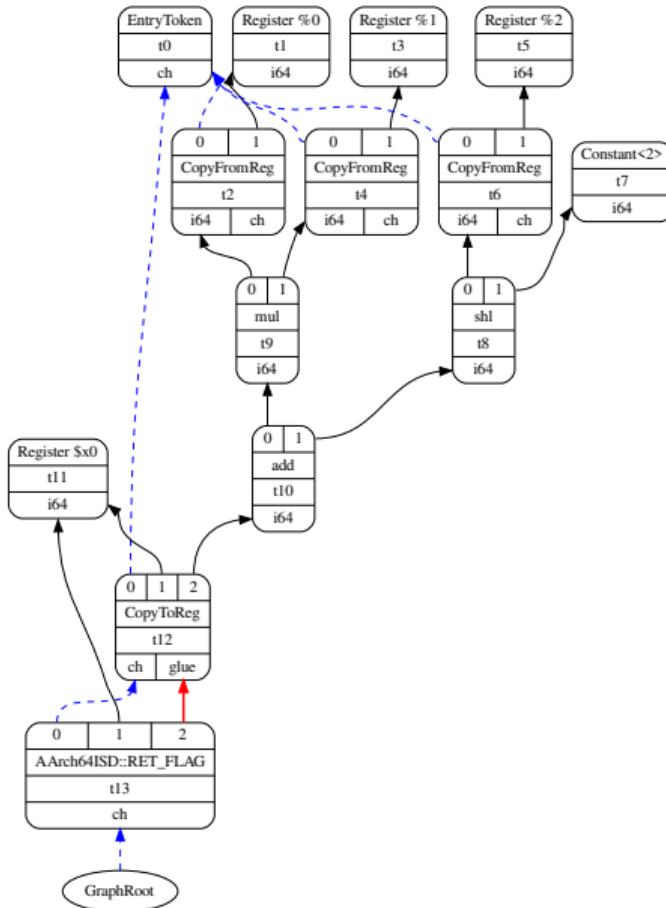
- ▶ Conv. to generic-MIR then legalize to MIR
- ▶ Reuses SD patterns
- ▶ Faster than SelDAG
- ▶ Few architectures
- ▶ Handles many cases, SelDAG-fallback
- ▶ Default AArch64 -00

# LLVM SelectionDAG: IR to ISelDAG

- ▶ Construct DAG for basic block
  - ▶ EntryToken as ordering chain
- ▶ Legalize data types
  - ▶ Integers: promote or expand into multiple
  - ▶ Vectors: widen or split (or scalarize)
- ▶ Legalize operations
  - ▶ E.g., conditional move, etc.
- ▶ Optimize DAG, e.g. some pattern matching,  
removing unneeded sign/zero extensions

```
llc -march=aarch64 -view-isel-dags
```

Note: needs LLVM debug build



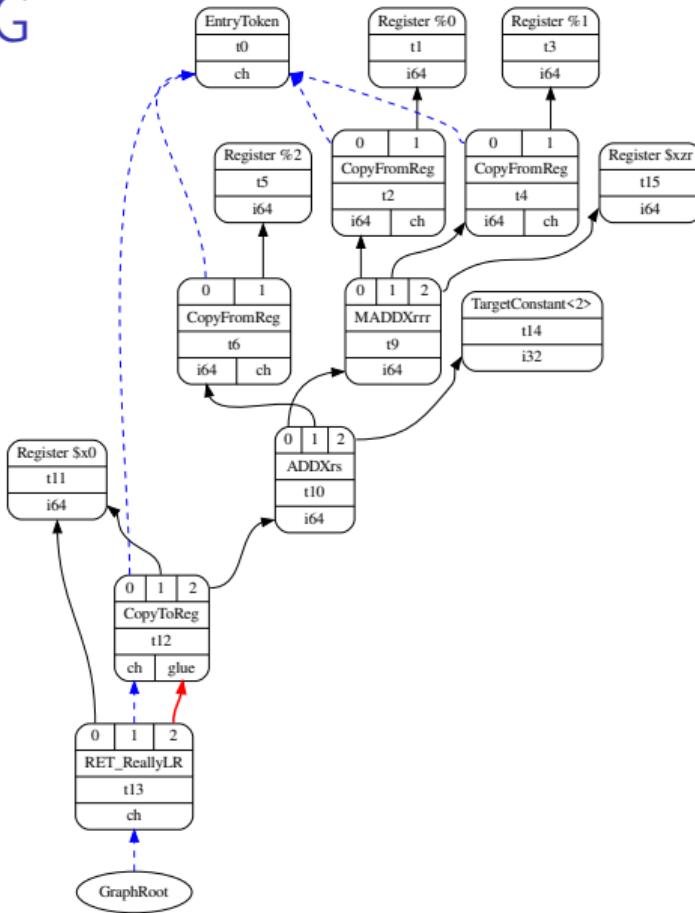
isel input for fn:

## LLVM SelectionDAG: ISelDAG to DAG

- ▶ Mainly pattern matching
- ▶ Simple patterns specified in TableGen
  - ▶ Matching/selection compiled into bytecode
  - ▶ `SelectionDAGISel::SelectCodeCommon()`
- ▶ Complex selections done in C++
- ▶ Scheduling: linearization of graph

```
llc -march=aarch64 -view-sched-dags
```

Note: needs LLVM debug build



## LLVM GlobalISel

- ▶ DAGs do have limitations w.r.t. instruction selection
- ▶ Graph loses original scheduling information (why problematic?)
- ▶ Building a separate IR is expensive
- ▶ Core idea of GlobalISel: do legalization and ISel in MachineIR
- ▶ MIR is target-specific  $\leadsto$  introduce generic operations
- ▶ Permit matching/combining instructions from different blocks

## LLVM GlobalISel: Passes

- ▶ IRTranslator: translate LLVM-IR into gMIR
  - ▶ One-to-one mapping, except call and switch
- ▶ PreLegalizerCombiner
- ▶ Localizer: try to improve placement of constant materialization
- ▶ Legalizer: replace illegal instructions with legal gMIR instrs
- ▶ PostLegalizerCombiner
- ▶ RegBankSelect: assign register bank (reduce copies between banks)
- ▶ InstructionSelect

## Instruction Selection – Summary

- ▶ Instruction Selection: transform generic into arch-specific instructions
- ▶ Often focus on optimizing tiling costs
- ▶ Target instructions often more complex, e.g., multi-result
- ▶ Macro Expansion: simple, fast, but inefficient code
- ▶ Peephole optimization on sequences/trees to optimize
- ▶ Tree Covering: allows for better tiling of instructions
- ▶ DAG/Graph Covering: support multi-res instrs., but  $\mathcal{NP}$ -complete
- ▶ Practical problems are very different from theory

# Instruction Selection – Questions

- ▶ What is the (nowadays typical) input and output IR for ISel?
- ▶ Why is good instruction selection important for performance?
- ▶ Why is peephole optimization beneficial for nearly all ISel approaches?
- ▶ How can peephole opt. be done more effectively than on neighboring instrs.?
- ▶ What are options to transform an SSA-IR into data flow trees?
- ▶ Why is a greedy strategy not optimal for tree pattern matching?
- ▶ When is DAG covering beneficial over tree covering?
- ▶ Which ISel strategies does LLVM implement? Why?
- ▶ What are advantages/disadvantages of SelDAG and GlobalISel?

# Code Generation for Data Processing

## Lecture 9: SSA Destruction and Liveness Analysis

Alexis Engelke

Chair of Data Science and Engineering (I25)  
School of Computation, Information, and Technology  
Technical University of Munich

Winter 2025/26

# Register Allocation: Overview

- ▶ Destruct SSA form = resolve  $\phi$ -nodes
- ▶ Map unlimited/virtual registers to limited/architectural registers
- ▶ When running out of registers, move values to stack

```
gauss(%0) {  
    %2 = SUBXri %0, 1  
    %3 = MADDXrrr %0, %2, 0  
    %4 = MOVXconst 2  
    %5 = SDIVrr %3, %4  
    ret %5  
}
```

```
gauss(X0) {  
    X1 = SUBXri X0, 1  
    X0 = MADDXrrr X0, X1, XZR  
    X1 = MOVXconst 2  
    X0 = SDIVrr X0, X1  
    ret X0  
}
```

# Register Allocation: Strategy Overview

- ▶ “Conventional:”  $\phi$ -node elimination before register allocation
  - ▶  $\phi$ -nodes replaced with sequences of copies
  - ▶ RegAlloc input is no longer in SSA form
  - ~~> RegAlloc is a  $\mathcal{NP}$ -hard problem
  - ▶ Widely used in practice
- ▶ “More modern:” register allocation on SSA form
  - ▶  $\phi$ -nodes eliminated after RegAlloc
  - ~~> RegAlloc becomes easier, but remains  $\mathcal{NP}$ -hard in practice
  - ▶ Mainly academic research

## SSA Destruction

- ▶ Goal: eliminate  $\phi$ -nodes
- ▶ For now, continue to assume unlimited registers
- ▶ Remember:  $\phi$ -nodes are executed on the edge
- ▶ Idea: predecessors write their value to that location at the end

## SSA Destruction: Example 1

```
identity(%0)
e:
  br %loop
loop:
  %3 = phi [ 0, %e ], [ %4, %loop ]
  %4 = add %3, 1
  %5 = cmp ult %4, %0
  br %5, %loop, %ret
ret:
  ret %3
```

```
identity(%0)
e:
  %3 = copy 0
  br %loop
loop:
  %4 = add %3, 1
  %5 = cmp ult %4, %0
  %3 = copy %4
  br %5, %loop, %ret
ret:
  ret %3
```

- ▶ Original value lost in return block!

# Lost Copy Problem

- ▶ Critical edge: edge from block with mult. succs. to block with mult. preds.
- ▶ Problem: cannot place move on such edges
  - ▶ When placing in predecessor, they would also execute for other successor  
⇒ unnecessary and – worse – incorrect



- ▶ *Break critical edges: insert an empty block*

## SSA Destruction: Example 1 – Critical Edge Split

```
identity(%0)
e:
  br %loop
loop:
  %3 = phi [ 0, %e ], [ %4, %critedge ]
  %4 = add %3, 1
  %5 = cmp ult %4, %0
  br %5, %critedge, %ret
critedge:
  br %loop
ret:
  ret %3
```

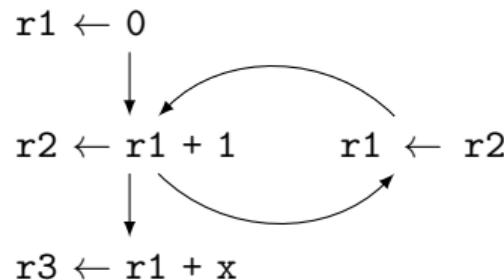
```
identity(%0)
e:
  %3 = copy 0
  br %loop
loop:
  %4 = add %3, 1
  %5 = cmp ult %4, %0
  br %5, %critedge, %ret
critedge:
  %3 = copy %4
  br %loop
ret:
  ret %3
```

- ▶ Problem fixed, but one extra branch per loop iteration

# Handling Critical Edges

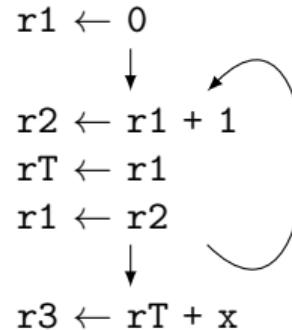
## Breaking Edges

- ▶ Insert new block for moves
- + Simple, no analyses needed
- Bad performance in loops



## Copy Used Values

- ▶ Move used values to new reg.
- + Performance might be better
- Needs more registers



## SSA Destruction: Example 1 – Value Copy

```
identity(%0)
e:
  br %loop
loop:
  %3 = phi [ 0, %e ], [ %4, %loop ]
  %4 = add %3, 1
  %5 = cmp ult %4, %0
  br %5, %loop, %ret
ret:
  ret %3
```

```
identity(%0)
e:
  %3 = copy 0
  br %loop
loop:
  %4 = add %3, 1
  %5 = cmp ult %4, %0
  %3c = copy %3
  %3 = copy %4
  br %5, %loop, %ret
ret:
  ret %3c
```

- ▶ Problem fixed; register allocator can hopefully omit a copy

## SSA Destruction: Example 2

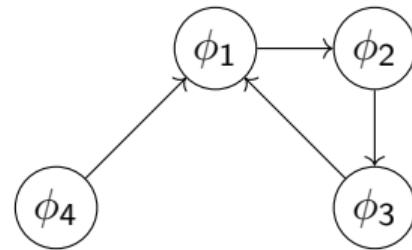
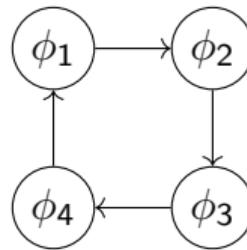
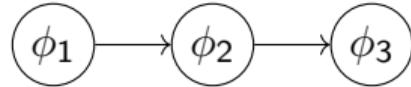
```
odd(%0)
e:
  br %loop
loop:
  %3 = phi [ %0, %e ], [ %8, %body ]
  %4 = phi [ 1, %e ], [ %5, %body ]
  %5 = phi [ 0, %e ], [ %4, %body ]
  %6 = cmp ne %3, 0
  br %6, %body, %ret
body:
  %8 = sub %3, 1
  br %loop
ret:
  ret %4
```

```
odd(%0)
e:
  %3 = copy %0
  %4 = copy 1
  %5 = copy 0
  br %loop
loop:
  %6 = cmp ne %3, 0
  br %6, %body, %ret
body:
  %8 = sub %3, 1
  %3 = copy %8
  %4 = copy %5
  %5 = copy %4
  br %loop
ret:
  ret %4
```

- ▶ Value of  $\phi$  (%5) is wrong!

## Swap Problem / $\phi$ Dependencies

- ▶ Problem:  $\phi$ -nodes can depend on each other
- ▶ Can be chains (ordering matters) or cycles (need to be broken)
- ▶ Note: only  $\phi$ -nodes defined in same block are relevant/problematic



$$\begin{aligned}\phi_1 &= \phi(\phi_2, \dots) \\ \phi_2 &= \phi(\phi_3, \dots) \\ \phi_3 &= \phi(v, \dots)\end{aligned}$$

$$\begin{aligned}\phi_1 &= \phi(\phi_2, \dots) \\ \phi_2 &= \phi(\phi_3, \dots) \\ \phi_3 &= \phi(\phi_4, \dots) \\ \phi_4 &= \phi(\phi_1, \dots)\end{aligned}$$

$$\begin{aligned}\phi_1 &= \phi(\phi_2, \dots) \\ \phi_2 &= \phi(\phi_3, \dots) \\ \phi_3 &= \phi(\phi_1, \dots) \\ \phi_4 &= \phi(\phi_1, \dots)\end{aligned}$$

# Handling PHI Cycles

1. Compute number of other  $\phi$ -nodes reading other  $\phi$  on same edge
2. For each  $\phi$  with 0 readers: handle node/chain
  - ▶ No readers  $\rightsquigarrow$  start of chain
  - ▶ Handling node may unblock next element in chain
3. For all remaining  $\phi$ -nodes: must be cycles, reader count always 1
  - ▶ For trivial cycles (single node), do nothing
  - ▶ Choose arbitrary node, load to temporary register, unblock value
  - ▶ Handle just-created chain
  - ▶ Write temporary register to target

$\rightsquigarrow$  Resolving  $\phi$  cycles requires an extra register (or stack slot)

## SSA Destruction: Example 2 – Fixed

```
odd(%0)
e:
  br %loop
loop:
  %3 = phi [ %0, %e ], [ %8, %body ]
  %4 = phi [ 1, %e ], [ %5, %body ]
  %5 = phi [ 0, %e ], [ %4, %body ]
  %6 = cmp ne %3, 0
  br %6, %body, %ret
body:
  %8 = sub %3, 1
  br %loop
ret:
  ret %4
```

```
odd(%0)
e:
  %3 = copy %0
  %4 = copy 1
  %5 = copy 0
  br %loop
loop:
  %6 = cmp ne %3, 0
  br %6, %body, %ret
body:
  %8 = sub %3, 1
  %3 = copy %8
  %4t = copy %4
  %4 = copy %5
  %5 = copy %4t
  br %loop
ret:
  ret %4
```

- ▶  $\phi$ -cycle on edge body  $\rightarrow$  loop broken with temporary register

# SSA Destruction: Exercise

```
fn(%0, %1) {  
b1:  
  %2 = add %0, %1  
  br %b2  
b2:  
  %3 = phi [%1, %b1], [%4, %b3]  
  %4 = phi [%0, %b1], [%3, %b3]  
  %5 = phi [%2, %b1], [%3, %b3]  
  %6 = phi [0, %b1], [%8, %b3]  
  %7 = icmp lt %3, %6  
  br %7, %b3, %b4  
b3:  
  %8 = add %6, 1  
  %9 = icmp gt %8, %1  
  br %9, %b4, %b2  
b4:  
  %10 = phi [%4, %b2], [%3, %b3]  
  %11 = phi [%5, %b2], [%8, %b3]  
  %12 = add %10, %11  
  ret %12  
}
```

1. Dependencies between  $\phi$ -nodes?
2. Critical Edges? (Draw CFG)
3. Destruct SSA into form with unlimited registers.
  - 3.1 ... by breaking critical edges
  - 3.2 ... by copying used values

# SSA Destruction: Alternative Approach

- ▶ Other approach: generate a lot of copies, clean up later

```
odd(%0)
e:
  br %loop
loop:
  %3 = phi [ %0, %e ], [ %8, %body ]
  %4 = phi [ 1, %e ], [ %5, %body ]
  %5 = phi [ 0, %e ], [ %4, %body ]
  %6 = cmp ne %3, 0
  br %6, %body, %ret
body:
  %8 = sub %3, 1
  br %loop
ret:
  ret %4
```

```
odd(%0)
e:
  %3in = copy %0
  %4in = copy 1
  %5in = copy 0
  br %loop
loop:
  %3 = copy %3in
  %4 = copy %4in
  %5 = copy %5in
  %6 = cmp ne %3, 0
  br %6, %body, %ret
body:
  %8 = sub %3, 1
  %3in = copy %8
  %4in = copy %5
  %5in = copy %4
  br %loop
ret:
  ret %4
```

## Register Allocation: Simple Approaches

- ▶ Simplest thing that could possibly work:  
allocate a one stack slot for every (SSA) variable/argument
  - ▶ Reload all operands immediately before the instruction
  - ▶ Store to stack slot after computation
- + Simple, always works, debugging easy
- Extremely slow: values are stored and immediately reloaded
- Extremely inefficient memory usage: huge stack frames

