

Code Generation for Data Processing

Alexis Engelke

Winter 2024/25

Contents

1. Introduction and Interpretation	1
1.1. Organization	1
1.2. Overview	3
1.3. High-Level Structure of Compilers	5
1.4. Interpretation	7
1.5. Context of Compilation	9
2. Compiler Front-end	13
2.1. Lexing	13
2.2. Parsing	15
2.3. Semantic Analysis	22
2.4. Miscellaneous	23
3. Intermediate Representations	27
4. LLVM-IR	41
4.1. Overview	41
4.2. LLVM-IR	42
4.3. API	47
4.4. IR Implementation	49
4.5. IR Design	51
5. Analyses and Transformations	55
5.1. Motivation	55
5.2. Dead Code Elimination	55
5.3. Dominator Tree	59
5.4. Common Subexpression Elimination	60
5.5. Simple Transformations	61
5.6. Loop Analysis	62
5.7. LLVM Passes	64
6. Vectorization	67
7. Instruction Selection	79
7.1. Overview	79
7.2. Macro Expansion	80
7.3. Tree Covering	82
7.4. DAG Covering	85
7.5. Graph Covering	87

7.6. ISel in Practice	88
7.7. LLVM Instruction Selection	88
8. Register Allocation	95
8.1. Avoiding Register Allocation	95
8.2. Handling PHI Nodes	96
8.3. Better Register Allocation	101
8.4. Generating Assembly	107
8.5. Summary	108
9. Object Files, Linker, and Loader	109
9.1. Object Files	110
9.2. Executable Files	116
9.3. Linker Optimizations	117
9.4. Static Libraries	119
9.5. Shared Libraries	119
10. Unwinding and Debuginfo	123
10.1. Stack Unwinding	123
10.2. Call Frame Information	125
10.3. Debug Information	130
11. JIT Compilation and Sandboxing	135
11.1. JIT Compilation	135
11.2. Sandboxing	140
12. Binary Translation	145
12.1. Overview	145
12.2. Binary Translation	145
12.3. Guest State	148
12.4. Fast DBT	148
12.5. Correct DBT	150
13. Query Compilation	155
13.1. Overview	155
13.2. Subscripts	156
13.3. Query Execution	156
A. Exercise Solutions	165

1. Introduction and Interpretation

1.1. Organization

[Slide 2] 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

[Slide 3] 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

[Slide 4] Lecture Organization

- Lecturer: Dr. Alexis Engelke engelke@in.tum.de
- Time slot: Thu 10-14, 02.11.018
- Material: <https://db.in.tum.de/teaching/ws2425/codegen/>

Exam

- Written exam, 90 minutes, **no retake**, date TBD
- (Might change to oral on very low registration count)

[Slide 5] Exercises

- Regular homework, often with programming exercise
- Submission via POST request (see assignments)
 - Grading with $\{*, +, \sim, -\}$, feedback on best effort
- Exercise session modes:
 - Present and discuss homework solutions
 - Hands-on programming or analysis of systems (needs laptop)

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

[Slide 6] 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!

[Slide 7] Compiler Lectures @ TUM

Compiler Construction IN2227, SS, THEO	Program Optimization IN2053, WS, THEO	Virtual Machines IN2040, SS, THEO
Front-end, parsing, semantic analyses, types	Analyses, transformations, abstract interpretation	Mapping programming paradigms to IR/bytecode

Programming Languages CIT3230000, WS	Code Generation CIT3230001, WS
Implementation of advanced language features	Back-end, machine code generation, JIT comp.

[Slide 8] 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

[Slide 9] Proebsting's Law

“Compiler advances double computing power every 18 years.”

– Todd Proebsting, 1998^a

^a<http://proebsting.cs.arizona.edu/law.html>

- Still optimistic; depends on number of abstractions

The performance increases compilers can make on existing code are typically low. However, optimizing compilers gain more abilities in simplifying needlessly complex code, enabling the use of more abstractions and therefore higher level code. These abstractions are removed/optimized during compilation, enabling languages to promote these as *zero-cost abstractions*. They do, however, have a cost: compile times.

Also note that some of these “zero-cost” abstractions actually *do* have some run-time cost. For example, the mere possibility of C++ exceptions can cause less efficient machine code and might prevents optimizations due to the more complex control flow possibilities.

1.2. Overview

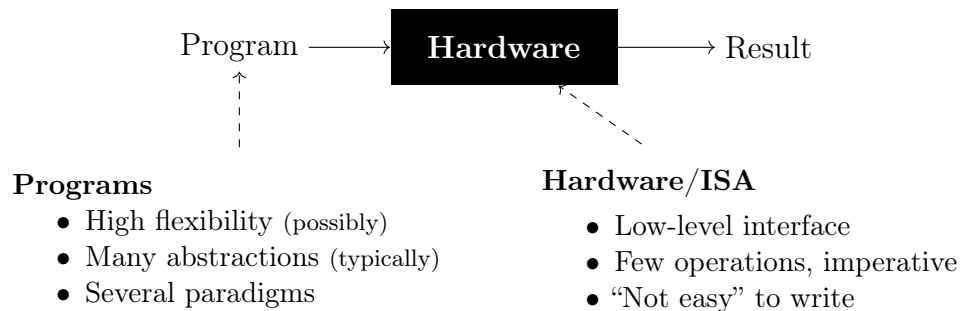
[Slide 10] Motivational Example: Brainfuck