## Register Allocation: Simple Approaches

- ▶ Next simplest thing that could possibly work:  
avoid reload if value is still in a register
  - ▶ When assigning target register: choose any register
  - ▶ Across basic blocks, spill everything
- + Still simple, always works, debugging easy
- Very slow: unnecessary stores, loop variants in memory
- Extremely inefficient memory usage: huge stack frames

## Register Allocation: Problems of Simple Approaches

- ▶ Many avoidable spills
  - ▶ Spill of last use of a value can/should be omitted
  - ▶ Stack slots for unused variables should be reused (why?)
- ▶ 2-address instructions (destructive source) require value copy
- ▶ Bad eviction decisions: value might be used immediately afterwards
- ▶ Loop-variants should be kept in registers if possible
  - ~~> Need information about what variables need to be preserved

## Liveness: Definitions

- ▶ *Live*: value might be used by later operation
  - ▶ After last (possible) use in program flow, the value becomes *dead*
- ▶ *Live ranges*: set of ranges in program where value is live
  - ▶ Not necessarily contiguous, e.g. in case of branches
- ▶ *Live-in/Live-out*: values live at begin/end of basic block/instruction
  - ▶ For  $\phi$  nodes:  $\phi$  is live-in in block, operands are live-out in predecessors  
(Note: different literature uses different definitions)

# Liveness: Example

|       |       |                                       |                           |
|-------|-------|---------------------------------------|---------------------------|
| $a_1$ |       | $a_1 \leftarrow \dots$                | liveIn=∅                  |
|       | $b_1$ | $b_1 \leftarrow 0$                    | liveOut={ $a_1, b_1, c$ } |
|       |       | $c \leftarrow \dots$                  |                           |
|       |       | br loop                               |                           |
| $a_2$ |       |                                       |                           |
|       | $b_2$ | loop: $a_2 \leftarrow \phi(a_1, a_3)$ | liveIn={ $a_2, b_2, c$ }  |
|       |       | $b_2 \leftarrow \phi(b_1, b_3)$       | liveOut={ $a_2, b_2, c$ } |
|       |       | br $b_2 < c$ , body, ret              |                           |
| $a_3$ |       |                                       |                           |
|       | $b_3$ | body: $a_3 \leftarrow a_2 + b_2$      | liveIn={ $a_2, b_2, c$ }  |
|       |       | $b_3 \leftarrow b_2 + 1$              | liveOut={ $a_3, b_3, c$ } |
|       |       | br body                               |                           |
|       |       | ret: ret $a_2$                        | liveIn={ $a_2$ }          |
|       |       |                                       | liveOut=∅                 |

## Liveness: Definitions<sup>57</sup>

- ▶  $Defs(B)$ : values defined in  $B$  (no defs from  $\phi$ )
- ▶  $Uses(B)$ : values used in  $B$  (no uses in  $\phi$ )
- ▶  $UpwardExposed(B)$ : values used in  $B$  before a definition in  $B$ 
  - ▶ SSA: this is  $Uses(B) \setminus (Defs(B) \cup \PhiDefs(B))$
- ▶  $\PhiDefs(B)$ : values defined by  $\phi$  at entry of  $B$
- ▶  $\PhiUses(B)$ : values used in  $\phi$ s in successors of  $B$

$$LiveOut(B) = \PhiUses(B) \cup \bigcup_{S \in \text{succs}(B)} (LiveIn(S) \setminus \PhiDefs(S))$$

$$LiveIn(B) = \PhiDefs(B) \cup UpwardExposed(B) \cup (LiveOut(B) \setminus Defs(B))$$

<sup>57</sup> B Boissinot and F Rastello. "Liveness". In: *SSA-based Compiler Design*. Ed. by F Rastello and F Bouchez Tichadou. 2022, pp. 107–122. DOI: 10.1007/978-3-030-80515-9\_9.

# Liveness: Iterative Data Flow Algorithm<sup>58</sup>

- ▶ Precompute per-basic block sets
  - ▶ Requires backwards pass over instructions of basic block
- ▶ Initialize  $LiveIn$ / $LiveOut$  to empty sets
- ▶ Iteratively recompute  $LiveIn$ / $LiveOut$  until convergence
- ▶ Block order has strong impact on runtime; post-order preferable
- ▶ Exact live ranges can be tracked
  - ▶  $LiveOut \Rightarrow$  live range extends to end of block; otherwise ends at last use inside block

<sup>58</sup>GA Kildall. "A unified approach to global program optimization". In: *POPL*. 1973, pp. 194–206. 

# Liveness: Iterative Algorithm Example

|       |       |                                       |                             |
|-------|-------|---------------------------------------|-----------------------------|
| $a_1$ |       | $a_1 \leftarrow \dots$                | $ue = \emptyset$            |
| $a_2$ | $b_1$ | $b_1 \leftarrow 0$                    | $liveIn = \emptyset$        |
| $a_3$ | $b_2$ | $c \leftarrow \dots$                  | $liveOut = \{a_1, b_1, c\}$ |
|       |       | <b>br loop</b>                        |                             |
|       |       | loop: $a_2 \leftarrow \phi(a_1, a_3)$ | $ue = \{c\}$                |
|       |       | $b_2 \leftarrow \phi(b_1, b_3)$       | $liveIn = \{a_2, b_2, c\}$  |
|       |       | <b>br</b> $b_2 < c$ , body, ret       | $liveOut = \{a_2, b_2, c\}$ |
|       | $b_3$ | body: $a_3 \leftarrow a_2 + b_2$      | $ue = \{a_2, b_2\}$         |
|       |       | $b_3 \leftarrow b_2 + 1$              | $liveIn = \{a_2, b_2, c\}$  |
|       |       | <b>br</b> body                        | $liveOut = \{a_3, b_3, c\}$ |
|       |       | ret: ret $a_2$                        | $ue = \{a_2\}$              |
|       |       |                                       | $liveIn = \{a_2\}$          |
|       |       |                                       | $liveOut = \emptyset$       |

## Liveness: Exercise

Compute  $liveIn$ ,  $liveOut$  for all blocks and live ranges for values.

```
b1 : a = f(0)
      br b1
b2 : b = f(a)
      br b3
b3 : br b > 0, b2, b4
b4 : c = phi(b, e)
      d = f(c)
      br c > 0, b3, b5
b5 : e = f(d)
      br e > 0, b4, b6
b6 : ret e
```

# Liveness: Iterative Algorithm Analysis

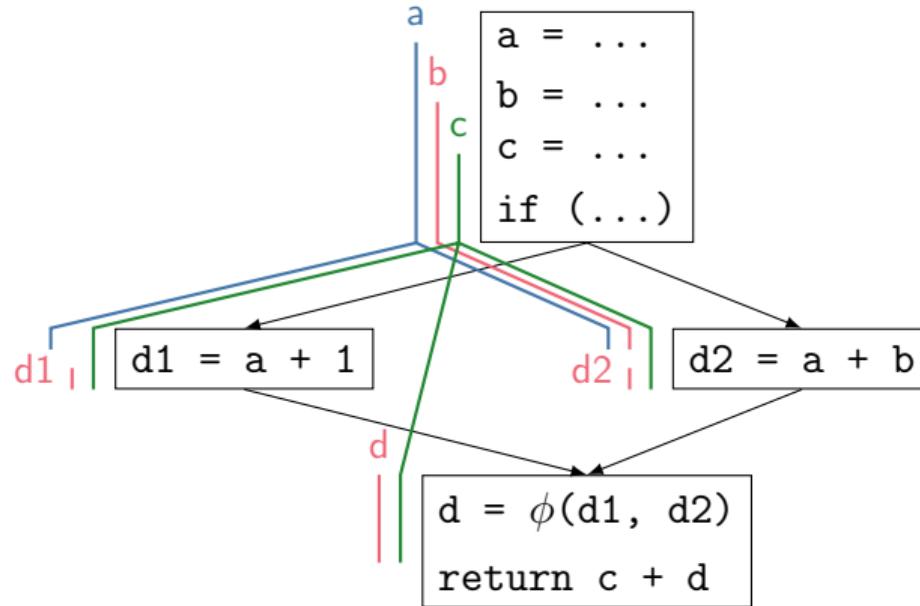
- ▶ Convergence: must converge, liveness sets grow monotonically
- ▶ Max. iterations:  $d(G) + 3^{59}$ 
  - ▶  $d(G)$  is depth = max. number of backedges on cycle-free path in  $G$
- ▶ Variation: worklist instead of round-robin<sup>60</sup>

<sup>59</sup> JB Kam and JD Ullman. "Global data flow analysis and iterative algorithms". In: *JACM* 23.1 (1976), pp. 158–171. 

<sup>60</sup> KD Cooper, TJ Harvey, and K Kennedy. "An empirical study of iterative data-flow analysis". In: *CIC*. 2006, pp. 266–276

# Liveness on SSA

- ▶ SSA values have one definition; uses must be dominated by that definition
- ⇒ Live ranges are sub-trees of dominator tree



# Liveness: Two-Pass Algorithm on SSA<sup>61</sup>

- ▶ Pass 1: Compute live-in/live-out in post-order, ignoring backedges
  - ▶ This is the first iteration of the iterative algorithm
- ▶ Pass 2: Extend live-in of loop headers to entire loops
  - ▶ Intuition: loop is SCC, so values live in header must be live in entire loop
  - ▶ Traverse loop forest in DFS
  - ▶ Add  $LiveIn(Hdr) \setminus \PhiDefs(Hdr)$  to  $LiveIn$  and  $LiveOut$  of loop blocks
- ▶ Complexity:  $\mathcal{O}(|E| \cdot \#vars + \#instrs)$
- ▶ Limitation: only works for reducible graphs

<sup>61</sup> B Boissinot et al. "A non-iterative data-flow algorithm for computing liveness sets in strict SSA programs". In: *ASPLAS*. 2011, pp. 137–154.

# Liveness: Two-Pass Algorithm on SSA for Irreducible CFG<sup>62</sup>

- ▶ Two-pass algorithm can be adapted for irreducible CFGs
- ▶ Adjust side entry to loop  $s \rightarrow t$  to outermost loop containing  $t$  excluding  $s$ 
  - ▶ Side entry: entry not going through header block (remember: header is ambiguous)
- ▶ Adjusted CFG doesn't need to be materialized, different CFG traversal
- ▶ No change in asymptotic complexity

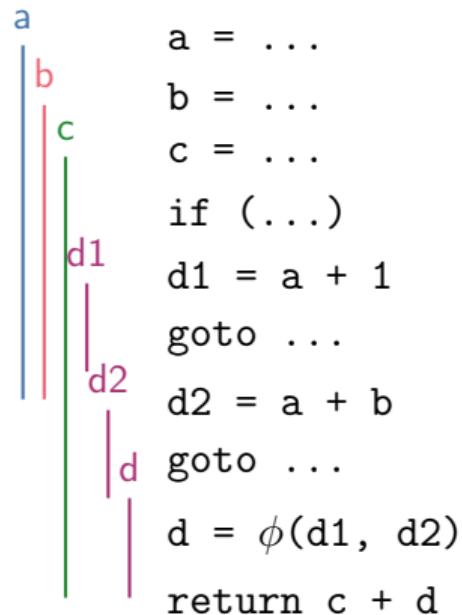
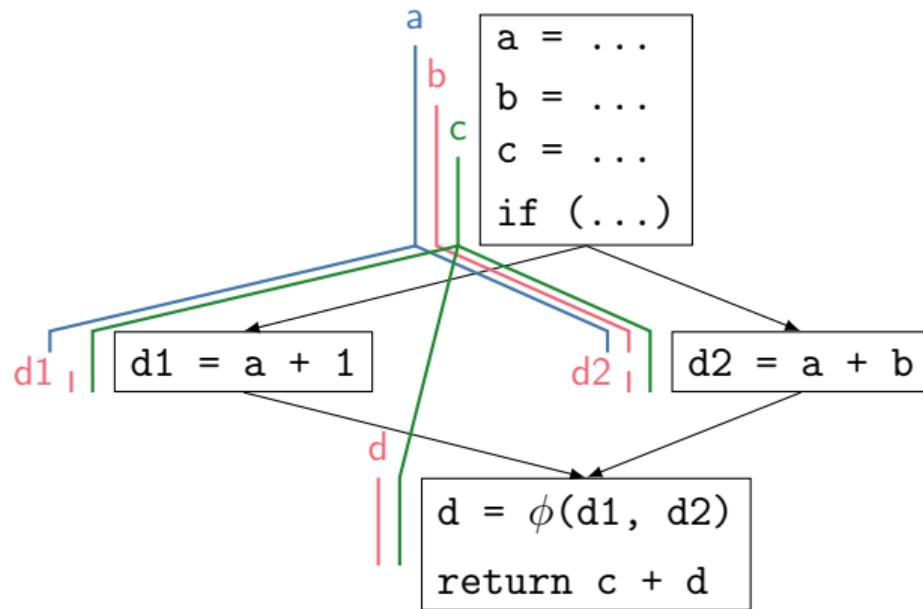
<sup>62</sup> B Boissinot and F Rastello. "Liveness". In: *SSA-based Compiler Design*. Ed. by F Rastello and F Bouchez Tichadou. 2022, pp. 107–122. DOI: 10.1007/978-3-030-80515-9\_9.

# Liveness: Over-Approximation

- ▶ Precise liveness analysis is somewhat expensive for large functions
- ▶ Also requires super-linear amount of memory to store
- ▶ For JIT compilation, not always feasible
- ~~> Over-approximation of liveness information
- ▶ Typical approach: single live interval per value<sup>63</sup>
  - ▶ Define fixed block order (typically RPO or similar)
  - ▶ Store begin and end location of live interval
  - ▶ Value might be marked as live in between even if it actually isn't

<sup>63</sup> M Poletto and V Sarkar. "Linear scan register allocation". In: *TOPLAS* 21.5 (1999), pp. 895–913. 

# Liveness: Live Ranges vs. Single Live Interval



- ▶ Single live interval can be substantially worse

# Live Interval Computation

- ▶ Can be done in single pass over program<sup>64</sup>
- ▶ Also can omit computation of live-in/live-out sets<sup>65</sup>
- ▶ Conceptually no large difference to two-pass algorithm  
except: growing live ranges simply adjusts end of interval

<sup>64</sup> C Wimmer and M Franz. "Linear scan register allocation on SSA form". In: CGO. 2010, pp. 170–179. 

<sup>65</sup> A Kohn, V Leis, and T Neumann. "Adaptive execution of compiled queries". In: ICDE. 2018, pp. 197–208. 

# Liveness Through Path Exploration<sup>66</sup>

- ▶ Alternative approach: trace uses back to their definition
- ▶ For every use:
  - ▶ If defined in the current block, stop (defined)
  - ▶ If in live-in of current block, stop (already propagated)
  - ▶ Add value to live-in
  - ▶ If value is a  $\phi$ -node, stop
  - ▶ Add value to live-out of all predecessors
  - ▶ Recursively continue in all predecessors
- ▶ Complexity:  $\mathcal{O}(|E| \cdot \#vars + \#instrs)$
- ▶ Used in LLVM

<sup>66</sup> F Rastello. "On Sparse Intermediate Representations: Some Structural Properties and Applications to Just-In-Time Compilation". Habilitation thesis. Inria Grenoble Rhône-Alpes, 2012. 

## SSA Destruction and Liveness Analysis – Summary

- ▶ SSA form must be destructed before machine code generation
- ▶  $\phi$ -nodes executed concurrently on edges
  - ▶ Critical edges need special handling: splitting or value copying
  - ▶  $\phi$ -nodes can have dependencies on each other
- ▶ Good register allocation needs liveness analysis
- ▶ Iterative data flow algorithm can need many iterations
- ▶ SSA form permits two-pass liveness analysis
- ▶ For faster analysis, liveness can be over-approximated

## SSA Destruction and Liveness Analysis – Questions

- ▶ What are the two main problems when destructing  $\phi$ -nodes?
- ▶ Why are critical edges problematic and how to deal with them?
- ▶ Which dependencies between  $\phi$ -nodes can occur? How to handle?
- ▶ Why is liveness information critical for reasonable code quality?
- ▶ When is a single-use SSA value live after its use?
- ▶ Why does SSA form make liveness analysis easier?
- ▶ How to compute the live ranges of values on SSA form?
- ▶ What are two benefits of storing just a single live interval?

# Code Generation for Data Processing

## Lecture 10: Register Allocation

Alexis Engelke

Chair of Data Science and Engineering (I25)  
School of Computation, Information, and Technology  
Technical University of Munich

Winter 2025/26

# Register Allocation

- ▶ Question: are there enough registers for all values?
  - ▶ *Register pressure* = number of values live at some point
  - ▶ Register pressure  $> \#$  registers  $\Rightarrow$  move some values to stack (spilling)
- ▶ Question: when spilling, which values and where to store/reload?
  - ▶ Spilling is expensive, so avoid spilling frequently used values
- ▶ Question: for unspilled values, which register to assign?
  - ▶ Also: respect register constraints, etc.

# Register Allocation Strategies

## Scan-based

- ▶ Iterate over the program
- ▶ Decide locally what to do
- ▶ Greedily assign registers
  
- + Fast, good for straight code
- Code quality often bad
- ▶ Used for -O0 and JIT comp.
  
- ▶ Traditionally done on non-SSA form

## Graph-based

- ▶ Compute *interference graph*
  - ▶ Nodes are values
  - ▶ Edge  $\Rightarrow$  live ranges overlap
- ▶ Holistic approach
  
- + Often generate good code
- Expensive, superlinear run-time
- ▶ Used for optimized code

# Linear Scan Register Allocation<sup>67</sup>

Read sections 3 and 4 of the paper.

1. How does the algorithm utilize liveness information?
2. How does the selected order of blocks impact the algorithm?
3. How are values selected for spilling?
4. When are values reloaded from the stack?
5. Which issues of register allocation are not discussed?

<sup>67</sup> M Poletto and V Sarkar. "Linear scan register allocation". In: *TOPLAS* 21.5 (1999), pp. 895–913. 

# Linear Scan Register Allocation

- + low compile-time, simple
- very suboptimal code, live intervals grossly over-approximated

- ▶ What's missing?
  - ▶ Registers to load spilled values
  - ▶ Shuffling of values between blocks
  - ▶ Register constraints (e.g., for instructions or function calls)
- ▶ Other disadvantage: once a value is spilled, it is spilled everywhere
  - ▶ Some other approaches based on lifetime splitting<sup>68</sup>
- ▶ Function calls: clobber lots of registers

<sup>68</sup> O Traub, G Holloway, and MD Smith. "Quality and speed in linear-scan register allocation". In: *SIGPLAN* 33.5 (1998), pp. 142–151. 

# Scan-based Register Allocation<sup>71</sup>

## Iterate over basic blocks<sup>69</sup>

- ▶ Start with register assignment from predecessor
  - ▶ Multiple predecessors: choose assignment from any one
- ▶ Iterate over instructions top-down
  - ▶ Ensure all instruction operands are in registers
    - ▶ When out of registers: move any value to stack
  - ▶ For operands in their last use: mark register as free
  - ▶ Assign instruction result to new free register
- ▶ Shuffle values back into registers where successor expects them<sup>70</sup>

<sup>69</sup>Typically: reverse post-order, so most predecessors are seen before successors, except for loops.

<sup>70</sup>Without critical edges, only relevant for blocks with one successor — others are visited afterwards by RPO definition.

<sup>71</sup>Mostly following Go: <https://github.com/golang/go/blob/5f7abe/src/cmd/compile/internal/ssa/regalloc.go>

# Scan-based Register Allocation – Spilling

What to spill?

- ▶ Spill value with furthest use in future<sup>72</sup>
  - ▶ Frees register for longest time
  - ▶ Requires information on next use to be stored during analysis
  - ▶ But: avoid spilling values computed inside loops (esp. loop-carried dependencies), reloads are fine<sup>73</sup>
  - ▶ Downside: superlinear run-time

Where to store?

- ▶ Stack
- ▶ Spilling to FP/vector registers... occasionally proposed, not used in practice

<sup>72</sup>C Wimmer and H Mössenböck. "Optimized interval splitting in a linear scan register allocator". In: VEE. 2005, pp. 132–141.

<sup>73</sup>Intel Optimization Reference Manual (Aug. 2023), Assembly/Compiler Coding Rules 38 and 45

# Scan-based Register Allocation – Spilling

Where to insert store?

- ▶ Option 1: spill exactly where required
  - ▶ Downside: multiple spills of same value, many reloads
- ▶ Option 2: spill once, immediately after computation
  - ▶ Later “spills” to the stack are less costly
  - ▶ May lead to spills on code paths that don’t need it
- ▶ Option 3: compute best place using dominator tree
  - ▶ Spill store must dominate all subsequent loads

## Scan-based Register Allocation – Register Assignment

- ▶ Merge blocks: choose predecessor with most values in registers
  - ▶ High likelihood of reducing the number of stores
  - ▶ Re-loads are pushed into predecessors
- ▶ Propagate register constraints bottom-up as hints first
  - ▶ E.g.: call parameters, instruction constraints, assignment for merge block
  - ▶ Reduces number of moves

## Graph Coloring: Overview

- ▶ Analyze values that are live at the same time
- ▶ Construct *interference graph*
  - ▶ Nodes: values; edge  $(a, b) \Rightarrow a$  and  $b$  have overlapping live ranges
- ▶ Idea: Find  $k$ -coloring of the graph
  - ▶ Each color corresponds to one register
- ▶  $\mathcal{NP}$ -complete problem for  $k > 2$

# Chaitin's Algorithm<sup>74</sup>

- ▶ Find node with  $< k$  edges
  - ▶ Such nodes can always be colored
  - ▶ Store node onto a stack
  - ▶ Remove node and its edges from interference graph
- ▶ If nodes remain: must have  $\geq k$  edges
  - ▶ Spill value to stack  $\rightsquigarrow$  new loads before uses
  - ▶ Update interference graph, restart
- ▶ Pop nodes from stack, assign color not used by neighbor

<sup>74</sup>GJ Chaitin. "Register allocation & spilling via graph coloring". In: *SIGPLAN 17.6* (1982), pp. 98–101. 

## Chaitin's Algorithm: Coalescing

- ▶ Avoid reg-reg moves by assigning them the same color
  - ▶ E.g., moves resulting from register constraints or two-address instructions
- ▶ Idea: *coalescing* of graph nodes
  - ▶ Merge move-related nodes that don't interfere
- ▶ Redo liveness analysis and rebuild interference graph
  - ▶ Might result in fewer interferences, more coalescing

+

- Avoids moves
- Can make colorable graph uncolorable  $\rightsquigarrow$  more spilling

## Chaitin's Algorithm: Register Constraints

- ▶ Add pre-colored nodes in interference graph: one per register
  - ▶ These nodes interfere with each other
- ▶ Nodes that require a specific register interfere with all other pre-colored nodes
- ▶ Typically insert moves into constrained reg right before use

## Graph Coloring: More Approaches

- ▶ Chaitin's algorithm can fail to color colorable graphs
- ▶ Several other approaches

- ▶ Interference graph on SSA is chordal
- ▶ Permits polynomial time coloring of graph
- ▶ SSA destruction done after assigning registers
- ▶ Spilling/coalescing must be done separately before coloring
- ▶ Optimal spilling remains  $\mathcal{NP}$ -complete
- ▶ After spill code insertion, SSA form needs to be reconstructed
- ▶ Pre-colored nodes make problem  $\mathcal{NP}$ -complete again
- ▶ Approaches: split into sub problems, repairing

# Stack Frame Allocation

- ▶ Optionally setup frame pointer
  - ▶ Required for variably-sized stack frame  
Otherwise: cannot access spilled variables or stack parameters
- ▶ Optionally re-align stack pointer
- ▶ Save callee-saved registers, maybe also link register
- ▶ Optionally add code for stack canary
- ▶ Compute stack frame size and adjust stack pointer
  - ▶ Mainly size of `allocas`, but needs to respect alignment
  - ▶ Ensure sufficient space for parameters passed on the stack
  - ▶ Ensure stack pointer is sufficiently aligned
- ▶ Stack pointer adjustment *may* be omitted for leaf functions
  - ▶ Some ABIs guarantee a *red zone*

# Block Ordering

- ▶ Order blocks to make use of fall-through in machine code
- ▶ Avoid sequences of `b.cond; b`
  - ▶ Sometimes cannot be avoided: conditional branches often have shorter range
- ▶ Block ordering has implications for branch prediction
  - ▶ Forward branches default to not-taken, backward taken
  - ▶ Unlikely blocks placed “out of the way” of the main execution path
  - ▶ Indirect branches are predicted as fall-through

## Register Allocation – Summary

- ▶ Scan-based approaches are fast, but lead to suboptimal code
- ▶ Graph coloring yields better results, but is much slower
- ▶ Coalescing important to reduce moves, esp. after SSA destruction
- ▶ SSA-based register allocation impractical
- ▶ Register allocation/spilling heavily relies on heuristics in practice

## Register Allocation – Questions

- ▶ Why is register allocation a difficult problem?
- ▶ What are practical constraints for register allocation?
- ▶ What is the idea of linear scan and what are its practical problems?
- ▶ How to construct an interference graph?

# Code Generation for Data Processing

## Lecture 11: Object Files, Linker, and Loader

Alexis Engelke

Chair of Data Science and Engineering (I25)  
School of Computation, Information, and Technology  
Technical University of Munich

Winter 2025/26

## Overview: Post-compilation

- ▶ Compiler emits object file
  - ▶ Somehow? Some format?
- ▶ Linker merges object files and determines required shared libraries
  - ▶ Somehow resolves missing symbols?
- ▶ Linker creates executable file
  - ▶ Somehow? Some format the OS understands?
- ▶ Kernel loads executable file into memory
- ▶ Someone loads shared libraries

# Code Model and Position Independent Code

- ▶ Code Model = address constraints
- ▶ Allows for better code
  - ▶ Long addrs/offsets = more instrs.
- ▶ Exact constraints arch/ABI-specific
- ▶ x86-64 SysV ABI:
  - ▶ Small: code and data max. 2 GiB
  - ▶ Medium: code max. 2 GiB
  - ▶ Large: no restrictions
- ▶ non-PIC: absolute addresses fixed at link-time
  - ▶ Addrs can be encoded directly
  - ▶ Sometimes slightly faster
  - ▶ Not possible for shared libs
- ▶ PIC: address random at load time
  - ▶ Offsets need be PC-relative
  - ▶ Addresses need fixup at load time (e.g., in jump tables)

---

Compiler needs to know code model

## Section 53

### Object Files

# Executable and Linkable Format (ELF)<sup>75</sup>

- ▶ Widely used format for code
- ▶ Supported file types:
  - ▶ REL: relocatable/object file
  - ▶ EXEC: executable (non-PIE)
  - ▶ DYN: shared library/PIE
  - ▶ CORE: coredump
- ▶ ELF header: general information
- ▶ Program headers: used for execution
  - ▶ Only for non-object files
- ▶ Section headers: used for linking
  - ▶ Typically always present, for non-object files just informational and not required

|  |
|--|
| ELF Header                             |
| Program Headers<br>(not for REL)       |
| .text                                  |
| .rodata                                |
| .data                                  |
| ...                                    |
| e.g., symtab, debug                    |
| Section Headers<br>(primarily for REL) |

<sup>75</sup>gABI specification: <https://gabi.xinuos.com/>

# ELF Header

```
// from glibc's elf.h
typedef struct {
    unsigned char e_ident[EI_NIDENT]; /* Magic number and other info */
    Elf64_Half e_type; /* Object file type */
    Elf64_Half e_machine; /* Architecture */
    Elf64_Word e_version; /* Object file version */
    Elf64_Addr e_entry; /* Entry point virtual address */
    Elf64_Off e_phoff; /* Program header table file offset */
    Elf64_Off e_shoff; /* Section header table file offset */
    Elf64_Word e_flags; /* Processor-specific flags */
    Elf64_Half e_ehsize; /* ELF header size in bytes */
    Elf64_Half e_phentsize; /* Program header table entry size */
    Elf64_Half e_phnum; /* Program header table entry count */
    Elf64_Half e_shentsize; /* Section header table entry size */
    Elf64_Half e_shnum; /* Section header table entry count */
    Elf64_Half e_shstrndx; /* Section header string table index */
} Elf64_Ehdr;
```

## ELF Sections

- ▶ Structures content of object files for linker
  - ▶ Linker later merges content sections of same “type”
- ▶ Some sections have “meta” information (e.g., symbols)
- ▶ .text – program text/code, executable
- ▶ .rodata – read-only data
- ▶ .data – initialized data, writable
- ▶ .bss – zero-initialized data, no storage, writable
  - ▶ Name history: block started by symbol
- ▶ .strtab – string table for symbol names
- ▶ .syms – symbol table, references string table for names
- ▶ .shstrtab – string table for section header names

# ELF String Table

- ▶ Sequence of NUL-terminated character sequences
- ▶ String identified by byte offset
- ▶ Must start with a NUL byte: string 0 always empty string
- ▶ Must end with a NUL byte: all strings are terminated

Example .strtab:

|          |   |   |   |   |   |   |   |           |   |   |   |          |
|----------|---|---|---|---|---|---|---|-----------|---|---|---|----------|
| \0       | v | a | r | n | a | m | e | \0        | f | o | o | \0       |
| String 0 |   |   |   |   |   |   |   | String 4  |   |   |   | String 9 |
| ""       |   |   |   |   |   |   |   | "varname" |   |   |   | "foo"    |
|          |   |   |   |   |   |   |   | "name"    |   |   |   |          |

# ELF Section Header

```
typedef struct {
    Elf64_Word sh_name; /* Section name (string tbl index) */
    Elf64_Word sh_type; /* Section type */
    // SHT_{NULL,PROGBITS,SYMTAB,STRTAB,RELA,HASH,NOBITS,...}
    Elf64_Xword sh_flags; /* Section flags */
    // SHF_{WRITE,ALLOC,EXECINSTR,MERGE,STRINGS,...}
    Elf64_Addr sh_addr; /* Section virtual addr at execution */
    Elf64_Off sh_offset; /* Section file offset */
    Elf64_Xword sh_size; /* Section size in bytes */
    Elf64_Word sh_link; /* Link to another section */
    Elf64_Word sh_info; /* Additional section information */
    Elf64_Xword sh_addralign; /* Section alignment */
    Elf64_Xword sh_entsize; /* Entry size if section holds table */
} Elf64_Shdr;
// first section is always undefined/SHT_NULL
```

## Example: Section Headers

```
void external(void);
static void bar(void) {}
void foo(void) { bar(); }
void func(void) {
    foo(); external(); }
```

### Section Headers:

| [Nr] | Name               | Type     | ES | Flg | Lk | Inf | Al |
|------|--------------------|----------|----|-----|----|-----|----|
| [ 0] | NULL               | PROGBITS | 00 |     | 0  | 0   | 0  |
| [ 1] | .text              | REL A    | 00 | AX  | 0  | 0   | 1  |
| [ 2] | .rela.text         | PROGBITS | 18 | I   | 10 | 1   | 8  |
| [ 3] | .data              | PROGBITS | 00 | WA  | 0  | 0   | 1  |
| [ 4] | .bss               | NOBITS   | 00 | WA  | 0  | 0   | 1  |
| [ 5] | .comment           | PROGBITS | 01 | MS  | 0  | 0   | 1  |
| [ 6] | .note.GNU-stack    | PROGBITS | 00 |     | 0  | 0   | 1  |
| [ 7] | .note.gnu.property | NOTE     | 00 | A   | 0  | 0   | 8  |
| [ 8] | .eh_frame          | PROGBITS | 00 | A   | 0  | 0   | 8  |
| [ 9] | .rela.eh_frame     | REL A    | 18 | I   | 10 | 8   | 8  |
| [10] | .symtab            | SYMTAB   | 18 |     | 11 | 4   | 8  |
| [11] | .strtab            | STRTAB   | 00 |     | 0  | 0   | 1  |
| [12] | .shstrtab          | STRTAB   | 00 |     | 0  | 0   | 1  |

# Symbol Table

- ▶ Describes symbolic reference to object/function
- ▶ Names in associated string table, referenced by byte offset
- ▶ Binding: local (static), weak, or global

```
typedef struct {
    Elf64_Word st_name; /* Symbol name (string tbl index) */
    unsigned char st_info; /* Symbol type and binding */
    unsigned char st_other; /* Symbol visibility */
    Elf64_Section st_shndx; /* Section index */
    Elf64_Addr st_value; /* Symbol value */
    Elf64_Xword st_size; /* Symbol size */
} Elf64_Sym;
```

# Example: Symbol Table

```
void external(void);  
static void bar(void) {}  
void foo(void) { bar(); }  
void func(void) {  
    foo(); external(); }
```

- ▶ Ndx=UND: undefined
  - ▶ value is zero
- ▶ Ndx=ABS: no section base
  - ▶ value is absolute
- ▶ Ndx=num: section idx.
  - ▶ value is offset into sec.
  - ▶ later refers to address

## Section Headers:

| [Nr]   | Name      | Type     | Size                | ES     | Flg | Lk | Inf | Al |
|--------|-----------|----------|---------------------|--------|-----|----|-----|----|
| [ 0]   |           | NULL     | 000000              | 00     |     | 0  | 0   | 0  |
| [ 1]   | .text     | PROGBITS | 00001a              | 00     | AX  | 0  | 0   | 1  |
| // ... |           |          |                     |        |     |    |     |    |
| [10]   | .symtab   | SYMTAB   | 0000a8              | 18     |     | 11 | 4   | 8  |
|        |           |          | sizeof(Elf64_Sym)   | --/    |     |    |     |    |
|        |           |          | link to strtab      | -----/ |     |    |     |    |
|        |           |          | first non-local sym | -----/ |     |    |     |    |
| [11]   | .strtab   | STRTAB   | 00001f              | 00     |     | 0  | 0   | 1  |
| [12]   | .shstrtab | STRTAB   | 00006c              | 00     |     | 0  | 0   | 1  |

## Symbol table '.symtab' contains 7 entries:

| Num: | Val | Size | Type    | Bind   | Vis     | Ndx | Name     |
|------|-----|------|---------|--------|---------|-----|----------|
| 0:   | 000 | 0    | NOTYPE  | LOCAL  | DEFAULT | UND |          |
| 1:   | 000 | 0    | FILE    | LOCAL  | DEFAULT | ABS | <stdin>  |
| 2:   | 000 | 0    | SECTION | LOCAL  | DEFAULT | 1   | .text    |
| 3:   | 000 | 1    | FUNC    | LOCAL  | DEFAULT | 1   | bar      |
| 4:   | 001 | 6    | FUNC    | GLOBAL | DEFAULT | 1   | foo      |
| 5:   | 007 | 19   | FUNC    | GLOBAL | DEFAULT | 1   | func     |
| 6:   | 000 | 0    | NOTYPE  | GLOBAL | DEFAULT | UND | external |

## Example: Writing Code to .text

```
void external(void);  
static void bar(void) {}  
void foo(void) { bar(); }  
void func(void) {  
    foo(); external(); }
```

|                          |                    |  |
|--------------------------|--------------------|--|
| 0000000000000000 <bar>:  |                    |  |
| 0: c3                    | ret                |  |
| 0000000000000001 <foo>:  |                    |  |
| 1: e8 fa ff ff ff        | call 0 <bar>       |  |
| 6: c3                    | ret                |  |
| 0000000000000007 <func>: |                    |  |
| 7: 48 83 ec 08           | sub rsp,0x8        |  |
| b: e8 00 00 00 00        | call 10 <func+0x9> |  |
| c: R_X86_64_PLT32        | foo-0x4            |  |
| 10: e8 00 00 00 00       | call 15 <func+0xe> |  |
| 11: R_X86_64_PLT32       | external-0x4       |  |
| 15: 48 83 c4 08          | add rsp,0x8        |  |
| 19: c3                   | ret                |  |

- ▶ Symbol may be unknown
- ▶ Linker needs to resolve offset later
- ~~ Relocations

# Relocations

- ▶ Problem: symbol values unknown before linking
  - ▶ Address is always unknown
  - ▶ External symbols: unavailable; other section: even distance is unknown
- ▶ Idea: store *relocations* ⇒ linker patches code/data
- ▶ Relocation: quadruple of (offset in sec., type, symbol idx, addend)
- ▶ Contained in REL/RELA/RELR sections

## Static Relocation

ET\_REL

## Dynamic Relocation ET\_EXEC/ET\_DYN

- ▶ For static linker (ld)
- ▶ Either: resolve or emit dyn. reloc

- ▶ For dynamic linker/loader
- ▶ Shall be fast, outside code

# Relocation Types

- ▶ Types and meaning defined by psABI<sup>76</sup>

**P**: address of place being relocated; **S**: symbol address; **L**: PLT addr. for symbol; **Z**: sym. size;  
**A**: addend; **B**: dynamic base address of shared obj.; **G**: GOT offset; **GOT**: GOT address

| Name                   | Field | Calculation       | Name                | Field | Calculation       |
|------------------------|-------|-------------------|---------------------|-------|-------------------|
| R_X86_64_64            | 64    | $S + A$           | R_X86_64_32         | 32    | $S + A$ (zext)    |
| R_X86_64_PC32          | 32    | $S + A - P$       | R_X86_64_32S        | 32    | $S + A$ (sext)    |
| R_X86_64_GOT32         | 32    | $G + A$           | R_X86_64_GOTOFF64   | 64    | $S + A - GOT$     |
| R_X86_64_PLT32         | 32    | $L + A - P$       | R_X86_64_GOTPC32    | 32    | $GOT + A - P$     |
| R_X86_64_GLOB_DAT      | addr  | $S$               | R_X86_64_GOT64      | 64    | $G + A$           |
| R_X86_64_JUMP_SLOT     | addr  | $S$               | R_X86_64_GOTPCREL64 | 64    | $G + GOT + A - P$ |
| R_X86_64_RELATIVE      | addr  | $B + A$           | R_X86_64_GOTPC64    | 64    | $GOT + A - P$     |
| R_X86_64_GOTPCREL      | 32    | $G + GOT + A - P$ | R_X86_64_PLTOFF64   | 64    | $L - GOT + A$     |
| R_X86_64_GOTPCRELX     |       |                   | R_X86_64_SIZE32     | 32    | $Z + A$           |
| R_X86_64_REX_GOTPCRELX |       |                   | R_X86_64_SIZE64     | 64    | $Z + A$           |

<sup>76</sup>x86-64: HJ Lu et al. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*. 2022. 