- 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? ☹

[Slide 11] Program Execution



[Slide 12] 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;
  }
```

[Slide 13] Program Execution

Compiler



- Translate program to other lang.
- Might optimize/improve program

- C, C++, Rust → machine code
- Python, Java → bytecode

Multiple compilation steps can precede the “final interpretation”

Interpreter



- Directly execute program
- Computes program result

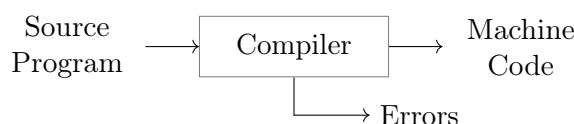
- Shell scripts, Python bytecode, machine code (conceptually)

1.3. High-Level Structure of Compilers

[Slide 14] 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 \rightarrow machine code
 - JIT compilation of JavaScript, WebAssembly, Java bytecode, ...
 - Database query compilation
 - ISA emulation/binary translation

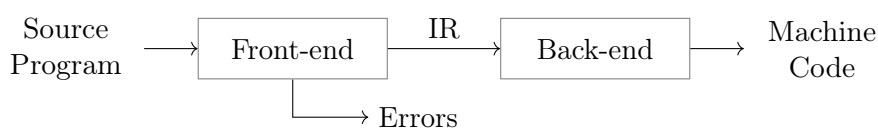
[Slide 15] Compiler Structure: Monolithic



- Inflexible architecture, hard to retarget

Some languages like C are designed to be compilable in a single pass without building any intermediate representation of the code between source and assembly. Single-pass compilers exist, but often have very limited possibilities to transform the code. They might not even know basic code properties, e.g., the size of the stack frame, during compilation of a function.

[Slide 16] 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

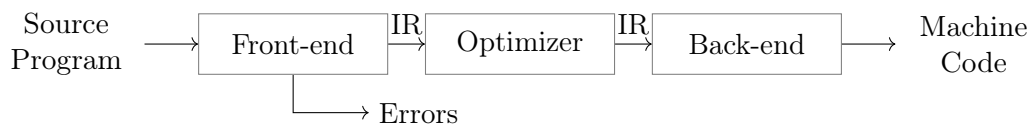
- Translate IR to target architecture
- Can assume valid IR (\leadsto no errors)
- Possibly one back-end per arch.

- Contains \mathcal{NP} -complete problems

After parsing, all information is encoded in the IR, including references to source code constructs for debugging support. The input source code is (at least conceptually) no longer needed.

In practice, there are very rare cases where the back-end can also raise errors. This can happen, for example, when some very architecture-specific constraints might be hard to verify during parsing (e.g., inline assembly constraints in combination with available registers).

[Slide 17] 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

Example Clang/LLVM (will be covered in more detail later): Clang parses the input into an abstract syntax tree (IR 1), uses this for semantic analyses; then Clang transforms the code into LLVM-IR (IR 2), which is primarily used for optimization; then the LLVM back-end transforms the code further into LLVM's Machine IR (IR 3), executes some low-level optimizations and register allocation there; the assembly printer of the back-end then lowers the code further to LLVM's machine code representation (IR 4), before finally emitting machine code. Some optimizations inside this pipeline, e.g. vectorization, might even build further representation of the code.

Why are compilers using so many different code representations? Different transformations work best at different abstraction levels. Diagnosing unused variables, for example, requires information about the source code. Optimization of arithmetic computations is easier in a data-flow-focused representation, where no explicit variables exist. Low-level modifications, like folding operations into complex addressing modes of the ISA, need a code representation where ISA instructions are already present.

[Slide 18] Compiler Front-end

1. Tokenizer: recognize words, numbers, operators, etc. *Re*
 - Example: $a+b*c \rightarrow \text{ID}(a) \text{ PLUS ID}(b) \text{ TIMES ID}(c)$
2. Parser: build (abstract) syntax tree, check for syntax errors *CFG*

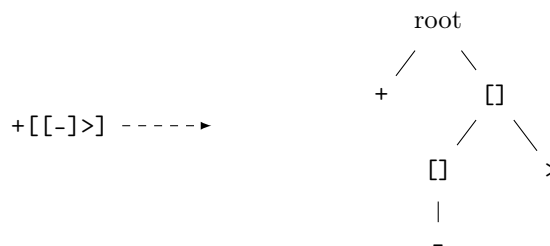
- Syntax Tree: describe grammatical structure of complete program Example: `expr("a", op("+"), expr("b", op("*"), expr("c")))`
 - Abstract Syntax Tree: only relevant information, more concise Example: `plus("a", times("b", "c"))`
3. Semantic Analysis: check types, variable existence, etc.
 4. IR Generator: produce IR for next stage
 - This might be the AST itself

[Slide 19] 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

1.4. Interpretation**[Slide 20] 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

**[Slide 21] 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);
}
```

[Slide 22] Motivational Example: Brainfuck – Optimization

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

In-Class Exercise:

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

[Slide 23] Motivational Example: Brainfuck – Optimization

- Fold offset into operation
 - `right(2) add(1) = addoff(2, 1) 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.

[Slide 24] 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

Dispatching multiple instructions at once can be problematic due to the explosion of cases that need to be implemented (often results in large jump tables and lots of code with resulting cache misses and branch mispredictions). Often, it is advisable to not always switch over multiple neighbored instructions, but instead combine common sequences into superinstructions.

[Slide 25] Motivational Example: Brainfuck – Threaded Interpretation

- 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;
}
```

With threaded interpretation there is not a single indirect jump instruction inside the dispatcher, but one indirect jump instruction per operation. Each of these indirect jumps then occupies a different branch prediction slot in the CPU. If an operation of type X is typically followed by an operation of type Y, with threaded interpretation the CPU has a much better chance of correctly predicting the dispatch branch to the next operation, because the indirect jump at the end of operation X typically jumps to operation Y. Without threaded interpretation, there would be only a single indirect branch, which is much harder to predict.

Threaded interpretation is especially useful on older and less powerful CPUs. Recent CPUs (e.g., Intel since Skylake, AMD since Zen 3, Apple Silicon) store the history of indirect branches and use this for better prediction. On such processors, threaded interpretation might not improve performance (or gains might be lower).

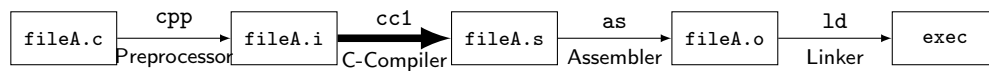
[Slide 26] Fast Interpretation

- 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
- 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

1.5. Context of Compilation

[Slide 27] 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.)

[Slide 28] 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

[Slide 29] Motivational Example: Brainfuck – Runtime Environment

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

In-Class Exercise:

How to efficiently emulate an infinitely sized array?

[Slide 30] 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

[Slide 31] Introduction and Interpretation – Summary

- Compilation vs. interpretation and combinations

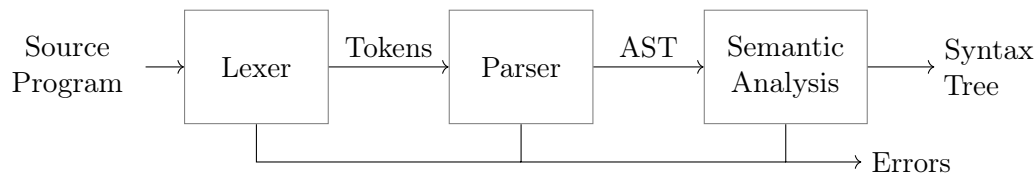
- Compilers are key to usable/performant languages
- Target language typically machine code or bytecode
- Three-phase architecture widely used
- Interpretation techniques: bytecode, threaded interpretation, ...
- JIT compilation imposes different constraints

[Slide 32] 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 key differences between AoT and JIT compilation?

2. Compiler Front-end

[Slide 34] 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

2.1. Lexing

[Slide 35] Lexer

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

`a+b*c` → `ID(a) PLUS ID(b) TIMES ID(c)`

- Typically representable as regular expressions

[Slide 36] 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'œ'`
- String literals `"abc\x12\n"`
- Internal `EOF COMMENT UNKNOWN INDENT DEDENT`
 - Comments might be useful for annotations, e.g. `// fallthrough`

Indentation-based languages like Python need separate tokens for indent/dedent, the indentation level is tracked in the lexer. Parsing numbers may need special care to correctly handle all possible cases of integer and floating-point numbers.

[Slide 37] 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};
    }
}
```

This is just a minimal and non-optimized implementation to illustrate the concept. Performance-focused implementations do not use explicit regular expressions but write the state machine into code.

The struct `Token` has room for improvement. First, a `string_view` is unnecessarily large with 16 bytes, most tokens are smaller than 2^{16} bytes. Some tracking of the source locations is advisable for attaching diagnostics to their origin inside the code, for example by storing a file ID and the byte offset into the file. By tracking the byte offsets of line breaks, the line number can be reconstructed in $\mathcal{O}(\log n)$ from the byte offset.