# Relocation Section

Section Headers:

| [Nr] | Name       | Type               | Size       | ES   | Flg         | Lk     | Inf | Al |
|------|------------|--------------------|------------|------|-------------|--------|-----|----|
| [ 1] | .text      | PROGBITS           | 00001a     | 00   | AX          | 0      | 0   | 1  |
| [ 2] | .rela.text | RELA               | 000030     | 18   | I           | 10     | 1   | 8  |
|      |            | sizeof(Elf64_Rela) | --/        |      |             |        |     |    |
|      |            | I: info            | is section | link | -----/      |        |     |    |
|      |            |                    | link to    | syms | -----/      |        |     |    |
|      |            | target             | sec.       | for  | relocations | -----/ |     |    |
| [10] | .symtab    | SYMTAB             | 0000a8     | 18   |             | 11     | 4   | 8  |

Relocation section '.rela.text' at offset 0x1e0 contains 2 entries:

| Offset            | Info             | Type           | Symbol's Name + Addend |
|-------------------|------------------|----------------|------------------------|
| 0000000000000000c | 0000000400000002 | R_X86_64_PC32  | foo - 4                |
| 00000000000000011 | 0000000600000004 | R_X86_64_PLT32 | external - 4           |

# Relocations on RISC Architectures

- ▶ RISC architectures typically have *more* relocation types
  - ▶ Example: AArch64<sup>77</sup> has >50 relocations
- ▶ Building a 64-bit address requires several instructions  
(AArch64: one for bits 0–15, 16–31, ...)
  - ▶ Each instruction needs a different relocation to patch in the bits!

```
movz x0, #:abs_g0_nc:globalVariable
movk x0, #:abs_g1_nc:globalVariable
movk x0, #:abs_g2_nc:globalVariable
movk x0, #:abs_g3:globalVariable
```

- ▶ Often: page-granular address with added offset for low bits
  - ▶ adrp for  $\pm 4$  GiB range, add or load offset for low bits
  - ▶ Scaled load offsets require different relocations for each scale

## Branch Relocations

- ▶ Branches (often) have limited range; compiler must assume max. distance
- ▶ x86-64:  $\pm 2$  GiB range, if larger use `mov` and indirect jump
- ▶ AArch64:  $\pm 128$  MiB range  $\rightsquigarrow$  executable sections must be  $< 127$  MiB  
linker will insert veneer between different `.text` sections
  - ▶ Veneer allowed to clobber inter-procedural scratch registers `x16/x17`
- ▶ *badly designed ISA*:  $\pm 1$  MiB range  $\rightsquigarrow$  needs ind. jump *often*

# Branch Relocations on RISC-V

1. Compile the code<sup>78</sup> with:

```
clang --target=riscv64 -c -o rv.o rv.c -falign-functions=16

int f() { return 0; }
int g() { return f(); }
int h() { return g(); }
```

2. Look at the relocations and disassembly: `llvm-objdump -dr rv.o`  
How are the function calls lowered? What types of relocations are there?  
Look up the relocations types in the psABI<sup>79</sup> – what do they mean?
3. Link the file<sup>80</sup>: `ld.1ld -shared -o rv.so rv.o` and disassemble `rv.so`.  
What is different now?

<sup>78</sup><https://godbolt.org/z/7d3PWzY4f>

<sup>79</sup><https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/b17fc7b9b/riscv-elf.adoc#relocations>

<sup>80</sup>Compiler Explorer: Output → uncheck *Compile to object*, check *Link to binary*.

## Section 54

### Executable Files

# Linker<sup>81</sup>

- ▶ Goal: combine multiple input files (.o/.so/.a) into executable or shared lib.
- 1. Find and load all input files
- 2. Scan input, store symbols, resolve symbols on-the-fly
- 3. Create synthetic section (GOT, PLT, relocations for output file)
- 4. Process relocations: create PLT/GOT entry and dynamic reloc.
- 5. Optimize and deduplicate sections
- 6. Write section to output file
  - ▶ Apply relocations which are now known; compress sections; etc.

<sup>81</sup>Interesting blog on linkers, etc.: [F Song](#). *Personal Blog*.  (visited on 01/07/2026).

# ELF Executable File

- ▶ Entry in ELF header: entry address of the program
  - ▶ Kernel will configure program to start execution there
- ▶ Program execution doesn't start at `main`
  - ▶ Need to run global constructors: e.g., to initialize variables in C++
  - ▶ Init I/O, heap, etc.
- ▶ Program execution (by default) start at `_start`
  - ▶ Typically provided by C runtime, calls `__libc_start_main`
- ▶ Compiler driver (e.g. `gcc`, `clang`) can output low-level tool invocations:  
`clang -### ...`

## ELF Executable File: Program Headers

- ▶ Program headers: instructions for loading the program
- ▶ PT\_PHDR: describes program headers
- ▶ PT\_LOAD: loadable segment
  - ▶ Specifies virtual address, file offset, file size/memory size, permission
  - ▶  $vaddr \& (pgsize-1) == offset \& (pgsize-1)$  – kernel will just `mmap` the file
  - ▶ memory size > file size  $\Rightarrow$  filled up with zeros (for .bss)
- ▶ PT\_INTERP/PT\_DYNAMIC: when PIE or with shared libraries
- ▶ PT\_GNU\_STACK: permissions indicate whether stack is non-executable

# Example: Program Headers

Program Headers:

| Type      | Offset                         | VirtAddr    | FileSiz   | MemSiz    | Flg | Align  |
|-----------|--------------------------------|-------------|-----------|-----------|-----|--------|
| LOAD      | 0x0000000                      | 0x004000000 | 0x0a0d5e  | 0x0a0d5e  | R E | 0x1000 |
| LOAD      | 0x0a17d8                       | 0x004a27d8  | 0x005ab8  | 0x00b2e8  | RW  | 0x1000 |
|           | offset in file                 | -/          |           |           |     |        |
|           | virtual address                | -----/      |           |           |     |        |
|           | bytes provided in file         | -----/      |           |           |     |        |
|           | segment size in mem            | -----/      |           |           |     |        |
|           | (memsz > filesz = zero-filled) |             |           |           |     |        |
|           | mmap protection                | -----/      |           |           |     |        |
| // ...    |                                |             |           |           |     |        |
| GNU_STACK | 0x0000000                      | 0x000000000 | 0x0000000 | 0x0000000 | RW  | 0x10   |

- ▶ Note: the kernel always maps full pages from the file cache
- ▶ Note: first segment includes ELF header and program headers

# Loading a Binary to Memory

- ▶ Load ELF header and program header
- ▶ If ET\_DYN (↔ PIE), set random base added to all addresses
- ▶ Look if PT\_INTERP is present
  - ▶ If present, load interpreter using same algorithm (but no nested interpreters)
- ▶ Iterate over PT\_LOAD and mmap segments
  - ▶ May needs zeroing of last page and mapping extra zero pages
- ▶ Setup initial stack frame and auxiliary vector (e.g., with phdr address)
- ▶ Start execution at (the interpreter's) entry

---

This is the kernel's job

## Section 55

### Linker Optimizations

# Eliminating Duplicate Strings/Constants

- ▶ Sections in different object may contain same data, e.g. strings
  - ▶ Critical for debug info (file names, function names, etc.)
- ▶ Idea: linker finds and deduplicates strings and other constant data
- ▶ Precondition: relative order of entries irrelevant
- ▶ SHF\_MERGE – fixed-size entries, size stored in header
  - ▶ Collect all entries in hash map; afterwards emit all keys
- ▶ SHF\_MERGE|SHF\_STRINGS – NUL-terminated strings, entsize is char width
  - ▶ Precondition: strings must not contain NUL-byte
  - ▶ Tail merging: `foobar\0 + bar\0 ~> foobar\0`
  - ▶ Sort strings from tail (e.g., radix sort), deduplicate neighbors

# COMDAT Groups

```
//--- inline1.cpp
inline int x(int n) {
    return n ? x(n-1) + n : 1;
}
int f(int n) { return x(n); }

//--- inline2.cpp
inline int x(int n) {
    return n ? x(n-1) + n : 1;
}
int g(int n) { return x(n); }

int main() {}

// clang++ -c -o inline1.o inline1.cpp
// clang++ -c -o inline2.o inline2.cpp
// clang++ -o inline inline{1,2}.o
```

1. Inspect sections/symbols of the object files with `llvm-readelf -gCSs`.
  - ▶ What sections are there?
  - ▶ Which symbol bindings?
2. Likewise, inspect the executable file
  - ▶ How many instances of `x(int)` exist?

# Linker Garbage Collection

- ▶ Problem: objects may contain unused functions
  - ▶ Compiler can't know whether function is used
- ▶ Idea: put all function into separate sections, drop unused sections
- ▶ Sections are considered as inseparable units
- ▶ GC roots: exported symbols, init functions, ...
- ▶ Iteratively mark all referenced sections, drop unmarked sections
- ▶ Downside: may need longer relocations  $\rightsquigarrow$  possibly less efficient code
- ▶ GCC/Clang -ffunction-sections, ld --gc-sections

# Identical Code Folding

- ▶ Problem: objects may contain duplicate code
  - ▶ Same function compiled in many objs, e.g. template instantiation
- ▶ Idea: deduplicate read-only sections (same flags, contents, relocations(!))
- ▶ Hash all sections and their relocations, remove duplicates
- ▶ Repeat until convergence
  - ▶ Only after folding `foo1` and `foo2`, these become equivalent:

```
int funcA(void) { foo1(); } int funcB(void) { foo2(); }
```
- ▶ Caution: function pointers may be guaranteed to be different
- ▶ LLD has more aggressive deduplication

# Link-Time Optimization

- ▶ Problem: no optimizations across object files
  - ▶ Inlining, constant propagation+cloning, specialized call conv., ...
  - ▶ Optimization across language boundaries
- ▶ Idea 1: glue all source code together, compile with `-fwhole-program`
  - ▶ Downside: single core, problematic with same-name static functions
- ▶ Idea 2: Use static binary optimization during linking (severely limited)
- ▶ Idea 3: dump IR into object, glue IR together, optimize (`-flto`)
  - ▶ Done as very first step at link-time
- ▶ LTO is widely used and highly effective

# Link-Time Optimization in LLVM

- ▶ During object file compilation: just run simplification pipeline
- ▶ Fat LTO: all LLVM-IR modules from objects merged into single module
  - ▶ All existing optimizations can be applied, (conceptually) best performance
  - ▶ But: single-threaded compilation makes this not viable
- ▶ Thin LTO<sup>82</sup>: do cross-object analysis only on summary
  - ▶ Every object has a summary index: function sizes, linkage, call graph
  - ▶ Serial cross-module analysis: identify inlining candidate, other global opt.
  - ▶ Parallel optimization/compilation: apply optimizations found in analysis

<sup>82</sup>T. Johnson, M. Amini, and X.D. Li. "ThinLTO: scalable and incremental LTO". In: CGO. 2017, pp. 111–121

## Section 56

### Static Libraries

## Static Libraries

- ▶ Archive of relocatable object files
- ▶ Header often contains index mapping symbol to object file
- ▶ Linker takes only object files that are needed
- ▶ Code/data copied into final executable
  
- + Simple and fast, no ABI problems, no extra library needed at run-time
- Larger executable files, library changes need relinking

## Section 57

### Shared Libraries

## Shared Libraries

- ▶ Problem: code duplication, large executables, recompile needed for changes
- ▶ Idea: *share* code between different executables
- ▶ Executable references functions/objects in shared library
  - ▶ Shared libraries can refer to other shared libraries, too
  - ▶ Linker needs to retain dynamic relocations and symbols  
(dynamic symbol = externally visible symbol)
- ▶ Run-time loader links executable and libraries program start
  - ▶ Find and load libraries from different paths, resolve all relocations

# Shared Libraries: Changes in Compiler

None 😊  
(almost)

- ▶ When building a shared library, code must be position-independent

## Shared Libraries: Changes in Linker

- ▶ Relocations to symbols in shared libraries must be retained
  - ▶ Store dynamic relocations and symbols in separate sections ( `.dynsym`, `.rela.dyn` )
- ▶ Create table (GOT) for pointers to external function/objects
  - ▶ Allocate space where loader puts addresses, add relocations
- ▶ Create stub functions for external functions (PLT)
  - ▶ Compiler still creates near call, which gets redirected to stub
  - ▶ Stub jumps to address stored in table
- ▶ Emit `PT_DYNAMIC` segment with info for loader
  - ▶ Point loader to needed libs, relocations, `syms`, `strtab`, ...

# Global Offset Table (GOT) and Procedure Linkage Table (PLT)

- ▶ Global Offset Table: pointer table filled by loader
  - ▶ Linker emits dynamic relocations for GOT; loader fills addresses
  - ▶ Often subject to RELRO: after relocations are applied, GOT becomes read-only
- ▶ Procedure Linkage Table: stubs that perform jump using GOT

```
00401030 <func@plt>:  
401030: ff 25 8a 2f 00 00 jmp QWORD PTR [rip+0x2f8a] # GOT slot
```
- ▶ PLT can be disabled (-fno-plt): indirect jump is duplicated
  - ▶ Compiler emits indirect calls/jumps instead of near calls to PLT
  - ▶ Linker cannot convert into near jump if target is in same DSO

## PT\_DYNAMIC segment

- ▶ Loader needs to know needed libraries, flags, locations of relocations, etc.
  - ▶ Sections headers might be unavailable and more info is needed
- ▶ Info for loader stored in dynamic section

| Type       | Name/Value                  |
|------------|-----------------------------|
| (NEEDED)   | Shared library: [libm.so.6] |
| (NEEDED)   | Shared library: [libc.so.6] |
| (GNU_HASH) | 0x4003c0                    |
| (STRTAB)   | 0x4004b8                    |
| (SYMTAB)   | 0x4003e0                    |
| (STRSZ)    | 259 (bytes)                 |
| (SYMENT)   | 24 (bytes)                  |
| // ...     |                             |
| (NULL)     | 0x0                         |

# Symbol Lookup

- ▶ Symbol lookup using linear search + strcmp is slow
- ▶ Idea: linker creates hash table
  - ▶ Hash symbol names and store them in hash table
  - ▶ Dynamic symbols grouped by hash bucket
  - ▶ Additional bloom filter to avoid useless walks for absent symbols
- ▶ Lookup:
  - ▶ Compute hash of target symbol string
  - ▶ Check bloom filter, if absent: abort
  - ▶ Iterate through symbols in bucket, compare names (and version)
- ▶ Documentation unfortunately sparse<sup>83</sup>

<sup>83</sup> A Roenky. *ELF: better symbol lookup via DT\_GNU\_HASH.* (visited on 12/14/2022)

# Miscellaneous Things

- ▶ Purpose of all these dynamic entries
- ▶ Symbols: versioning and DSO visibility
- ▶ Copy relocations for more efficient globals in main executable
- ▶ Thread-local storage
- ▶ Constructors/destructors: called at load/unload of DSO
- ▶ Indirect functions (ifunc)
  - ▶ Function to dynamically determine actual address of symbol
  - ▶ Used e.g. for determining `memcpy` variant based on CPU features
- ▶ Dynamic loading of DSOs (`dlopen`)

# Object Files, Linker, and Loader – Summary

- ▶ Compiler needs to know code model to emit proper asm code/relocations
- ▶ ELF format used for relocatable files, executables and shared libraries
- ▶ ELF relocatables structured in sections and have static relocations
- ▶ ELF dynamic executables grouped in segments and have dynamic relocations
  - ▶ Need dynamic loader to resolve dynamic relocations and shared libraries
- ▶ Linker combines relocatable files into executables or shared libraries
- ▶ Optimizations can happen at link time: COMDAT group merging, string/constant merging, section garbage collection, identical code folding, link-time optimization, linker relaxation

## Object Files, Linker, and Loader – Questions

- ▶ Which ELF file types exist? What is different?
- ▶ What are typical sections found in an ELF relocatable file?
- ▶ What information is contained in a symbol table?
- ▶ What information is required for a relocation?
- ▶ What are typical differences between static and dynamic relocations?
- ▶ How can linker relaxation improve program performance?
- ▶ Which steps and possible optimization does a linker perform?
- ▶ How does the OS load a binary into memory?
- ▶ What is the difference between static and shared libraries?
- ▶ How are symbols from other shared libraries resolved?

# Code Generation for Data Processing

## Lecture 12: Unwinding and Debuginfo

Alexis Engelke

Chair of Data Science and Engineering (I25)  
School of Computation, Information, and Technology  
Technical University of Munich

Winter 2025/26

## Motivation: Meta-Information on Program

- ▶ Machine code suffices for execution → not true
- ▶ Needs program headers and entry point
- ▶ Linking with shared libraries needs dynamic symbols and interpreter
- ▶ Stack unwinding needs information about the stack
  - ▶ Size of each stack frame, destructors to be called, etc.
  - ▶ Vital for C++ exceptions, even for non-C++ code
- ▶ Stack traces require stack information to find return addresses
  - ▶ Use cases: core dumps, debuggers, profilers
- ▶ Debugging experience enhanced by variables, files, lines, statements, etc.

# Adding Meta-Information with GCC

`-g`  
`-fexceptions`  
`-fasynchronous-unwind-tables`

- ▶ `-g` supports different formats and levels (and GNU extensions)
- ▶ Exceptions must work without debuginfo
- ▶ Unwinding through code without exception-support must work

# Stack Unwinding

- ▶ Needed for exceptions (`_Unwind_RaiseException`) or forced unwinding
- ▶ Search phase: walk through the stack, check whether to stop at each frame
  - ▶ May depend on exception type, ask *personality function*
  - ▶ Personality function needs extra language-specific data
  - ▶ Stop once an exception handler is found
- ▶ Cleanup phase: walk again, do cleanup and stop at handler
  - ▶ Personality function indicates whether handler needs to be called
  - ▶ Can be for exception handler or for calling destructors
  - ▶ If yes: personality function sets up registers/sp/pc for landing pad
  - ▶ Non-matching handler or destructor-only: landing pad calls `_Unwind_Resume`

## Stack Unwinding: Requirements

- ▶ Given: current register values in unwind function
- ▶ Need: iterate through stack frames
  - ▶ Get address of function of the stack frame
  - ▶ Get pc and sp for *this function*
  - ▶ Find personality function and language-specific data
  - ▶ Maybe get some registers from the stack frame

## Stack Unwinding: `setjmp/longjmp`

- ▶ Simple idea – all functions that run code during unwinding do:
  - ▶ Register their handler at function entry
  - ▶ Deregister their handler at function exit
- ▶ Personality function sets `jmpbuf` to landing pad
- ▶ Unwinder does `longjmp`

- + Needs no extra information, small binary footprint
- High overhead in non-exceptional case

# Stack Unwinding: Frame Pointer

- ▶ Frame pointers allow for fast unwinding
- ▶ fp points to stored caller's fp
- ▶ Return address stored adjacent to frame pointer
- ⊕ Fast and simple, also without exception
- ⊖ Not all programs have frame pointers
  - ▶ Overhead of creating full stack frame
  - ▶ Causes loss of one register (esp. x86)
- ⊖ Not generally possible to restore callee-saved registers
- ▶ Not sufficient!

x86\_64:

```
push rbp
mov rbp, rsp
// ...
mov rsp, rbp
pop rbp
ret
```

aarch64:

```
stp x29, x30, [sp, -32]!
mov x29, sp
// ...
ldp x29, x30, [sp], 32
ret
```

## Stack Unwinding: Without Frame Pointer

- ▶ Definition: *canonical frame address (CFA)* is `sp` at the function call
- ▶ Given: `pc` and `sp` (bottom of stack frame/call frame)
  - ▶ In parent frames:  $retaddr - 1 \sim pc$  and  $CFA \sim sp$
- ▶ Need to map `pc` to stack frame layout
  - ▶ Stack frame varies throughout function, e.g. prologue, stack arguments
- ▶ Case 1: some register used as frame pointer – CFA constant offset to `fp`
  - ▶ E.g., for variable stack frame size, stack realignment on function entry
- ▶ Case 2: no frame pointer: CFA is constant offset to `sp`

↝ Unwinding *must* restore register values

- ▶ Frame pointer of caller, caller's landing pad can use callee-saved regs
- ▶ Need to know where return address is stored (to identify caller)

## Call Frame Information

- ▶ Table mapping each instr. to info about registers and CFA
- ▶ CFA: register with signed offset (or arbitrary expression)
- ▶ Register:
  - ▶ Undefined – unrecoverable (default for caller-saved reg)
  - ▶ Same – unmodified (default for callee-saved reg)
  - ▶ Offset( $N$ ) – stored at address CFA+ $N$
  - ▶ Register( $reg$ ) – stored in other register
  - ▶ or arbitrary expressions

# Call Frame Information – Example 1

- ▶ Table row: frame info starting from address
  - ▶ Note: instruction pointed to by pc has not yet executed
- ▶ Table columns: CFA, return address, registers

|      |              | CFA      | rip        | rbx        | rbp  | ... |
|------|--------------|----------|------------|------------|------|-----|
|      | foo:         |          |            |            |      |     |
| 0x0: | push rbx     | rsp+0x08 | [CFA-0x08] | same       | same |     |
| 0x1: | mov ebx, edi | rsp+0x10 | [CFA-0x08] | [CFA-0x10] | same |     |
| 0x3: | call bar     | rsp+0x10 | [CFA-0x08] | [CFA-0x10] | same |     |
| 0x8: | mov eax, ebx | rsp+0x10 | [CFA-0x08] | [CFA-0x10] | same |     |
| 0xa: | pop rbx      | rsp+0x10 | [CFA-0x08] | [CFA-0x10] | same |     |
| 0xb: | ret          | rsp+0x08 | [CFA-0x08] | same       | same |     |

## Call Frame Information – Example 2

|       |              | CFA      | rip        | rbx  | rbp        | ... |
|-------|--------------|----------|------------|------|------------|-----|
|       | foo:         |          |            |      |            |     |
| 0x0:  | push rbp     | rsp+0x08 | [CFA-0x08] | same | same       |     |
| 0x1:  | mov rbp, rsp | rsp+0x10 | [CFA-0x08] | same | [CFA-0x10] |     |
| 0x4:  | shl rdi, 4   | rbp+0x10 | [CFA-0x08] | same | [CFA-0x10] |     |
| 0x8:  | sub rsp, rdi | rbp+0x10 | [CFA-0x08] | same | [CFA-0x10] |     |
| 0xb:  | mov rdi, rsp | rbp+0x10 | [CFA-0x08] | same | [CFA-0x10] |     |
| 0xe:  | call bar     | rbp+0x10 | [CFA-0x08] | same | [CFA-0x10] |     |
| 0x13: | leave        | rbp+0x10 | [CFA-0x08] | same | [CFA-0x10] |     |
| 0x14: | ret          | rsp+0x08 | [CFA-0x08] | same | same       |     |

## Call Frame Information – Example 3

|       |               | CFA      | rip        | rbx  | rbp  | ... |
|-------|---------------|----------|------------|------|------|-----|
|       | foo:          |          |            |      |      |     |
| 0x0:  | sub rsp, 8    | rsp+0x08 | [CFA-0x08] | same | same |     |
| 0x4:  | test edi, edi | rsp+0x10 | [CFA-0x08] | same | same |     |
| 0x6:  | js 0x12       | rsp+0x10 | [CFA-0x08] | same | same |     |
| 0x8:  | call positive | rsp+0x10 | [CFA-0x08] | same | same |     |
| 0xd:  | add rsp, 8    | rsp+0x10 | [CFA-0x08] | same | same |     |
| 0x11: | ret           | rsp+0x08 | [CFA-0x08] | same | same |     |
| 0x12: | call negative | rsp+0x10 | [CFA-0x08] | same | same |     |
| 0x17: | add rsp, 8    | rsp+0x10 | [CFA-0x08] | same | same |     |
| 0x1a: | ret           | rsp+0x08 | [CFA-0x08] | same | same |     |

## Call Frame Information – Exercise

- ▶ Download `ex12.txt` from the course website
- ▶ Construct the CFI tables for both functions  
(you can omit lines that don't change)

## Call Frame Information: Encoding

- ▶ Expanded table can be huge
- ▶ Contents change rather seldomly
  - ▶ Mainly in prologue/epilogue, but mostly constant in-between
- ▶ Idea: encode table as bytecode
- ▶ Bytecode has instructions to create a new row
  - ▶ Advance machine code location
- ▶ Bytecode has instructions to define CFA value
- ▶ Bytecode has instructions to define register location
- ▶ Bytecode has instructions to remember and restore state

# Call Frame Information: Bytecode – Example 1

|                 | CFA    | rip     | rbx      |                            |
|-----------------|--------|---------|----------|----------------------------|
| foo:            |        |         |          |                            |
| 0: push rbx     | rsp+8  | [CFA-8] |          | DW_CFA_def_cfa: RSP +8     |
| 1: mov ebx, edi | rsp+16 | [CFA-8] | [CFA-16] | DW_CFA_offset: RIP -8      |
| 3: call bar     | rsp+16 | [CFA-8] | [CFA-16] | DW_CFA_advance_loc: 1      |
| 8: mov eax, ebx | rsp+16 | [CFA-8] | [CFA-16] | DW_CFA_def_cfa_offset: +16 |
| a: pop rbx      | rsp+16 | [CFA-8] | [CFA-16] | DW_CFA_offset: RBX -16     |
| b: ret          | rsp+8  | [CFA-8] | [CFA-16] | DW_CFA_advance_loc: 10     |
|                 |        |         |          | DW_CFA_def_cfa_offset: +8  |

## Call Frame Information: Bytecode – Example 2

|     |              | CFA    | rip     | rbp      |                              |
|-----|--------------|--------|---------|----------|------------------------------|
|     | foo:         |        |         |          |                              |
| 0:  | push rbp     | rsp+8  | [CFA-8] |          | DW_CFA_def_cfa: RSP +8       |
| 1:  | mov rbp, rsp | rsp+16 | [CFA-8] | [CFA-16] | DW_CFA_offset: RIP -8        |
| 4:  | shl rdi, 4   | rbp+16 | [CFA-8] | [CFA-16] | DW_CFA_advance_loc: 1        |
| 8:  | sub rsp, rdi | rbp+16 | [CFA-8] | [CFA-16] | DW_CFA_def_cfa_offset: +16   |
| b:  | mov rdi, rsp | rbp+16 | [CFA-8] | [CFA-16] | DW_CFA_offset: RBP -16       |
| e:  | call bar     | rbp+16 | [CFA-8] | [CFA-16] | DW_CFA_advance_loc: 3        |
| 13: | leave        | rbp+16 | [CFA-8] | [CFA-16] | DW_CFA_def_cfa_register: RBP |
| 14: | ret          | rsp+8  | [CFA-8] | [CFA-16] | DW_CFA_advance_loc: 16       |
|     |              |        |         |          | DW_CFA_def_cfa: RSP +8       |

## Call Frame Information: Bytecode – Example 3

|                   | CFA    | rip     |                            |
|-------------------|--------|---------|----------------------------|
| foo:              |        |         | DW_CFA_def_cfa: RSP +8     |
| 0: sub rsp, 8     | rsp+8  | [CFA-8] | DW_CFA_offset: RIP -8      |
| 4: test edi, edi  | rsp+16 | [CFA-8] | DW_CFA_advance_loc: 4      |
| 6: js 0x12        | rsp+16 | [CFA-8] | DW_CFA_def_cfa_offset: +16 |
| 8: call positive  | rsp+16 | [CFA-8] | DW_CFA_advance_loc: 13     |
| d: add rsp, 8     | rsp+16 | [CFA-8] | DW_CFA_remember_state:     |
| 11: ret           | rsp+8  | [CFA-8] | DW_CFA_def_cfa_offset: +8  |
| 12: call negative | rsp+16 | [CFA-8] | DW_CFA_advance_loc: 1      |
| 17: add rsp, 8    | rsp+16 | [CFA-8] | DW_CFA_restore_state:      |
| 1a: ret           | rsp+8  | [CFA-8] | DW_CFA_advance_loc: 8      |
|                   |        |         | DW_CFA_def_cfa_offset: +8  |

Remember stack: {}

## Call Frame Information: Bytecode – Exercise

- ▶ For the functions in `ex12.txt`:  
encode your CFI tables in DWARF CFI bytecode
- ▶ Can you reduce the size of the bytecode by changing or  
omitting instructions while maintaining correctness?
- ▶ Some ABIs guarantee a *red zone* below stack pointer
  - ▶ Red zone: area guaranteed to not be clobbered by signal handlers
  - ▶ E.g. x86-64 System V: 128 byte

## Call Frame Information: Bytecode

- ▶ DWARF<sup>86</sup> specifies bytecode for call frame information
- ▶ Self-contained section `.eh_frame` (or `.debug_frame`)
- ▶ Series of entries; two possible types distinguished using header
- ▶ Frame Description Entry (FDE): description of a function
  - ▶ Code range, instructions, pointer to CIE, language-specific data
- ▶ Common Information Entry (CIE): shared information among multiple FDEs
  - ▶ Initial instrs. (prepended to all FDE instrs.), personality function, alignment factors (constants factored out of instrs.), ...
- ▶ `readelf --debug-dump=frames <file>`  
`llvm-dwarfdump --debug-frame <file>`

<sup>86</sup>DWARF Debugging Information Committee. *DWARF Debugging Information Format Version 5*. Feb. 2017. 

## Call Frame Information: .eh\_frame\_hdr<sup>87</sup>

- ▶ Problem: linear search over – possibly many – FDEs is slow
- ▶ Idea: create binary search table over FDEs at link-time
- ▶ Ordered list of all function addresses and their FDE
- ▶ Unwinder does binary search to find matching FDE
- ▶ Separate program header entry: PT\_GNU\_EH\_FRAME
- ▶ Unwinder needs loader support to find these
  - ▶ `_dl_find_object` or `dl_iterate_phdr`
- ▶ FDEs and indices are cached to avoid redundant lookups

<sup>87</sup> [https://refspecs.linuxfoundation.org/LSB\\_4.1.0/LSB-Core-generic/LSB-Core-generic/ehframechpt.html](https://refspecs.linuxfoundation.org/LSB_4.1.0/LSB-Core-generic/LSB-Core-generic/ehframechpt.html)

## Call Frame Information: Assembler Directives

- ▶ Compilers produces textual CFI
- ▶ Assembler encodes CFI into binary format
  - ▶ Allows for integration of annotated inline assembly
  - ▶ Inline-asm also needs CFI directives
- ▶ Register numbers specified by psABI
- ▶ Wrap function with `.cfi_startproc/.cfi_endproc`
- ▶ Many directives map straight to DWARF instructions
  - ▶ `.cfi_def_cfa_offset 16; .cfi_offset %rbp, -16;`  
`.cfi_def_cfa_register %rbp`

## Call Frame Information: Assembler Directives – Example

```
int bar(int*);  
int foo(unsigned long x) {  
    int arr[x * 4];  
    return bar(arr);  
}  
  
gcc -O -S foo.c  
  
        .globl foo  
        .type foo, @function  
foo:  
        .cfi_startproc  
        push rbp  
        .cfi_def_cfa_offset 16  
        .cfi_offset 6, -16  
        mov rbp, rsp  
        .cfi_def_cfa_register 6  
        shl rdi, 4  
        sub rsp, rdi  
        mov rdi, rsp  
        call bar  
        leave  
        .cfi_def_cfa 7, 8  
        ret  
        .cfi_endproc  
        .size foo, .-foo
```

# Unwinding from Signal Handler

- ▶ Unwinding is conceptually supported even from signal handlers
  - ↝ Possible to get backtraces in-program in signal handler
- ▶ Unwind info must be correct at every single instruction (“asynchronous”)
  - ▶ Otherwise, it only needs to be correct at calls (“synchronous unwinding”)
- ▶ Is throwing exceptions from signal handlers safe? No!
  - ▶ Variables can be in an inconsistent state, e.g. in the middle of a copy
  - ▶ Possible and viable only under very limited and controlled circumstances

## Asynchronous vs. Synchronous Unwind Info

| Unwind Info                        | Synchronous   | Asynchronous      |
|------------------------------------|---------------|-------------------|
| Correct at                         | only at calls | every instruction |
| Usable for exceptions              | Yes           | Yes               |
| Usable for profiling               | —             | Yes               |
| Usable for stack trace in debugger | —             | Yes               |
| Unwind info size                   | smaller       | larger            |
| Usable on Windows                  | —             | Yes               |

## Unwinding: Other Platforms

- ▶ Unwinding depends *strongly* on OS and architecture
- ▶ GNU/Linux and many BSDs use DWARF
- ▶ Apple/Darwin has custom format for common case
- ▶ Windows has custom format
- ▶ IBM AIX has their own format
- ▶ AArch32 has another custom format (synchronous only)
- ▶ Additionally: minor differences for return address, stack handling, ...

---

Needs to work reliably for exception handling

# Compact Unwind Information I

- ▶ Problem: DWARF bytecode is rather repetitive and large
  - ▶ Many prologues have largely same structure, resulting in similar bytecode
- ▶ Problem: DWARF bytecode is not cheap to interpret
- ▶ Idea 1: represent each function with a fixed-size descriptor often sufficient for synchronous unwind info
- ▶ Code generator ensures that stack frame layout is same at all calls
- ▶ Restrict location of callee-saved registers due to limited encoding
- + Much more compact, typically 4-5 bytes per function
- Synchronous only, imposes code generator restrictions

## Compact Unwind Information II

- ▶ Representing asynchronous unwind info in compact form is difficult
- ▶ Windows/AArch64: impose fixed instr sequence for prologue/epilogue
- ▶ OpenVMS/x86-64: “async” unwind info is incorrect in prologue...
- ▶ ELF platforms have no compact unwind info (yet)

## C++ Exceptions

- ▶ Unwind info merely describes stack frame
- ▶ Missing: handler address, exception types
- ▶ Separate table mapping call to handler and exception types
- ▶ Typically in section `.gcc_except_table` (also non-GCC)
- ▶ Table interpreted by personality function (e.g. `__gxx_personality_v0`)
- ▶ Pointer to this table is encoded as language-specific data (LSDA) in unwind info

# Debugging: Wanted Features

- ▶ Get back trace ~~ CFI
- ▶ Map address to source file/line
- ▶ Show global and local variables
  - ▶ Local variables need scope information, e.g. shadowing
  - ▶ Data type information, e.g. int, string, struct, enum
- ▶ Set break point at line/function
  - ▶ Might require multiple actual breakpoints: inlining, template expansion
- ▶ Step through program by line/statement

## Debug Frame Information

- ▶ `.debug_frame` is very similar to `.eh_frame`
- ▶ Caveat: there are subtle encoding differences
- ▶ `eh_frame` allows for some (GNU) extensions

## Line Table

- ▶ Map instruction to: file/line/column and ISA mode
- ▶ Also: mark start of stmt; start of basic block; prologue end/epilogue begin
  - ▶ Provide breakpoint hints for lines, function entry/exit
- ▶ Table can be huge; idea: encode as bytecode
- ▶ Extracted information are bytecode registers
- ▶ Conceptually similar to CFI encoding
- ▶ `llvm-dwarfdump -v --debug-line` or `readelf -wlL`

# Debugging: Wanted Features

- ▶ Get back trace ~~> CFI
- ▶ Map address to source file/line ~~> Line Table
- ▶ Show global and local variables
  - ▶ Local variables need scope information, e.g. shadowing
  - ▶ Data type information, e.g. int, string, struct, enum
- ▶ Set break point at line/function ~~> Line Table/??
  - ▶ Might require multiple actual breakpoints: inlining, template expansion
- ▶ Step through program by line/statement ~~> Line Table

# DWARF: Hierarchical Program Description

- ▶ Extensible, flexible, Turing-complete<sup>88</sup> format to describe program
- ▶ Forest of Debugging Information Entries (DIEs)
  - ▶ Tag: indicates what the DIE describes
  - ▶ Set of attributes: describe DIE (often constant, range, or arbitrary expression)
  - ▶ Optionally children
- ▶ Rough classification:
  - ▶ DIEs for types: base types, typedef, struct, array, enum, union, ...
  - ▶ DIEs for data objects: variable, parameter, constant
  - ▶ DIEs for program scope: compilation unit, function, block, ...