Another optimization strategy is string interning, where identifiers are converted into unique integers (or pointers) during parsing. During later phases, comparing interned strings is much more efficient, as it is just an integer/pointer comparison. Another benefit is that the entire input file does not need to be kept in memory during parsing.

[Slide 38] Lexing C??=

```
main() <%
    // yay, this is C99??/
    puts("hi_world!");
    puts("what's_up??!");
%>
```

Output: what's up|

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

Besides digraphs, trigraphs, and the preprocessor, C has another weird property: identifier names can be split by `\`, which concatenates two lines. It is necessary to construct the “real” identifier first. To simplify memory management in such cases, a bump pointer allocator (allocate large chunks of memory from the OS, then simply bump the end pointer for every allocation) can be useful to store such constructed names.

[Slide 39] Lexer Implementation

- Essentially a DFA (for most languages)
 - Set of regexes \rightarrow NFA \rightarrow 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

A tool to generate perfect hash tables from a set of keywords is gperf. Example, compile with `gperf -L C++ -C -E -t <input>`:

```
struct keyword {char* name; int val; }
%%
int, 1
char, 2
void, 3
if, 4
else, 5
while, 6
return, 7
```

2.2. Parsing

[Slide 40] Parsing

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

[Slide 41] 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
```

[Slide 42] 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!
}
```

[Slide 43] 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 \Rightarrow 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});
}

```

[Slide 44] Ambiguity

$$expr \rightarrow expr + expr \mid expr * expr \mid expr = expr \mid (expr) \mid \text{number}$$
Input: $4 + 3 * 2$ 

The grammar, as specified, is ambiguous, there are two possible ways to parse the input.

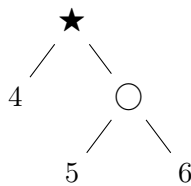
[Slide 45] Ambiguity – Rewrite Grammar?

$$primary \rightarrow (expr) \mid \text{number}$$

$$expr \rightarrow primary + expr \mid primary * expr \mid primary = expr \mid primary$$
Input: $4 + 3 * 2$ Input: $4 * 3 + 2$ 

The grammar is no longer ambiguous, but the result might not be expected, conventionally, multiplication has a stronger binding than addition.

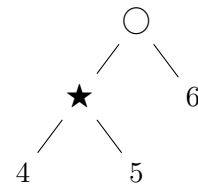
[Slide 46] Ambiguity – PrecedenceInput: $4 \star 5 \bigcirc 6$



If $prec(○) > prec(★)$ or equal prec.
and $★$ is right-associoc.

Examples:

- $4 + 5 \cdot 6$ ($prec(\cdot) > prec(+)$)
- $a = b = c$ ($=$ is right-associoc.)
 $b = c$ should be executed first



If $prec(○) < prec(★)$ or equal prec.
and $★$ is left-associoc.

Examples:

- $4 + 5 < 6$ ($prec(<) < prec(+)$)
- $a + b - c$ ($+$ is left-associoc.)
 $a + b$ should be executed first

[Slide 47] 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.
 - $- < \text{unary} - < [] / ->$

```
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: // ...
            // ...
    }
}
```

[Slide 48] Hand-written Parsing – Expression Parsing

- Only allow ops. with higher prec. on the right child
 - Right-associoc.: 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;
}
```

In-Class Exercise:

`a = 3 * 2 + 1;` `a = b + c + d = 1;` `a ? 1 : b ? 2 : 3;`

Example for input: `a = 3 * 2 + 1;`

	Rec. Depth 1	Rec. Depth 2	Rec. Depth 3
minPrec	1		
lhs	a		
op (prec/assoc)	= (2/r)		
minPrec	1	2	
lhs	a	3	
op (prec/assoc)	= (2/r)	* (12/l)	
minPrec	1	2	13
lhs	a	3	2
op (prec/assoc)	= (2/r)	* (12/l)	+ (11/l)
minPrec	1	2	
lhs	a	3*2	
op (prec/assoc)	= (2/r)	+ (11/l)	
minPrec	1	2	12
lhs	a	3*2	1
op (prec/assoc)	= (2/r)	+ (11/l)	; (0/-)
minPrec	1	2	
lhs	a	(3*2)+1	
op (prec/assoc)	= (2/r)	; (0/-)	
minPrec	1		
lhs	a=((3*2)+1)		
op (prec/assoc)	; (0/-)		

[Slide 49] 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

[Slide 50] 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

[Slide 51] 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 '*'
```

[Slide 52] 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/... */ }
```

Compile with `bison -dg input.ypp`, it will emit a C++ header, the implementation file, and also a graph showing the state machine of the parser.

[Slide 53] 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

In fact, many compilers^a use hand-written parsers, because they allow for better error messages a more graceful handling of syntax errors, leading to more reported errors during a single (failing) compilation.

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

[Slide 54] 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-complete¹
- C++ *parsing* is hence *undecidable*²
 - Template instantiation combined with `C T * a` ambiguity

2.3. Semantic Analysis

[Slide 55] Semantic Analysis

- Syntactical correctness \nrightarrow 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, ...

¹TL Veldhuizen. *C++ templates are Turing complete*. 2003. URL: <http://port70.net/~nsz/c/c%2B%2B/turing.pdf>.

²J Haberman. *Parsing C++ is literally undecidable*. 2013. URL: <https://blog.reverberate.org/2013/08/parsing-c-is-literally-undecidable.html>.

[Slide 56] 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

In-Class Exercise:

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

There are two ways of implementing a scoped hash table:

- Chain of hash maps: $Scope = (Map[Name \rightarrow Type] \text{ names}, Scope \text{ parent})$. This is, however, very slow for deeply nested scopes, as all hash maps of the parent scopes must be queried. Hash map lookups are fairly expensive.
- Hash map of lists: $Map[Name \rightarrow List[Tuple[Depth, Type]]]$. For every identifier, the type at a given scope nesting depth is stored. Invalidation can be implemented with an epoch counter for every depth. The downside is that this hash map can grow very large, as entries are never removed.

[Slide 57] 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

2.4. Miscellaneous

[Slide 58] 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 noticable
- Somewhat more: Developer tools
 - Imagine: waiting for seconds just for updated syntax highlighting
 - Often uses tricks like incremental updates to parse tree

[Slide 59] 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

[Slide 60] 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)

[Slide 61] Recommended Lectures

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

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

[Slide 62] 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

[Slide 63] Compiler Front-end – Questions

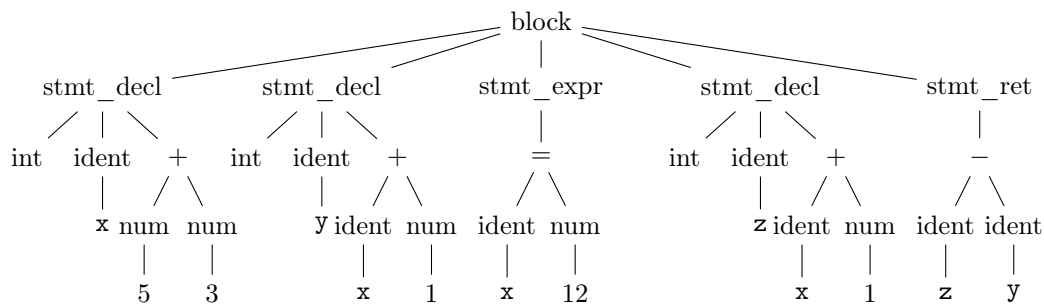
- What are typical components of a compiler front-end?
- What output does the lexer produce?
- How does a parser disambiguate rules?
- What is the typical way to handle operator precedence?
- Why are not all programming languages describable using CFGs?
- How to implement classes with virtual functions?