<sup>88</sup> J Oakley and S Bratus. "Exploiting the Hard-Working DWARF: Trojan and Exploit Techniques with No Native Executable Code". In: *WOOT*. 2011. 

# DWARF: Data Types

```
DW_TAG_structure_type [0x2e]
  DW_AT_byte_size (0x08)
  DW_AT_sibling (0x4a)
  DW_TAG_member [0x37]
    DW_AT_name ("x")
    DW_AT_type (0x4a "int")
    DW_AT_data_member_location (0x00)
  DW_TAG_member [0x40]
    DW_AT_name ("y")
    DW_AT_type (0x4a "int")
    DW_AT_data_member_location (0x04)

DW_TAG_base_type [0x4a]
  DW_AT_byte_size (0x04)
  DW_AT_encoding (DW_ATE_signed)
  DW_AT_name ("int")
```

```
DW_TAG_pointer_type [0xb1]
  DW_AT_byte_size (8)
  DW_AT_type (0xb6 "char *")
  DW_TAG_pointer_type [0xb6]
  DW_AT_byte_size (8)
  DW_AT_type (0xbb "char")
  DW_TAG_base_type [0xbb]
  DW_AT_byte_size (0x01)
  DW_AT_encoding (DW_ATE_signed_char)
  DW_AT_name ("char")
```

# DWARF: Variables

```
DW_TAG_variable [0xa3]
  DW_AT_name          ("x")
  DW_AT_decl_file    ("/path/to/main.c")
  DW_AT_decl_line    (2)
  DW_AT_decl_column  (0x2e)
  DW_AT_type          (0x4a "int")
  DW_AT_location      (0x3b:
    [0x08, 0x0c): DW_OP_breg3 RBX+0, DW_OP_lit1, DW_OP_shl, DW_OP_stack_value
    [0x0c, 0x0d): DW_OP_entry_value(DW_OP_reg5 RDI), DW_OP_lit1, \
                  DW_OP_shl, DW_OP_stack_value)

DW_TAG_formal_parameter [0x7f]
  DW_AT_name      ("argc")
  // ...
```

## DWARF: Expressions

- ▶ Very general way to describe location of value: bytecode
- ▶ Stack machine, evaluates to location or value of variable
  - ▶ Simple case: register or stack slot
  - ▶ But: complex expression to recover original value after optimization
    - e.g., able to recover  $i$  from stored  $i - 1$
  - ▶ Unbounded complexity!
- ▶ Can contain control flow
- ▶ Can dereference memory, registers, etc.
- ▶ Used for: CFI locations, variable locations, array sizes, . . .

# DWARF: Program Structure

- ▶ Follows structure of code
- ▶ Top-level: compilation unit
- ▶ Entries for namespaces, subroutines (functions)
  - ▶ Functions can contain inlined subroutines
- ▶ Lexical blocks to group variables
- ▶ Call sites and parameters
- ▶ Each node annotated with pc-range and source location

# Debugging: Wanted Features

- ▶ Get back trace ↪ CFI
- ▶ Map address to source file/line ↪ Line Table
- ▶ Show global and local variables ↪ DIE tree
  - ▶ Local variables need scope information, e.g. shadowing
  - ▶ Data type information, e.g. int, string, struct, enum
- ▶ Set break point at line/function ↪ Line Table/DIE tree
  - ▶ Might require multiple actual breakpoints: inlining, template expansion
- ▶ Step through program by line/statement ↪ Line Table

## Other Debuginfo Formats

- ▶ DWARF is big despite compression
- ▶ Cannot run in time-constrained environments
  - ▶ Unsuitable for in-kernel backtrace generation
- ▶ Historically: STABS – string based encoding
  - ▶ Complexity increased significantly over time
- ▶ Microsoft: PDB for PE
- ▶ Linux kernel: CTF for simple type information
- ▶ Linux kernel: BTF for BPF programs

## Unwinding and Debuginfo – Summary

- ▶ Some languages/setups must be able to unwind the stack
- ▶ Needs meta-information on call frames
- ▶ DWARF encodes call frame information in bytecode program
- ▶ Runtime must efficiently find relevant information
- ▶ Stack unwinding typically done in two phases
- ▶ Functions have associated personality function to steer unwinding
- ▶ DWARF encodes debug info in tree structure of DIEs
- ▶ DWARF info can become arbitrarily complex

## Unwinding and Debuginfo – Questions

- ▶ What are alternatives to stack unwinding?
- ▶ What are the benefits of stack unwinding through metadata?
- ▶ What are the two phases of unwinding? Why is this separated?
- ▶ How to construct a CFI table for a given assembly code?
- ▶ How to construct DWARF ops for a CFI table?
- ▶ How to find the correct CFI table line for a given address?
- ▶ What is the general structure of DWARF debug info?

# Code Generation for Data Processing

## Lecture 13: JIT Compilation and Sandboxing

Alexis Engelke

Chair of Data Science and Engineering (I25)  
School of Computation, Information, and Technology  
Technical University of Munich

Winter 2025/26

# JIT Compilation

- ▶ Ahead-of-Time compilation not always possible/sufficient
- ▶ “Dynamic source” code: pre-compilation not possible
  - ▶ JavaScript, eval(), database queries
  - ▶ Binary translation of highly-dynamic/JIT-compiled code
- ▶ Additional verification/analysis or increased portability desired
  - ▶ (e)BPF, WebAssembly
- ▶ Dynamic optimization on common types/values
  - ▶ Run-time sampling of frequent code paths, allows dynamic speculation
  - ▶ Relevant for highly dynamic languages – otherwise prefer PGO<sup>89</sup>

<sup>89</sup>Profile-Guided Optimization; GCC: `-fprofile-generate` to store information about branches/values; `-fprofile-use` to use it

## JIT Compilation: Simple Approach

- ▶ Use standard compiler, write shared library
- ▶ Can write compiler IR, or plain source code
- ▶ `dlopen + dlsym` to find compiled function
- ▶ Example: `libgccjit`

- + Simple, fairly easy to debug
- Very high overhead, needs IO

## JIT: Allocating Memory

- ▶ `malloc()` – memory often non-executable
- ▶ `alloca()` – memory often non-executable
- ▶ `mmap(PROT_READ|PROT_WRITE|PROT_EXEC)` –  $W \oplus X$  may prevent this
  - ▶  $W \oplus X$ : a page must never be writable and executable at the same time
  - ▶ Some OS's (e.g. OpenBSD) and CPUs (Apple Silicon) strictly enforce this
- ▶ For code generation: map pages read-write
  - ▶ NetBSD needs special argument to allow remapping the page as executable
- ▶ Before execution: change protection to (read-)execute

# JIT: Making Code Executable

- ▶ Adjust page-level protections: `mprotect`
  - ▶ OS will adjust page tables
  - ▶ Typically incurs TLB shootdown
- ▶ Other steps might be needed, highly OS-dependent
  - ▶ Read manual

# JIT: Making Code Executable

- ▶ Flush instruction cache
  - ▶ Flush DCache to unification point (last-level cache)
  - ▶ Invalidate ICache in *all* cores for virtual address range
    - ▶ After local flush, kernel might move thread to other core with old ICache
- ▶ x86: coherent ICache/DCache hierarchy – hardware detects changes
  - ▶ Also includes: transparent (but expensive) detection of self-modifying code
- ▶ AArch64, MIPS, SPARC, ... (Linux): user-space instructions
- ▶ ARMv7, RISC-V<sup>90</sup> (Linux), all non-x86 (Darwin): system call
- ▶ Skipping ICache flush: spurious, hard-to-debug problems

<sup>90</sup>RISC-V has user `fence.i`, but only affects current core

## Code Generation: Differences AoT vs. JIT

|                | Ahead-of-Time            | JIT Compilation                                     |
|----------------|--------------------------|---|
| Code Model     | Arbitrary                | Large or Small-PIC with custom PLT                  |
| Relocations    | Linker/Loader            | JIT compiler/linker                                 |
| Symbols        | Linker/Loader            | JIT compiler/linker<br>may need application symbols |
| Memory Mapping | OS/Loader                | JIT compiler/linker                                 |
| EHFrame        | Compiler/Linker/Loader   | JIT compiler/linker<br>register in unwind runtime   |
| Debuginfo      | Compiler/Linker/Debugger | JIT compiler<br>register with debugger              |

- ▶ JIT compiler and linker are often merged

## JIT: Code Model

- ▶ Code can be located anywhere in address space
  - ▶ Cannot rely on linker to put in, e.g., lowest 2 GiB
- ▶ Large code model: allows for arbitrarily-sized addresses
- ▶ Small-PIC: possible for relocations inside object
  - ▶ Needs new PLT/GOT for other symbols
- ▶ Overhead trade-off: wide immediates vs. extra indirection (PLT)
- ▶ Further restrictions may apply (ISA/OS)

# JIT: Relocations and Symbols

- ▶ JIT compiler must take care of relocations
  - ▶ Can try to directly process relocations during machine code gen.
  - ▶ Not always possible: cyclic dependencies
  - ▶ Option: behave like normal compiler with separate runtime linker
- ▶ Code may need to access functions/global variables from application
  - ▶ Option: JIT compiler “hard-codes” relevant symbols
  - ▶ Option: application registers relevant symbols
  - ▶ Option: application linked with `--export-dynamic` and use `dlsym`

# JIT: Memory Layout

- ▶ Never place code and (writable) data on same page
  - ▶  $W \oplus X$ ; and writes near code can trigger self-modifying code detection
  - ▶ Avoid many small allocations with one page each
  - ▶ But: editing existing code pages is problematic
- ▶ Choose suitable alignment for code
  - ▶ Page alignment is too large: poor cache utilization
  - ▶ ICache cache line size not too relevant, decode buffer size is typical value: 16 bytes
  - ▶ Some basic blocks (e.g., hot loop entries) can benefit from 16-byte alignment

## JIT: .eh\_frame Registration (required for C++)

- ▶ Unwinder finds .eh\_frame using program headers
- ▶ Problem: JIT-compiled code has no program headers
- ▶ Idea: JIT compiler registers new code with runtime
- ▶ System unwinder provides `__register_frame` and `__deregister_frame`
  - ▶ System unwinder: `libgcc_s` or LLVM's `libunwind`
  - ▶ Call with address of first Frame Description Entry (FDE)
  - ▶ Historically also called by init code

## JIT: GDB Debuginfo Registration (optional)

- ▶ GDB finds debug info from section headers of DSOs
- ▶ Problem: JIT-compiled code has no DSO
- ▶ Idea: JIT compiler registers new code with debugger
- ▶ Define function `__jit_debug_register_code` and global var.  
`__jit_debug_descriptor`
  - ▶ Call function on update; GDB places breakpoint in function
  - ▶ Prevent function from being inlined
- ▶ Descriptor is linked list of in-memory object files
  - ▶ Needs relocations applied, also for debug info
- ▶ Users: LLVM, Wasmtime, HHVM, ... ; consumers: GDB, LLDB

## JIT: Linux perf Registration (optional)

- ▶ perf tracks binary through backing file of `mmap`
- ▶ Problem 1: JIT-compiled code has no backing file for its `mmap` region
- ▶ Problem 2: after tracing, JIT-compiled code is gone
- ▶ Goal 1: map instructions to functions
- ▶ Goal 2: keep JIT-compiled code for detailed analysis
- ▶ Approach 1: dump function limits to `/tmp/perf-<PID>.map`<sup>91</sup>
  - ▶ Text file; format: `startaddr size name\n`
- ▶ Approach 2: *needs an extra slide*

<sup>91</sup><https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/perf/Documentation/jit-interface.txt>

## JIT: Linux perf JITDUMP format (optional)

- ▶ JIT-compiler dumps function name/address/size/code<sup>92</sup>
  - ▶ JITDUMP file: record list for each function, may contain debuginfo
  - ▶ File name must be `jit-<PID>.dump`
- ▶ JIT-compiler mmaps part of the file as executable somewhere
  - ▶ Only use: perf keeps track of executable mappings ⇔ mapping is JIT marker, s.t. perf can find the file later
- ▶ Need to run `perf report` with `-k 1` to use monotonic clock
- ▶ After profiling: `perf inject --jit -i perf.data -o jit.data`
  - ▶ Extracts functions from JITDUMP, each into its own ELF file
  - ▶ Changes mappings of profile to refer to newly created files
- ▶ `perf report -i jit.data` – Profit!

<sup>92</sup><https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/perf/Documentation/jitdump-specification.txt>

# Compilation Time

- ▶ Problem: code generation takes time
  - ▶ Especially high-complexity frameworks like GCC or LLVM
- ▶ Compilation time of JIT compilers often matters
  - ▶ Example: website needing JavaScript on page load
  - ▶ Example: compiling database query
- ▶ Functions executed once are not worth optimizing
- ▶ But: often not known in advance
- ▶ Idea: adaptive compilation
- ▶ Incrementally spend more time on optimization

## Compilation Time: Simple Approach

### Caching

- ▶ Doesn't work on first execution

# Adaptive Execution

- ▶ Execution tiers have different compile-time/run-time tradeoffs
  - ▶ Bytecode interpreter: very fast/slow
  - ▶ Fast compiler: medium/medium
  - ▶ Optimizing compiler: slow/fast
- ▶ Start with interpreter, profile execution
  - ▶ E.g., collect stats on execution frequency, dynamic types, ...
- ▶ For program worth optimizing, switch to next tier
  - ▶ Depends on profile information, e.g. only optimize hot code
  - ▶ Compile in background, switch when ready

## Adaptive Execution: Switching Tiers

- ▶ Switching only possible at compiler-defined points
  - ▶ Needs to serialize relevant state for other tier
- ▶ Simple approach: only switch at function boundaries
  - ▶ Simple, well-defined boundaries; unable to switch inside loop
- ▶ Complex approach: allow switching at loop headers/everywhere
  - ▶ Needs tracking of much more meta-information
  - ▶ All entry points need well-defined interface
  - ▶ All exit points need info to recover complete state
  - ▶ Severely limits optimizations; all loops become irreducible
- ▶ Using LLVM is possible, but not a good fit

# Adaptive Execution: Partial Compilation and Speculation

- ▶ Observation: even in hot functions, many branches are rarely used
- ▶ Optimizing cold code is wasted time(/energy)
- ▶ Observation (JS): functions often get called with same data type
- ▶ Specializing on structure allows removing string lookup for fields
- ▶ Idea: speculate on common path using profiling data
- ▶ Add check whether speculation holds; if not, use side-exit
  - ▶ Side-exit can be patched later with actual code
- ▶ Side-exit must serialize all relevant state for lower tier
  - ▶ “Deoptimization”

# Sandboxing

- ▶ Executing untrusted code without additional measures may harm system
- ▶ Untrusted input may expose vulnerabilities
- ▶ Goal 1: execute untrusted code without impacting security
  - ▶ Code in higher-level representation allows for further analyses but needs JIT compilation for performance
- ▶ Goal 2: limit impact potential of new vulnerabilities
- ▶ Other goals: portability, resource usage, performance, usability, language flexibility

## Approach: Sandbox Operating System as-is

- ▶ Idea: put entire operating system in sandbox (“virtual machine”)
- ▶ Widely used in practice
- ▶ Virtualization needs hardware and OS support
  - ▶ CPU has hypervisor mode which controls guest OS;  
offers nested paging, hypercalls from guest OS to hypervisor
- + Good usability and performance
- + Strong isolation
- Rather high overhead on resource usage: completely new OS
- Inflexible and high start latency (seconds)

## Approach: Sandbox Native Code as-is

- ▶ Idea: strongly restrict possibilities of native code
- ▶ Restrict system calls: seccomp
  - ▶ Filter program for system calls depending on arguments
- ▶ Separate namespaces: network, PID, user, mount, ...
  - ▶ Isolate program from rest of the system
  - ▶ Need to allow access to permitted resources
- ▶ Limit resource usage: memory, CPU, ... cgroups

## Approach: Sandbox Native Code as-is

- ▶ Frequently and widely used (“container”)
- + Good usability and performance, low latency (milliseconds)
- + Finer grained control of resources
- ~ Resource usage: often completely new user space
- Weak isolation: OS+CPU often bad at separation
  - ▶ Kernel has a fairly large interface, not hardened against bad actors
  - ▶ Privilege escalation happens not rarely

## Approach: Sandbox Native Code with Modification

- ▶ Software-based Fault Isolation (SFI)
- ▶ Idea: enforce limitations on machine code
  - ▶ Define restrictions on machine code, e.g. no unbounded memory access
  - ▶ Modify compiler to comply with restrictions
  - ▶ Verify program at load time
- ▶ Verifier ensures: no out-of-bounds accesses, no jump to unverified code
- ▶ Google Native Client<sup>93</sup>, originally x86-32, ported to x86-64 and ARM
- ▶ Designed as browser extension
- ▶ Native code shipped to browser, executed after validation

<sup>93</sup> B Yee et al. "Native client: A sandbox for portable, untrusted x86 native code". In: SP. 2009, pp. 79–93.

# NaCl Constraints on i386

- ▶ Problem: dynamic code not verifiable
  - ⇒ No self-modifying/dynamically generated code
- ▶ Problem: overlapping instructions
  - ⇒ All “valid” instructions must be reachable in linear disassembly
  - ⇒ Direct jumps must target valid instructions
  - ⇒ No instruction may cross 32-byte boundary
  - ⇒ Indirect jumps/returns must be and `eax, -32; jmp eax`
- ▶ Problem: arbitrary memory access inside virtual memory
  - ⇒ Separate process, use segmentation restrict accessible memory
- ▶ Problem: program can run arbitrary CPU instructions
  - ⇒ Blacklist “dangerous” instructions

# NaCl on non-i386 Systems

- ▶ Other architectures<sup>94</sup> use base register instead of segment offsets
  - ▶ Additional verification required
- ▶ Deprecated in 2017 in favor of WebAssembly
- ▶ Similar approach recently revived under new name LFI<sup>95</sup>

- + Nice idea, high performance (5–15% overhead)
- ~ Instruction blacklist not a good idea
- Not portable, severe restrictions on emitted code
- High verification complexity, error-prone

<sup>94</sup> D Sehr et al. “Adapting Software Fault Isolation to Contemporary {CPU} Architectures”. In: *19th USENIX Security Symposium (USENIX Security 10)*. 2010.