3. Intermediate Representations

[Slide 65] 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

[Slide 66] Intermediate Representations: Motivation



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

In this representation, it is very easy to see that the two $+1$ operations have different operands on the left side and are therefore not trivially redundant.

[Slide 67] 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! 😊

[Slide 68] Intermediate Representations

- Definitive program representation inside compiler
 - During compilation, only the (current) IR is considered

In practice, there are, of course, exceptions to the general rule; sometimes an IR contains references to a previous/higher-level IR. An example is LLVM's low-level Machine IR, which only represents single functions and therefore references to global variables use the higher-level LLVM IR.

- 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)

[Slide 69] 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
```

Tracking state, especially when memory is involved, is one of the main challenges during optimization. In the first example, the loop is not easily vectorizable, because `a` and `b` could point to the same underlying array (e.g., with `addvec(buf + 1, buf)`).

[Slide 70] Compiler Design: Effect of Languages – Declarative

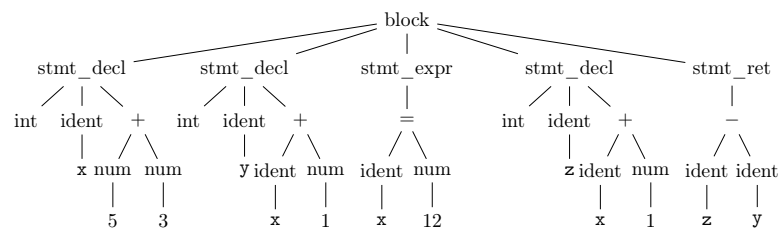
- Describes execution target
- Compiler has to derive good mapping to imperative hardware
- Allows for more optimizations
- Mapping to hardware non-trivial

- 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)
```

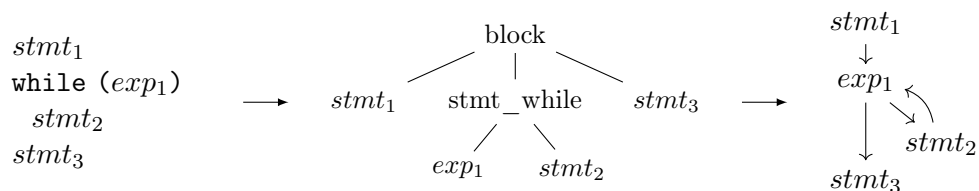
[Slide 71] Graph IRs: Abstract Syntax Tree (AST)

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



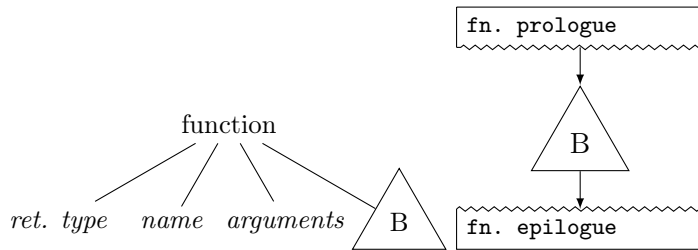
[Slide 72] Graph IRs: Control Flow Graph (CFG)

- 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

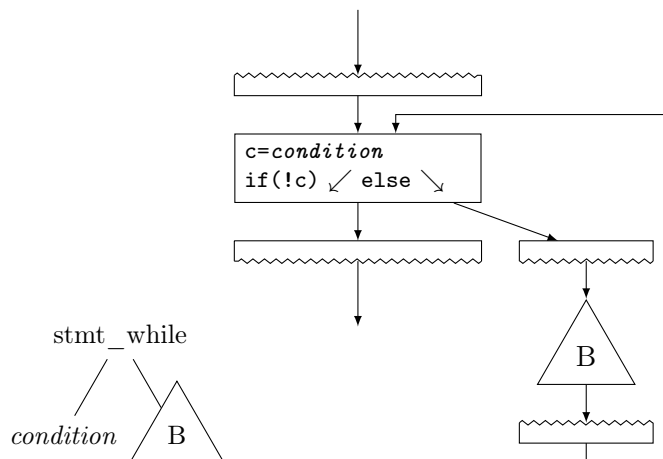


[Slide 73] Build CFG from AST – Function

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



[Slide 74] Build CFG from AST – While Loop



Written in pseudo-code:

```

IRValue generateCFG(ASTNode* node, BasicBlock*& insPos) {
    switch (node->kind()) {
    case ASTNode::Function:
        insPos = generatePrologue(node);
        generateCFG(node->child(0), insPos);
        generateEpilogue(insPos);
        return nullptr;
    case ASTNode::Block:
        for (ASTNode* child : node->children())
            generateCFG(child, insPos);
        return nullptr;
    case ASTNode::While: {
        BasicBlock* cond = newBlock();
        BasicBlock* body = newBlock();
        BasicBlock* end = newBlock();
        branchTo(insPos, cond);
        insPos = cond;
        IRValue brcond = generateCFG(node->child(0), insPos);
        // NB: generateCFG can modify insPos
        branchToCond(insPos, brcond, body, end);
        insPos = body;
        generateCFG(node->child(1), insPos);
        branchTo(insPos, cond);
        insPos = end;
        return nullptr;
    }
    }
}

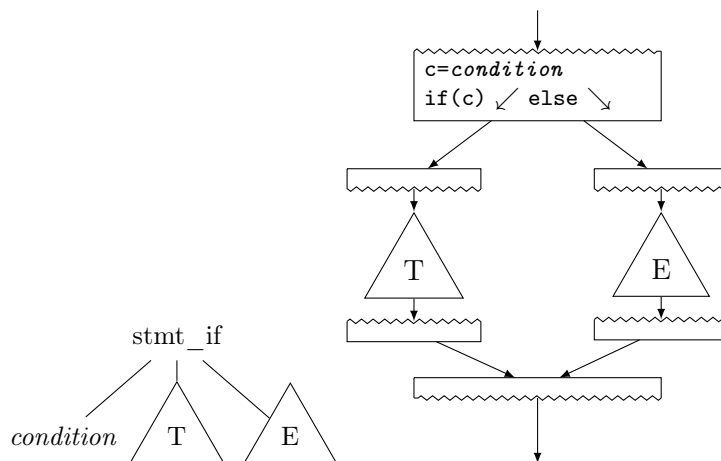
```

```

}
// ...
}
}

```

[Slide 75] Build CFG from AST – If Condition



[Slide 76] 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_D, B_D, B_D, B₃,
B₄, B_D, ... }

+ Fastest
– Table can be large,
needs ind. jump

[Slide 77] Build CFG from AST: Break, Continue, 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)

[Slide 78] 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

[Slide 79] CFG from C – Example

In-Class Exercise:

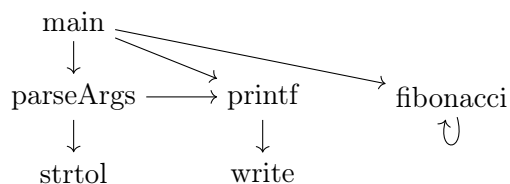
Derive the CFG for the these functions. Assume a `switch` instruction exists.

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

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

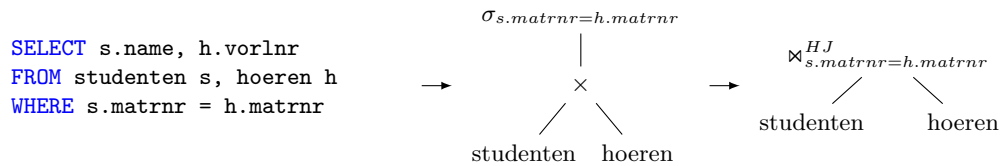
[Slide 80] Graph IRs: Call Graph

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



[Slide 81] 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.



[Slide 82] 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
  
```

[Slide 83] Linear IRs: Register Machines

- Operands stored in registers
- Operations read and write registers
- Typically: infinite number of registers
- Typically: three-address form

– $dst = src1 \text{ op } src2$

- 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
  
```

[Slide 84] 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>
>

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

[Slide 85] Example: Low GIMPLE

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

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

    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
```

[Slide 86] Example: Low GIMPLE with CFG

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

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>
}

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

[Slide 87] 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	\leftarrow	5	+	3
y	\leftarrow	x	+	1
x	\leftarrow	12		
z	\leftarrow	x	+	1
tmp_1	\leftarrow	z	-	y
return		tmp_1		

[Slide 88] 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

$ \begin{array}{lcl} x & \leftarrow & 5 + 3 \\ y & \leftarrow & x + 1 \\ x & \leftarrow & 12 \\ z & \leftarrow & x + 1 \\ tmp_1 & \leftarrow & z - y \\ \text{return} & & tmp_1 \end{array} $	\longrightarrow	$ \begin{array}{lcl} v_1 & \leftarrow & 5 + 3 \\ v_2 & \leftarrow & v_1 + 1 \\ v_3 & \leftarrow & 12 \\ v_4 & \leftarrow & v_3 + 1 \\ v_5 & \leftarrow & v_4 - v_2 \\ \text{return} & & v_5 \end{array} $
--	-------------------	--

[Slide 89] Single Static Assignment: Control Flow

- How to handle diverging values in control flow?
- Solution: Φ -nodes to merge values depending on predecessor
 - Value depends on edge used to enter the block
 - All Φ -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 ← Φ(entry: v1, then: v2)
      return v3

```

[Slide 90] Example: GIMPLE in SSA form

```

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

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

```

[Slide 91] SSA Construction – Local Value Numbering

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

Code

```

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

```

SSA IR

```

v1 ← add 5, 3
v2 ← add v1, 1
v3 ← const 12
v4 ← add v3, 1
v5 ← sub v4, v2
    ret v5

```

Variable Mapping

```

x  → v3
y  → v2
z  → v4
tmp1 → v5

```

[Slide 92] SSA Construction – Across Blocks

- SSA construction with control flow is non-trivial
- Key problem: find value for variable in predecessor
- Naive approach: Φ -nodes for all variables everywhere
 - Create empty Φ -nodes for variables, populate variable mapping
 - Fill blocks (as on last slide)
 - Fill Φ -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 Φ*

[Slide 93] SSA Construction – Across Blocks (“simple”¹)

- 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 Φ -node, fill on sealing
- Single predecessor: recursively query that
- Multiple preds.: add Φ -node, fill now

Confer the (very readable) paper for a more formal specification of the algorithm.
The removal of trivial and redundant Φ -nodes is not strictly required.

[Slide 94] SSA Construction – Example

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



```
func foo(v1)
entry: sealed; varmap: n → v1, res → v2
      v2 ← 1
header: sealed; varmap: n →  $\phi_1$ , res →  $\phi_2$ 
       $\phi_1 \leftarrow \phi(\text{entry: } v_1, \text{body: } v_6)$ 
       $\phi_2 \leftarrow \phi(\text{entry: } v_2, \text{body: } v_5)$ 
      v3 ← equal  $\phi_1$ , 0
      br v3, cont, body
body:  sealed; varmap: n → v6, res → v5
      v4 ← mul  $\phi_1$ ,  $\phi_1$ 
      v5 ← mul  $\phi_2$ , v4
      v6 ← sub  $\phi_1$ , 1
      br header
cont:  sealed; varmap: res →  $\phi_2$ 
      ret  $\phi_2$ 
```

[Slide 95] SSA Construction – Example

¹M Braun et al. “Simple and efficient construction of static single assignment form”. In: *CC*. 2013, pp. 102–122. URL: https://link.springer.com/content/pdf/10.1007/978-3-642-37051-9_6.pdf.

In-Class Exercise:

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

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

[Slide 96] SSA Construction – Pruned/Minimal Form

- Resulting SSA is *pruned* – all ϕ are used
- But not *minimal* – ϕ nodes might have single, unique value
- When filling ϕ , check that multiple real values exist
 - Otherwise: replace ϕ with the single value
 - On replacement, update all ϕ using this value, they might be trivial now, too
- Sufficient? Not for irreducible CFG
 - Needs more complex algorithms² or different construction method³

AD IN2053 “Program Optimization” covers this more formally

[Slide 97] SSA: Implementation

- Value is often just a pointer to instruction
- ϕ nodes placed at beginning of block
 - They execute “concurrently” and on the edges, after all
- Variable number of operands required for ϕ 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

²M Braun et al. “Simple and efficient construction of static single assignment form”. In: *CC*. 2013, pp. 102–122. URL: https://link.springer.com/content/pdf/10.1007/978-3-642-37051-9_6.pdf.

³R Cytron et al. “Efficiently computing static single assignment form and the control dependence graph”. In: *TOPLAS* 13.4 (1991), pp. 451–490. URL: <https://dl.acm.org/doi/pdf/10.1145/115372.115320>.

[Slide 99] 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

[Slide 100] 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 ϕ -instructions differ from normal instructions?

4. LLVM-IR

4.1. Overview

[Slide 102] LLVM¹

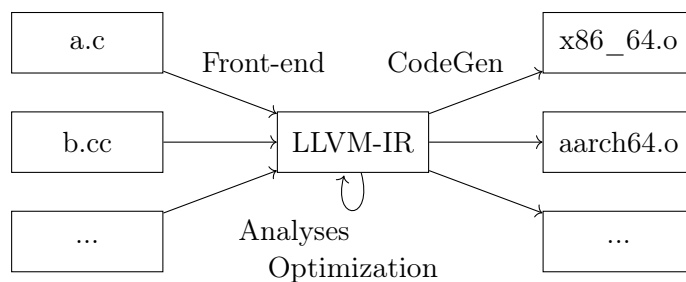
LLVM “Core” Library

- 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

[Slide 103] 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

¹C Lattner and V Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *CGO*. 2004, pp. 75–86. URL: <http://www.llvm.org/pubs/2004-01-30-CGO-LLVM.pdf>.

The single IR allows multiple front-ends to reuse the same back-end infrastructure. Thus, generating LLVM-IR provides an easy way to target a wide range of architectures.