<sup>95</sup> Z Yedidia. “Lightweight fault isolation: Practical, efficient, and secure software sandboxing”. In: *ASPLOS*. 2024, pp. 649–665

## Approach: Using Bytecode

- ▶ Idea: compile code to bytecode, JIT-compile on host
  - ▶ Benefit: verification easy – all code generated by trusted compiler
  - ▶ Benefit: more portable
- ▶ Java applets
- ▶ PNaCl: bytecode version of NaCl

- + Fairly high performance, portable
- ~ Heavy runtime environment
  - ▶ Especially criticized for Java applets
- Very high complexity and attack surface

## Approach: Subset of JavaScript: asm.js

- ▶ Situation: fairly fast JavaScript JIT-compilers present
- ▶ Idea: use subset of JavaScript known to be compilable to efficient code
  - ▶ All browsers/JS engines support execution without further changes
- ▶ asm.js<sup>96</sup>: strictly, statically typed JS subset; single array as heap
- ▶ JS code generated by compilers, e.g. Emscripten
- ▶ JavaScript has single numeric type, but asm.js supports int/float/double
  - ▶ Coercion to integer: `x|0`
  - ▶ Coercion to double: `+x`
  - ▶ Coercion to float: `Math.fround(x)`

## asm.js Example

```
var log = stdlib.Math.log;
var values = new stdlib.Float64Array(buffer);
function logSum(start, end) {
    start = start|0; // parameter type int
    end = end|0; // parameter type int

    var sum = 0.0, p = 0, q = 0;

    // asm.js forces byte addressing of the heap by requiring shifting by 3
    for (p = start << 3, q = end << 3; (p|0) < (q|0); p = (p + 8)|0) {
        sum = sum + +log(values[p>>3]);
    }

    return +sum;
}
```

Example taken from the specification

## Approach: Encode asm.js as Bytecode

- ▶ Parsing costs time, type restrictions increase code size
- ▶ Idea: encode asm.js source as bytecode
- ▶ First attempt: encode abstract syntax tree in pre-order
- ▶ Second attempt: encode abstract syntax tree in post-order
- ▶ Third attempt: encode as stack machine
- ▶ ... and WebAssembly was born

## Approach: Using Bytecode – WebAssembly

- ▶ Strictly-typed bytecode format encoding a stack machine
- ▶ Global variables and single, global array as memory
- ▶ Functions have local variables
  - ▶ Parameters pre-populated in first local variables
  - ▶ No dynamic/addressable stack space! ↪ part of global memory used as stack
- ▶ Operations use implicit stack
  - ▶ Stack has well-defined size and types at each point in program
- ▶ Structured control flow
  - ▶ Blocks to skip instructions, loop to repeat, if-then-else
  - ▶ No irreducible control flow representable

## Approach: Use Verifiable Bytecode – eBPF

- ▶ Problem: want to ensure termination within certain time frame
- ▶ Problem: need to make sure *nothing* can go wrong – no sandbox!
- ▶ Idea: disallow loops and undefined register values, e.g. due to branch
  - ▶ Combinatorial explosion of possible paths, all need to be analyzed
  - ▶ No longer Turing-complete
- ▶ eBPF: allow user-space to hook into various Linux kernel parts
  - ▶ E.g. network, perf sampling, ...
- ▶ Strongly verified register machine
- ▶ JIT-compiled inside kernel

## JIT Compilation and Sandboxing – Summary

- ▶ JIT compilation required for dynamic source code or bytecode
- ▶ Bytecode allows for simpler verification than machine code, but is more compact
- ▶ Producing JIT-compiled code needs CPU, OS, and runtime support
- ▶ JIT compilers can do/need to do different kinds of optimizations  
adaptive execution is key technique to hide compilation latency
- ▶ Sandboxing can be done at various levels and granularities
- ▶ Virtualization and containers widely used for whole applications
- ▶ Bytecode formats popular for ad-hoc distribution of programs

## JIT Compilation and Sandboxing – Questions

- ▶ When is JIT-compilation beneficial over Ahead-of-Time compilation?
- ▶ How can JIT-compilation be realized using standard compilers?
- ▶ How can code be made executable after writing it to memory?
- ▶ Why do some architectures require a system call for ICache flushing?
- ▶ How can JIT compilers trade between compilation latency and performance?
- ▶ Why is sandboxing important?
- ▶ What methods of deploying code for sandboxed execution are widely used?

# Code Generation for Data Processing

## Lecture 14: Binary Translation

Alexis Engelke

Chair of Data Science and Engineering (I25)  
School of Computation, Information, and Technology  
Technical University of Munich

Winter 2025/26

# Motivation

- ▶ Run program on other architecture
- ▶ Use-case: application compatibility
  - ▶ Other architecture with incompatible instruction encoding
  - ▶ Applications using unavailable ISA extensions<sup>97</sup>
- ▶ Use-case: architecture research
  - ▶ Development of new ISA extensions without existing hardware

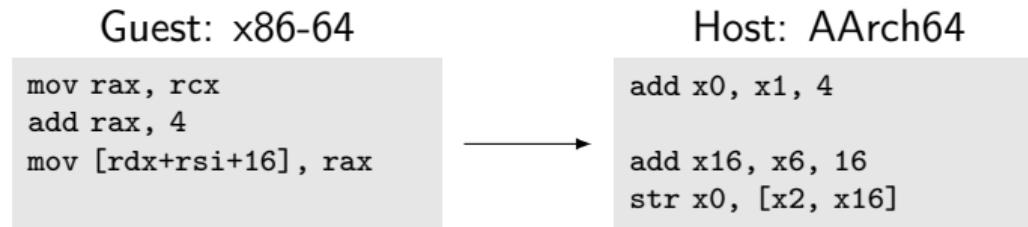
<sup>97</sup>Exception-based implementation possible, but slow.

# ISA Emulation

- ▶ Simplest approach: interpreting machine code
  - ▶ Simulate individual instructions, don't generate new code
- ▶ Frequently used approach before JIT-compilation became popular
- + Simple, works almost anywhere, high correctness
- Very inefficient

# Binary Translation

- ▶ Idea: translate guest machine code to host machine code
- ▶ Replace interpretation overhead with translation overhead
- ▶ Difficult: very rigid semantics, but few code constraints imposed
  - ▶ Self-modifying code, overlapping instructions, indirect jumps
  - ▶ Exceptions with well-defined states, status flags



Warning for same-ISA translation: passing all instructions through as-is is a bad idea! Behavior might differ.

# Static vs. Dynamic Binary Translation

## Static BT

- ▶ Translate guest executable into host executable
- ▶ Do translation before execution
- + Low runtime overhead
- Binaries tend to be huge
- Cannot handle all cases
  - ▶ E.g., JIT-compiled code

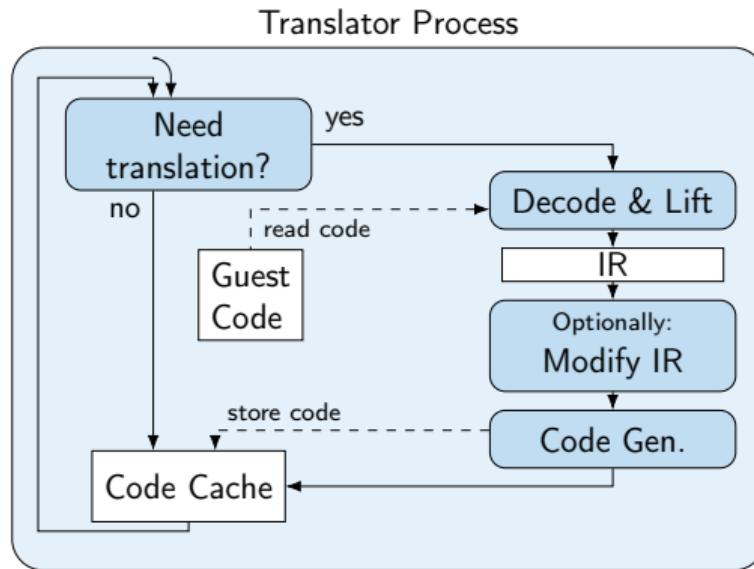
## Dynamic BT

- ▶ Translate code on-the-fly during program execution
- ▶ Host code just lives in memory
- + Allows for high correctness
- ~ Can use JIT optimizations
- Translation overhead at run-time

# Static Binary Translation

- ▶ Goal: create new binary for host with same functionality
- ▶ Program may access its own code/data in various ways
  - ▶ Guest binary must be retained as-is in-place
- ▶ Indirect jumps problematic
  - ▶ Need prediction of all possible targets
  - ▶ Keeping lots of dynamically possible entries prohibits optimizations
- ▶ JIT-compiled/self-modifying code impossible to handle
- ▶ Purely static translation impossible for the general case

# Dynamic Binary Translation



- ▶ Iteratively translate code chunks on-demand
  - ▶ Typically basic blocks
- ▶ Store new code in-memory for execution and later re-use
- ▶ Code executed in same address space as original
  - ▶ Guest code/data must be accessible

# Dynamic Binary Translation: Code Fragment

## RISC-V Code

```
400560: slli a0, a0, 2
400564: jalr x0, ra, 0 // ret
```

## Semantical representation

```
uintptr_t trans_400560(uint64_t* regs) {
    regs[10] = regs[10] << 2;
    return regs[1];
}
```

## Translation Engine

```
void emulate(uintptr_t pc) {
    uint64_t* regs = init();
    while (true)
        pc = translate(pc)(regs);
}
```

```
// or with tail call:
_Noreturn void trans_400560(uint64_t* regs) {
    regs[10] = regs[10] << 2;
    translate(regs[1])(regs);
    // unreachable
}
```

## Guest State

- ▶ Guest CPU state must be completely emulated
  - ▶ Registers: general-purpose, floating-point, vector, ...
  - ▶ Flags, control registers, system registers, segments, TLS base
- ▶ Memory – user-space emulation: use host address space
  - + no overhead through additional indirection
  - no isolation between emulator and guest
- ▶ Memory – system emulation: need software/hardware paging support
  - ▶ Software implementation: considerable performance overhead
  - ▶ Hardware implementation: guest and host need same page size

## Guest Interface

- ▶ User-space emulation: OS interface needs to be emulated
  - ▶ Mainly system calls, but also vDSO, memory maps, ...
  - ▶ Host libraries are hard to use: ABI differences (e.g. struct padding)
  - ▶ Syscall emulation tedious: different flag numbers, arguments, orders  
structs have different fields, alignments, padding bytes
- ▶ System-level emulation: CPU interface for operating systems
  - ▶ **Many** system/control registers
  - ▶ Different execution modes, memory configurations, etc.
  - ▶ Emulation of hardware components

# Dynamic Binary Translation: Optimizations

- ▶ Fully correct emulation of CPU (and OS) is slow
  - ▶ Every memory access is a potential page fault
  - ▶ Signals can be delivered at any instruction boundary
  - ▶ *many* other traps...
- ▶ But: these “special” features are used extremely rarely
- ▶ Idea: optimize for common case
- ▶ Aggressively trade correctness for performance

## Translation Granularity

- ▶ Larger translation granules allow for more optimization
  - ▶ E.g., omit status flag computation; fold immediate construction
- ▶ Instruction: great for debugging
- ▶ Basic block: allows for some important opt.
  - ▶ Easy to detect (up to next branch), easy to translate (no control flow)
- ▶ Superblock: up to next unconditional jump
  - ▶ Reduces transfers between blocks in fallthrough case
  - ▶ Translated code not necessarily executed
- ▶ Function: follow all conditional control flow
  - ▶ Allows most optimizations, e.g. for loop induction variables
  - ▶ Complex codegen, ind. jumps problematic, lot of code never executed

# Chaining

- ▶ Observation: many basic blocks have constant successors
  - ▶ Often conditional branches with fallthrough and constant offset
- ▶ (Hash)map lookup and indirect jump after every block expensive
- ▶ Idea: after successor is translated, patch end to jump directly to that code
  - ▶ First execution is expensive, later executions are fast

```
// Initially generated code           // After patching
// ...
mov rdi, 0x40068c
lea rsi, [rip+1f]
jmp translate_and_dispatch
1:.byte ... // store patch information
                                         // ...
                                         jmp trans_40068c
                                         // (garbage remains)
```

## Chaining: Limitations

- ▶ First execution still slow, patching adds overhead
  - ▶ Can speculatively translate continuations
  - ▶ Translation of possibly unneeded code adds overhead
- ▶ Does not work for indirect jumps
  - ▶ Not necessarily predictable, esp. when considering a single basic block
  - ▶ Occur fairly often: function returns
- ▶ Removing translated functions from code cache becomes harder
  - ▶ Arbitrary other code may directly branch to translated chunk
  - ▶ Often solved by limiting chaining to same page or memory region

# Return Address Prediction

- ▶ Observation: function calls very often return ordinarily
  - ▶ Return is an indirect jump, *but* highly predictable
  - ▶ But: even for “normal” code, this is not always the case:  
`setjmp/longjmp`, exceptions
- ▶ Hardware has return address stack keeping track of call stack
  - ▶ `call` pushes next address to stack, `ret` predicted to pop
  - ▶ Usually implemented as 16/32 entry ring buffer
- ▶ Idea: similarly optimize for common case of ordinary return

# Return Address Prediction in DBT

- ▶ Option 1: keep separate shadow stack of guest/host target pairs
  - ▶ Can be implemented as ring buffer, too
  - ▶ Pop from stack needs verification of actual guest return address
    - Doesn't use host hardware return address prediction
- ▶ Option 2: use host stack as shadow stack
  - ▶ Allows using host call/ret instructions
  - ▶ Verification before/after return still required
    - Can degenerate, need to bound shadow stack  
(guest might repeatedly call, discard return address, but never return)

# Status Flags

- ▶ Observation: many status flags are rarely used
- ▶ But: eager computation can be expensive
  - ▶ E.g., x86 parity (PF) or auxiliary carry (AF)
- ▶ Idea: compute flags only when needed
- ▶ On flag computation, store operands needed for flag computation
- ▶ Flag usage in same block allows for optimizations
  - ▶ E.g., use idiomatic branches (jle, ...)
- ▶ Flag usage in different block: compute flags from operands
  - ▶ More expensive, but happens seldomly

## Correct Binary Translation

- ▶ Goal 1: precise emulation – application works properly
- ▶ Goal 2: stealthness/isolation – application can't compromise DBT
- ▶ Problem: CPU and OS have huge and very-well-specified interfaces
  - ▶ ... and even if unspecified, software often depends on it
- ▶ Increased difficulty: different guest/host architectures
  - ▶ E.g., different page size or memory semantics
- ▶ Increased difficulty for user-space: different guest/host OS
  - ▶ Depending on syscall interface, nearly impossible (see WSL1)

# POSIX Signals

- ▶ POSIX specifies signals, which can interrupt program at any point
- ▶ Kernel pushes signal frame to stack with user context and calls signal handler
- ▶ Signal handler can read/modify user context and continue execution
- ▶ Synchronous signals: e.g., SIGSEGV, SIGBUS, SIGFPE, SIGILL
  - ▶ For example, due to page fault or FP exception
  - ▶ Delivered in response to “error” in current thread
- ▶ Asynchronous signals: e.g., SIGINT, SIGTERM, SIGCHILD
  - ▶ Delivered externally, e.g. using `kill`
  - ▶ Can be delivered to any thread at any time
  - ▶ (usually a bad idea to use them)

## Correct DBT: Signals

- ▶ DBT must register signal handler and propagate signals
- ▶ Synchronous signals
  - ▶ Delivered at “constrainable” points in program
  - ▶ *Must* recover fully consistent guest architectural state
  - ▶ JIT-compiled code must be sufficiently annotated for this
- ▶ Asynchronous signals
  - ▶ Can really be delivered at any time
  - ▶ Must not be immediately delivered to guest
  - ▶ ↵ Usually delivered when convenient
  - ▶ But: real-time signals have special semantics

## Correct DBT: Memory Accesses

- ▶ Option: emulating paging in software (slow, but works)
  - ▶ Every memory access becomes a hash table lookup
  - ▶ Shared memory still problematic: host OS might have larger pages
- ▶ Using host paging is much faster, but problematic for correctness
- ▶ Host OS might have larger pages
- ▶ Every memory access can cause a page fault (see signal handling)
- ▶ Guest can access/modify arbitrary addresses in its address space... including the DBT and its code cache
- ▶ Tracking read/write/execute permissions, e.g. check X before translation

## Correct DBT: Memory Ordering

- ▶ CPUs (aggressively) reorder memory operations
  - ▶ x86: total store ordering – stores can be reordered after loads
  - ▶ Most others: weak ordering – everything can be reordered
- ▶ Relevant for multi-core systems: other thread can observe ordering
- ▶ Atomic operations and fences limit reordering (e.g., acq/rel/seqcst)
- ▶ Emulating weak memory on TSO: easy
- ▶ Emulating TSO on weak memory: hard
  - ▶ Can try to make all operations atomic
  - ▶ Atomic operations often need alignment guarantees (not on x86)
  - ▶ Only viable solution so far: insert fences everywhere

## Correct DBT: Self-modifying Code

- ▶ Writable code regions (or with `MAP_SHARED`) can change at any time
- ▶ Idea: before translation, remap as read-only
- ▶ On page fault (`SIGSEGV`), remove relevant parts from code cache
  - ▶ Requires code cache segmentation and mapping of code to original page
- ▶ When executing possibly modifiable code: every store can change code!
- ▶ Doesn't easily work for shared memory, need to track this, too
  - ▶ Might be impossible when shared with other process

## Correct DBT: Floating-point

- ▶ Floating-point arithmetic is standardized in IEEE-754
- ▶ ...except for some details and non-standard operations
- ▶ x86 `maxsd`: if one operand is NaN, result is second operand
- ▶ RISC-V `fmax.d`: if one operand is NaN, result is non-NaN operand
- ▶ AArch64 `fmax`: if one operand is NaN, result is NaN operand
  - ▶ Unless configured differently in `fpcr`
- ▶ Correctness typically requires software emulation (e.g., QEMU does this)

## Correct DBT: OS and CPU Specifics

- ▶ Emulating all syscalls correctly is hard
  - ▶ Version-specifics, structure layouts, feature support
  - ▶ Huge interface
- ▶ `/proc/self/*` – how to emulate?
  - ▶ Catch all file system accesses? Follow all possible symlinks?
  - ▶ What if procfs is mounted somewhere else?
- ▶ `cpuid` – how to emulate?
  - ▶ Cache sizes, processor model, ...
  - ▶ Application can do timing experiment to detect DBT

## Binary Translation – Summary

- ▶ ISA emulation often used for cross-ISA program execution
- ▶ Binary Translation allows for more performance than interpretation
- ▶ Static Binary Translation handles whole program ahead-of-time
- ▶ Dynamic Binary Translation translates code on-demand
- ▶ ISA often highly restricts optimization possibilities
- ▶ Optimizations typically very low-level
- ▶ Correct emulation of CPU/OS challenging due to large interface

## Binary Translation – Questions

- ▶ What are use cases of binary translation?
- ▶ What is the difference between static and dynamic binary translation?
- ▶ Why is static BT strictly less powerful than dynamic BT?
- ▶ What are typical translation granularities for DBT?
- ▶ How to optimize control flow handling in DBT?
- ▶ Why is correct binary translation hard to optimize?
- ▶ What problem can occur when not emulating paging for user-space emulation?