For link-time optimization, the LLVM-IR is stored in the object files instead of the machine code. At link-time, a linker plugin detects these files, merges the LLVM-IR from all object files, and then runs the actual compilation as part of the linking step. We will look at LTO again later when discussing object file generation and linking.

4.2. LLVM-IR

[Slide 104] 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  
}
```

[Slide 105] LLVM-IR: Data types

- First class types:
 - `i<N>` – arbitrary bit width integer, e.g. `i1`, `i25`, `i1942652`
 - `ptr/ptr addrspace(1)` – pointer with optional address space
 - `float/double/half/bfloat/fp128/...`
 - `<N x ty>` – vector type, e.g. `<4 x i32>`
- Aggregate types:
 - `[N x ty]` – constant-size array type, e.g. `[32 x float]`
 - `{ ty, ... }` – struct (can be packed/opaque), e.g. `{i32, float}`

- Other types:
 - `ty (ty, ...)` – function type, e.g. `{i32, i32} (ptr, ...)`
 - `void`
 - `label/token/metadata`

Although structure types can be used in various places in the IR, e.g., a single instruction to load a large structure from memory, this is strongly discouraged: LLVM is not optimized for this and both code quality and compile times get considerably worse. Only use struct types for globals and to implement multiple return values.

[Slide 106] 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)

[Slide 107] LLVM-IR: Functions

- Functions definitions contain all code, not nestable
- Single return type (or `void`), multiple parameters, list of basic blocks
 - No basic blocks \Rightarrow 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>)`

[Slide 108] LLVM-IR: Basic Block

- Sequence of instructions
 - ϕ 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

[Slide 109] 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`

Although `call` does modify control flow in some sense, the assumption is that every function call returns ordinarily. When special control flow for exceptions is needed, the `invoke` instruction is used, which specifies one basic block as successor for the ordinary case and one basic block for the exceptional case.

[Slide 110] 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

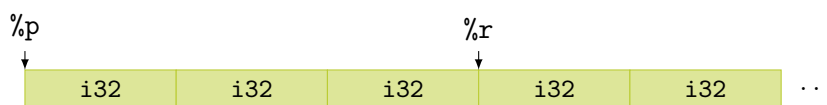
Technically, out-of-range shifts return `poison`, see below.

[Slide 111] 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

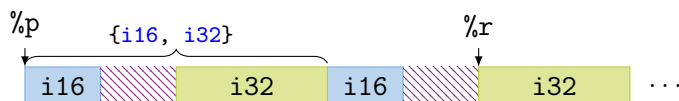
[Slide 112] LLVM-IR: `getelementptr` Examples

- `%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

[Slide 113] 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`

[Slide 114] 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, ...`

[Slide 115] 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

[Slide 116] LLVM-IR: Example

```
define dso_local <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  
}
```

[Slide 117] LLVM-IR: Example

```
define dso_local 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  
}
```

[Slide 118] LLVM-IR: Example

```
define dso_local 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  
}
```

[Slide 119] LLVM-IR: Example

In-Class Exercise:

```
@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
}
```

4.3. API

[Slide 120] LLVM-IR API

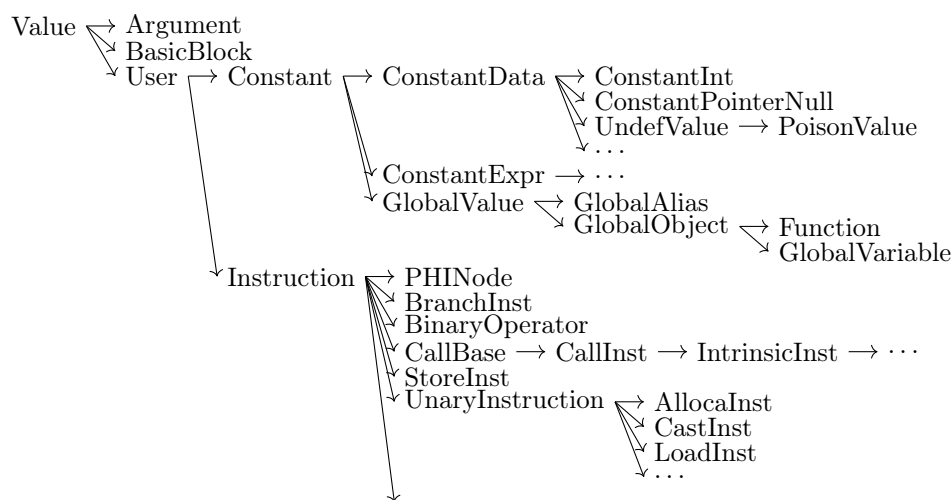
- LLVM offers two APIs: C++ and C
 - C++ is the full API, exposing nearly all internals
 - C API is more limited, but more stable
- Nearly all major versions have breaking changes
- Some support for multi-threading:
 - All modules/types/... associated with an LLVMContext
 - Different contexts may be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

[Slide 121] 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;
}
```

[Slide 122] LLVM-IR API: Almost Everything is a Value... (excerpt)

See LLVM Doxygen^a for a full graph.

^ahttps://llvm.org/doxygen/classllvm_1_1Value.html

[Slide 123] 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 manual²
- Preferably uses `ArrayRef`, `StringRef`, `Twine` for references
- LLVM implements custom streams instead of std streams
 - `outs()`, `errs()`, `dbgs()`

Many of these data types are used for efficiency. Standard C++ RTTI is inefficient while LLVM's implementation is very flexible, fast, and has a low memory usage in data structures.

`SmallVector` is preferred over `std::vector` not just because of the inline storage, but also because (for non-`char` types) it only uses 32-bit integers for length/capacity (lower memory usage, often sufficient) and grows more efficiently for trivially movable data structures.

`Twine` is a lazily evaluated string. For example, when specifying `Twine("foo") + 5`, on-stack data structures are constructed to represent this sequence, but the resulting string is constructed only when and if it is actually used. This also allows constructing strings directly into target buffers.

²<https://www.llvm.org/docs/ProgrammersManual.html>

Standard C++ streams are not just inefficient, implementations also tend to inject global constructors in all files. Therefore, LLVM has its own stream implementation. With `raw_svector_ostream` and `raw_string_ostream`, a `raw_ostream` can be used to write into a `SmallVector` or `std::string`.

[Slide 124] LLVM-IR API: Use Tracking

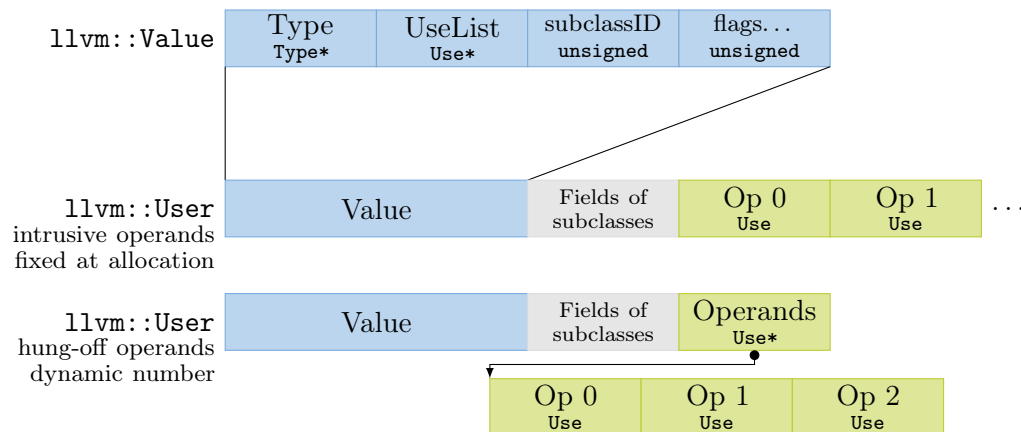
- Values track their users


```
llvm::Value* v = /* ... */;
for (llvm::User* u : v->users())
    if (auto i = llvm::dyn_cast<llvm::Instruction>(u))
        // ...
```
- Simplifies implementation of analyses
- Allows for easy replacement:


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

4.4. IR Implementation

[Slide 125] LLVM IR Implementation: Value/User

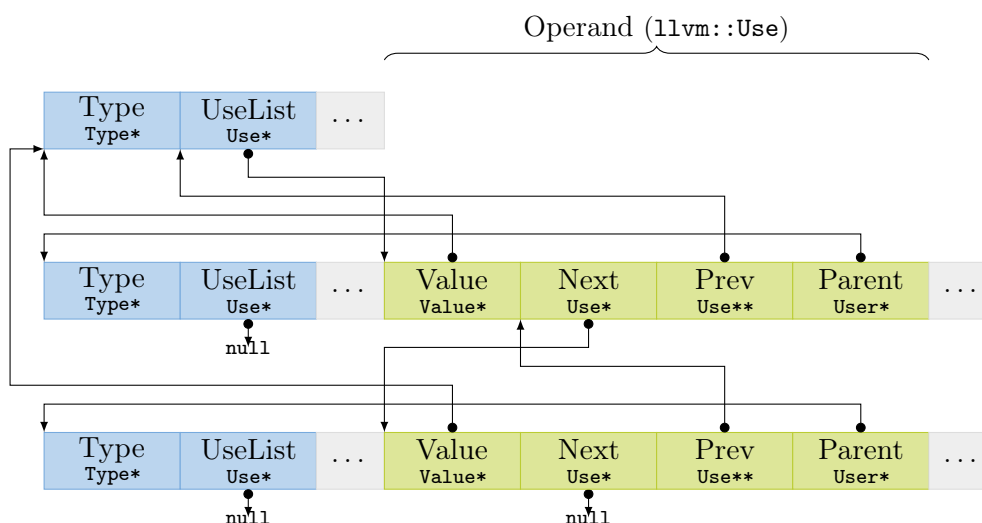


`PHINode` additionally stores n `BasicBlock*` after the operands, but aren't users of blocks.

Every LLVM `Instruction` is a separate heap allocation. As the number of operands is typically known when constructing the instruction, they are allocated after the instruction data structure (this is implemented by `User`).

It can happen that the number of operands increases beyond the allocated storage, for example, when a `PHINode` gets more operands than initially expected. In such cases, the operand list gets hung off into a separate allocation.

As a special case, `PHINode` needs to store the associated `BasicBlocks` in addition to the merged values. The blocks are stored after the operands, but are not operands themselves.

[Slide 126] LLVM IR Implementation: Use

The use list is a doubly-linked list. Starting from `Value::UseList`, one can find all used by following the `Use::Next` pointer. The `Use::Prev` pointer does not point to the previous `Use`, but the previous `Use::Next` pointer or the `Value::UseList` — this way, unlinking does not need to distinguish the special case of the beginning of the use list.

A `Use` also has a pointer to the actual `Value`, so that when inspecting an operand one can actually find the operand itself.

There is also a `Parent` pointer, which points to the `User` which owns the operand: when iterating over the use list, this is the only way to find out which instruction (`User`) uses the value.

In sum, an LLVM-IR operand is quite large, using 32 bytes on a 64-bit system. In addition to every instruction being a separate heap allocation and every operand update requires updating the use list (less data locality), the IR data structures are (in absolute terms) not very efficient — despite being fairly optimized for the use cases they serve.

[Slide 127] 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- ϕ requires iterating over ϕ -nodes

4.5. IR Design

[Slide 128] 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

[Slide 129] 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

[Slide 130] 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?

[Slide 131] 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)

[Slide 132] 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: ϕ nodes vs. block arguments?

[Slide 133] 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)
```

[Slide 134] 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
- IR design depends on purpose and integration constraints

[Slide 135] 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?
- How does LLVM's `replaceAllUsesWith` work? How could this work without building/maintaining user lists?
- How can an SSA-based IR make side effects explicit?
- How would you design an IR for optimizing Brainfuck?

5. Analyses and Transformations

5.1. Motivation

[Slide 137] 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-known¹

5.2. Dead Code Elimination

[Slide 138] 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

[Slide 139] 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
```

¹FE Allen and J Cocke. *A catalogue of optimizing transformations*. 1971. URL: <https://www.clear.rice.edu/comp512/Lectures/Papers/1971-allen-catalog.pdf>.

```
    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
}
```

[Slide 140] 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:
 - a) Check for deadness (zero users, no side-effects)
 - b) If dead, remove and add all operands to work queue