# Code Generation for Data Processing

## Lecture 15: Query Compilation

Alexis Engelke

Chair of Data Science and Engineering (I25)  
School of Computation, Information, and Technology  
Technical University of Munich

Winter 2025/26

# Motivation: Fast Query Execution

- ▶ Databases are often used in latency-critical situations
  - ▶ Mostly transactional workload
- ▶ Databases are often used for analyzing large data sets
  - ▶ Mostly analytical workload; queries can be complex
  - ▶ Latency not that important, but through-put is
- ▶ Databases are also used for storing data streams
  - ▶ Streaming databases, e.g. monitoring sensors
  - ▶ Throughput is important; but queries often simple

# Data Representation

- ▶ Relational algebra: set/bag of tuples
  - ▶ Tuple is sequence of data with different types
  - ▶ All tuples in one relation have same schema
  - ▶ Order does not matter
  - ▶ Duplicates might be possible (bags)
- ▶ Might have special values, e.g. NULL
- ▶ Values might be variably-sized, e.g. strings
- ▶ But: databases have *high* degree of freedom wrt. data representation

# Query Plan

- ▶ Query often specified in “standardized format” (SQL)
- ▶ SQL is transformed into (logical) query plan
- ▶ Logical query plan is optimized
  - ▶ E.g., selection push down, transforming cross products to joins, join ordering
- ▶ Physical query plan
  - ▶ Selection of actual implementation for operators
  - ▶ Determine use index structures, access paths, etc.

## Query Plan: Subscripts

- ▶ Query plan strongly depends on query
- ▶ Operators have query-dependent subscripts
  - ▶ E.g., selection/join predicate, aggregation function, attributes
  - ▶ Implementation of these also depends on schema
- ▶ Can include arbitrarily complex expressions
- ▶ Examples:  $\bowtie_{s.matrnr=h.matrnr}^{HJ}, \sigma_{a.x < 5 \cdot (b.y - a.z)}$

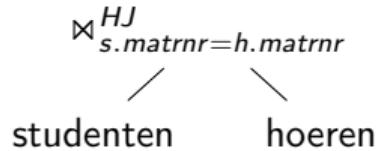
## Subscripts: Execution

- ▶ Option: keep as tree, interpret
  - + Simple, flexible
  - Slow
- ▶ Option: compile to bytecode
  - + More efficient
  - More effort to implement, some compile-time
- ▶ Option: compile to machine code
  - ▶ Code can be complex to accurately represent semantics
  - + Most efficient
  - Most effort to implement, may need short compile-times

# SQL Expressions

- ▶ Arithmetic expressions are fairly simple
  - ▶ Need to respect data type and check for errors (e.g., overflow)
  - ▶ Numbers in SQL are (fixed-point) decimals
- ▶ String operations can be more complex
  - ▶ like expressions
  - ▶ Regular expressions – strongly benefit from optimized execution
  - ▶ But: full-compilation may not be worth the effort  
often, calling runtime functions is beneficial
  - ▶ Support Unicode for increased complexity

# Query Execution: Simplest Approach



- ▶ Execute operators individually
- ▶ Materialize all results after each operator
- ▶ “Full Materialization”

- + Easy to implement
- + Can dynamically adjust plan
- Inefficient, intermediate results can be big

## Iterator Model<sup>98</sup>

- ▶ Idea: stream tuples through operators
- ▶ Every operator implements set of functions:
  - ▶ `open()`: initialization, configure with child operators
  - ▶ `next()`: return next tuple (or indicate end of stream)
  - ▶ `close()`: free resources
- ▶ Current tuple can be pass as pointer or held in global data space
  - ▶ Possible: only single tuple is processed at a time

<sup>98</sup> G Graefe. "Volcano—an extensible and parallel query evaluation system". In: *IEEE Transactions on Knowledge and Data Engineering* 6.1 (1994), pp. 120–135.

# Iterator Model: Example

```
struct TableScan : Iter {
    Table* table;
    Table::iterator it;
    void open() { it = table.begin(); }
    Tuple* next() {
        if (it != table.end())
            return *it++;
        return nullptr;
    } };
struct Select : Iter {
    Predicate p;
    Iter base;
    void open() { base.open(); }
    Tuple* next() {
        while (Tuple* t = base.next())
            if (p(t))
                return t;
        return nullptr;
    } };
struct Cross : Iter {
    Iter left, right;
    Tuple* curLeft = nullptr;
    void open() { left.open(); }
    Tuple* next() {
        while (true) {
            if (!curLeft) {
                if (!(curLeft = left.next())))
                    return nullptr;
                right.open();
            }
            if (Tuple* tr = right.next())
                return concat(curLeft, tr);
            curLeft = nullptr;
        }
    } };
};
```

- ▶ HashJoin builds hash table on first read; materialization might be useful

## Iterator Model

- ▶ “Pull-based” approach
- ▶ Widely used (e.g., Postgres)
- ▶ Often have separate function for `first()` or `rewind`

- + Fairly straight-forward to implement
- + Avoids data copies, no dynamic compilation
- Only single tuple processed at a time, bad locality
- Huge amount virtual function calls

## Push-based Model<sup>99</sup>

- ▶ Idea: operators push tuples through query plan bottom-up
- ▶ Every operator implements set of functions:
  - ▶ `open()`: initialization, store parents
  - ▶ `produce()`: produce items
    - ▶ Table scan calls `consume()` of parents
    - ▶ Others call `produce()` of their child
  - ▶ `consume()`: consume items from children, push them to parents
- ▶ Only one tuple processed at a time

<sup>99</sup> T Neumann. "Efficiently compiling efficient query plans for modern hardware". In: VLDB 4.9 (2011), pp. 539–550.

# Push-based Model: Example

```
struct TableScan {
    Table table;
    Consumer cons;
    void produce() {
        for (Tuple* t : table)
            cons.consume(t, this);
    }
};

struct Select {
    Predicate p;
    Producer prod;
    Consumer cons;
    void produce() { prod.produce(); }
    void consume(Tuple* t, Producer src) {
        if (p(t))
            cons.consume(t)
    }
};
```

```
struct Cross {
    Producer left, right;
    Consumer cons;
    Tuple* curLeft = nullptr;
    void produce() { left.produce(); }
    // Materializing one side might be better
    void consume(Tuple* t, Producer src) {
        if (src == left) {
            curLeft = t;
            right.produce();
        } else { // src == right
            cons.consume(concat(curLeft, t));
        }
    }
};
```

## Push-based Model

- ▶ “Push-based” approach
- ▶ More recent approach
- + Fairly straight-forward, but less intuitive than iterator
- + Avoids data copies, no dynamic compilation
- Only single tuple processed at a time, bad locality
- Huge amount virtual function calls

# Pull-based Model vs. Push-based Model<sup>100</sup>

- ▶ Two fundamentally different approaches
- ▶ Push-based approach can handle DAG plans better
  - ▶ Pull-model: needs explicit materialization or redundant iteration
  - ▶ Push-model: simply call multiple consumers
- ▶ Performance: nearly identical
  - ▶ Push-based model needs handling for limit operations otherwise table scan would not stop, even all tuples are dropped
- ▶ But: push-based code is nice after inlining

<sup>100</sup> A Shaikhha, M Dashti, and C Koch. "Push versus pull-based loop fusion in query engines". In: *Journal of Functional Programming* 28 (2018).

# Pipelining

- ▶ Some operators need materialized data for their operation
  - ▶ Pipeline breaker: operator materializes input
  - ▶ Full pipeline breaker: operator materializes complete input before producing
- ▶ Other operators can be *pipelined* (i.e., no materialization)
- ▶ Aggregations
- ▶ Join needs one side materialized (pipeline breaker on one side)
- ▶ Sorting needs all data (full pipeline breaker)
- ▶ System needs to take care of semantics, e.g. for memory management

## Code Generation for Push-Based Model

- ▶ Inlining code in push-based model yields nice code
- ▶ No virtual function calls
- ▶ Producer iterates over materialized tuples and loads relevant data
  - ▶ Tight loop over base table – data locality
- ▶ Operators of parent operators are applied inside the loop
- ▶ Pipeline breaker materializes result (e.g., into hash table)

# Code Generation: Example

$$\sigma_{s.\text{matrnr}=h.\text{matrnr}}$$

```
  |  
  X  
  / \   
studenten hoeren
```

```
struct Query {  
    Output out;  
    Table tabLeft, tabRight;  
    Tuple* curLeft = nullptr;  
    void produce() {  
        for (Tuple* tl : tabLeft) {  
            curLeft = tl;  
            for (Tuple* tr : tabRight) {  
                Tuple* t = concat(curLeft, tr);  
                if (t.s_matrnr == t.h_matrnr)  
                    out.write(t);  
            }  
        }  
    }  
};
```

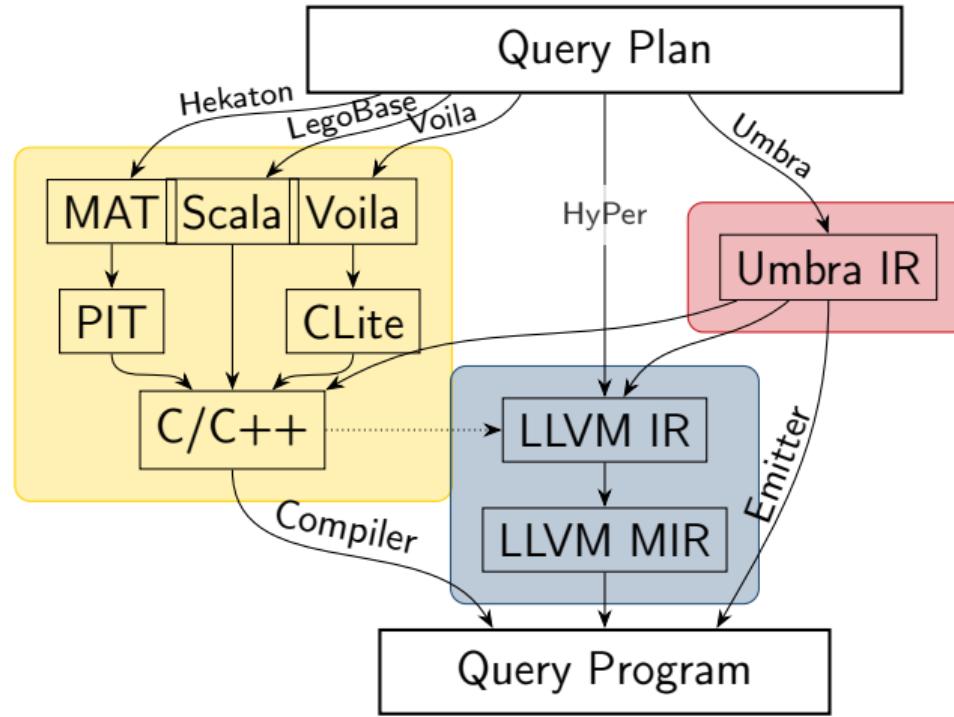
## How to Generate Code

- ▶ Code generator executes produce/consume methods
  - ▶ Method bodies don't do actual operations, but construct code
  - ▶ E.g., call `IRBuilder`
  - ▶ Call to helper functions for complex operations
    - ▶ e.g. hash table insert/lookup, string operations, memory allocation, etc.
- ▶ Resulting code doesn't contain produce/consume methods
  - ▶ only loops that iterate over data
  - ▶ No overhead of function calls
- ▶ Generate (at most) one function per pipeline
  - ▶ Allows for parallel execution of different pipelines

## What to Generate

- ▶ Code generation allows for substantial performance increase
  - ▶ *Fairly* popular, even in commercial systems, despite engineering effort
  - ▶ Competence in compiler engineering is a problem, though
- ▶ Bytecode
  - ▶ Extremely popular: fairly simple, portable, and flexible
- ▶ Machine code through programming language (C, C++, Scala, . . . )
  - ▶ Also popular: no compiler knowledge required, but compile-times are bad
- ▶ Machine code through compiler IR (mostly LLVM)
- ▶ Machine code through specialized IR (Umbra only)

# What to Generate

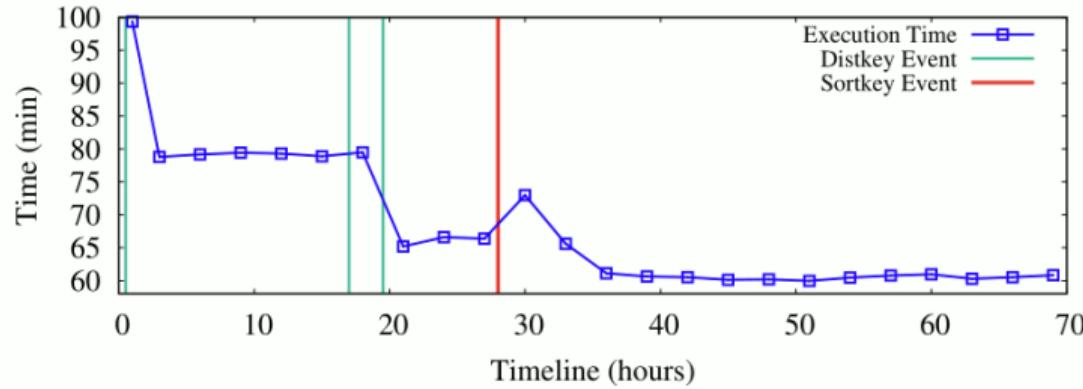


## Case Study: Amazon Redshift<sup>101</sup>

“Redshift generates C++ code specific to the query plan and the schema being executed. The generated code is then compiled and the binary is shipped to the compute nodes for execution [12, 15, 17]. Each compiled file, called a segment, consists of a pipeline of operators, called steps. Each segment (and each step within it) is part of the physical query plan. Only the last step of a segment can break the pipeline.”

<sup>101</sup> N Armenatzoglou et al. “Amazon Redshift Re-invented”. In: *SIGMOD*. 2022.

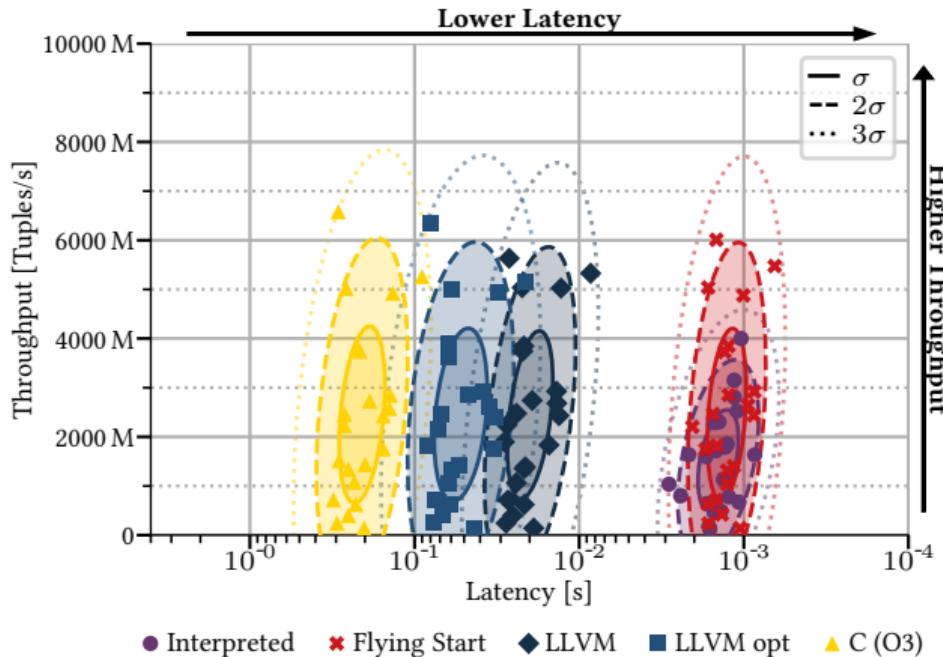
## Case Study: Amazon Redshift<sup>102</sup>



"Figure 7(a) illustrates [...] from an out-of-the-box TPC-H 30TB dataset [...]. The TPC-H benchmark workload runs on this instance every 30 minutes and we measure the end-to-end runtime. Over time, more and more optimizations are automatically applied reducing the total workload runtime. After all recommendations have been applied, the workload runtime is reduced by 23% (excluding the first execution that is higher due to compilation).

<sup>102</sup> N Armenatzoglou et al. "Amazon Redshift Re-invented". In: *SIGMOD*. 2022.

# Compile Times: Umbra



TPC-H sf=30, AMD Epyc 7713 (64 Cores, 1TB RAM)

# Vectorized Execution

- ▶ Problem: still only process single tuple at a time
- ▶ Doesn't utilize vector extensions of CPUs
- ▶ Idea: process multiple tuples at once
  - ▶ Also allows eliminating data-dependent branches, which not well-predictable
  - ▶ Esp. relevant when selectivity is between 10–90%
- ▶ Use of SIMD instructions requires column-wise store
  - ▶ Row-wise store would require gather operation for each load
  - ▶ Gather is very expensive

## Vectorized Execution: SIMD Instructions

- ▶ Obvious candidate: initial selection over tables
  - ▶ Load vector of elements, use SIMD operations for comparison
  - ▶ Write back compressed result to temporary location for use in subsequent operations
  - ▶ Special compress instructions (AVX-512, SVE) highly beneficial
- ▶ Other operations much more difficult to vectorize
  - ▶ Initial hash table lookup requires gather; collisions difficult
  - ▶ When many elements are masked out, performance suffers

## Vectorized Execution

- ▶ Bytecode interpretation substantially benefits from vectorized execution
- ▶ Key benefit: less dispatch overhead
- ▶ Typically much larger “vectors” (>1000)
- ▶ Comparison with non-vectorized machine code generation:
  - ▶ Vectorization often beneficial for initial scan
  - ▶ Code generation is faster than bytecode-interpreted vec. execution
  - ▶ But: a good vectorized engine is not necessarily *slow*
- ▶ Vectorized execution probably more popular than code generation

## Query Compilation – Summary

- ▶ Databases have trade-off between low latency and high throughput
- ▶ Evaluation needed for operators and subscripts
- ▶ Subscripts easy to compile
- ▶ Operator execution: full materialization vs. pipelined execution
- ▶ Pull-based vs. push-based execution
- ▶ Push-based allows for good code generation
- ▶ Bytecode and programming languages are widely used in practice
- ▶ Vectorized execution improves performance without native code gen.

## Query Compilation – Questions

- ▶ Why are low compile times important for databases?
- ▶ What is the difference between push-based and pull-based execution?
- ▶ Why does push-based execution allow for higher performance?
- ▶ How to generate code for a query?
- ▶ How does vectorized execution improve performance?
- ▶ Why do many database engines not use machine code generation?