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

[Slide 141] 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
}
```

In this example, the instruction `%abssum` can be removed. This reduces the number of users of `%a` and `%b` by 1. As no other instructions have a user count of 0 after this change, the algorithm terminates.

[Slide 142] 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)
1. Only mark instrs. with side effects as live
 2. Populate work list with newly added live instrs.
 3. While work list not empty:
 - a) Mark dead operand instructions as live and add to work list
 4. Remove instructions not marked as live

[Slide 143] 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)

This algorithm finds the dead value cycle of %b from the previous example. (Refer to the slide deck for the animated version.)

[Slide 144] Liveness-based DCE: Work List Implementation**In-Class Exercise:**

- What operations are performed on a work list?
 - Insert instruction
 - Remove any instruction
 - Test whether instruction is contained
 - Get and remove next instruction to handle
- How to implement an efficient work list?

[Slide 145] Optimization Example 2

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

[Slide 146] 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?

[Slide 147] CSE Attempt 1 – Example 1

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

[Slide 148] 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!

5.3. Dominator Tree

[Slide 149] 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 : $DOM(n) = \{d \mid d \text{ dom } n\}$
- Strictly dominate: $d \text{ sdom } n \Leftrightarrow d \text{ dom } n \wedge d \neq n$
- Immediate dominator: $\text{idom}(n) = d : d \text{ sdom } n \wedge \nexists d'. d \text{ sdom } d' \wedge d' \text{ sdom } n$

\Rightarrow All strict dominators are always executed before the block

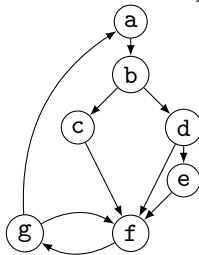
\Rightarrow All values from dominators available/usable

\Rightarrow All values not from dominators **not** usable

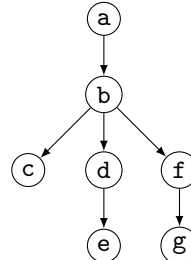
[Slide 150] Dominator Tree

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

Control Flow Graph



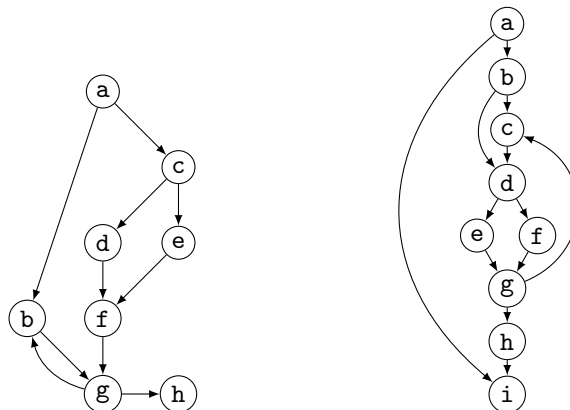
Dominator Tree



[Slide 151] Dominator Tree – Example

In-Class Exercise:

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



[Slide 152] Dominator Tree: Construction

- Naive: inefficient (but reasonably simple)²
 - 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 methods³
 - 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 compilers⁴
- Semi-NCA: $\mathcal{O}(n^2)$, but lower constant factors⁵

Most notable, LLVM doesn't use the Lengauer–Tarjan algorithm. Instead, they use the Semi-NCA algorithm, which has $\mathcal{O}(n^2)$ runtime, but lower constant factors and is therefore substantially faster for certain (typical) inputs^a.

^aJ Kuderski. “Dominator Trees and incremental updates that transcend times”. In: *LLVM Dev Meeting*. Oct. 2017. URL: https://llvm.org/devmtg/2017-10/slides/Kuderski-Dominator_Trees.pdf.

[Slide 153] 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 ⁶
 - $a.preNum < b.preNum \wedge a.postNum > b.postNum$
 - After updates, numbers might be invalid: recompute or walk tree
- Problem: dominance of unreachable blocks ill-defined \rightsquigarrow special handling

5.4. Common Subexpression Elimination

[Slide 154] CSE Attempt 2

- Option 1:
 - For identical instructions, store all
 - Add dominance check before replacing

²ES Lowry and CW Medlock. “Object code optimization”. In: *CACM* 12.1 (1969), pp. 13–22. URL: <https://dl.acm.org/doi/pdf/10.1145/362835.362838>.

³T Lengauer and RE Tarjan. “A fast algorithm for finding dominators in a flowgraph”. In: *TOPLAS* 1.1 (1979), pp. 121–141. URL: <https://dl.acm.org/doi/pdf/10.1145/357062.357071>

⁴Example: <https://github.com/WebKit/WebKit/blob/aabfacb/Source/WTF/wtf/Dominators.h>

⁵L Georgiadis. “Linear-Time Algorithms for Dominators and Related Problems”. PhD thesis. Princeton University, Nov. 2005

⁶PF Dietz. “Maintaining order in a linked list”. In: *STOC*. 1982, pp. 122–127. URL: <https://dl.acm.org/doi/pdf/10.1145/800070.802184>.

- Visit nodes in reverse post-order (i.e., topological order)
- Option 2:⁷
 - Do a DFS over dominator tree
 - Use scoped hashmap to track available values

Does this work? Yes.

[Slide 155] CSE: Hashing an Instruction (and Beyond)

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

[Slide 156] Global Value Numbering – or: advanced CSE

- 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 beneficial⁸

5.5. Simple Transformations

[Slide 157] Simple Transformations: Inlining

- Estimate whether inlining is beneficial
 - Savings of avoided call/computations/branches; cost of increased size
- Copy original function in place of the call
 - Split basic block containing function call
- Replace returns with branches and ϕ -node to/at continuation point
- Move `alloca` to beginning or save stack pointer
 - Prevent unbounded stack growth in loops
 - LLVM provides `stacksave/stackrestore` intrinsics
- Exceptions may need special treatment

⁷P Briggs, KD Cooper, and LT Simpson. *Value numbering*. Tech. rep. CRPC-TR94517-S. Rice University, 1997. URL: <https://www.cs.rice.edu/~keith/Promo/CRPC-TR94517.pdf.gz>.

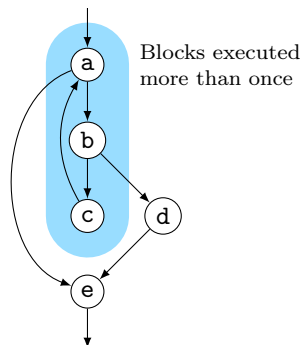
⁸K Gargi. “A sparse algorithm for predicated global value numbering”. In: *PLDI*. 2002, pp. 45–56.

[Slide 158] Simple Transformations: Mem2Reg and SROA

- Mem2reg: promote `alloca` to SSA values/`phis`
 - Condition: only `load/store`, no address taken
 - Essentially just SSA construction
 - Not run in default pipeline, subsumed by SROA
- SROA: scalar replacement of aggregate
 - Separate structure fields into separate variables
 - Also promote them to SSA

5.6. Loop Analysis**[Slide 159] What is a Loop?**

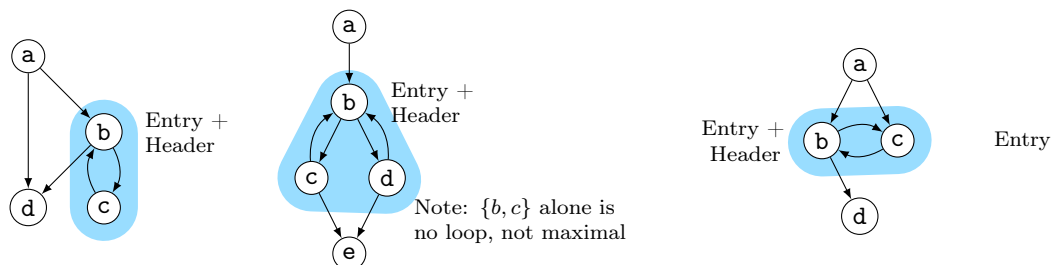
```
void func() {
  while (a()) {
    if (b()) {
      d();
      break;
    }
    c();
  }
  e();
}
```



- Loops in source code \neq loops in CFG
 - d is *not* part of loop: executed at most once
- \rightsquigarrow Need algorithm to find loops in CFG

[Slide 160] Loops

- Loop: maximal SCC L with at least one internal edge⁹ (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 $L \setminus \{h\}$



⁹P Havlak. "Nesting of reducible and irreducible loops". In: *TOPLAS* 19.4 (1997), pp. 557–567. URL: <https://dl.acm.org/doi/pdf/10.1145/262004.262005>.

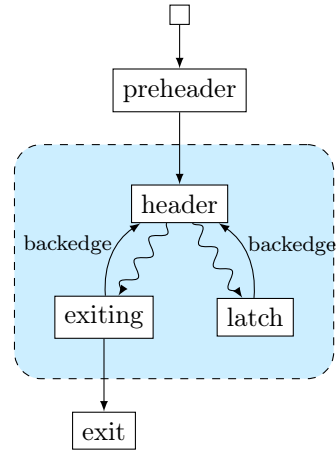
[Slide 161] Natural Loops

- Natural Loop: loop with single entry
 - \Rightarrow Header is unique
 - \Rightarrow Header dominates all block
 - \Rightarrow 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 $\exists h \in L . \forall n \in L . h \text{ dom } n$

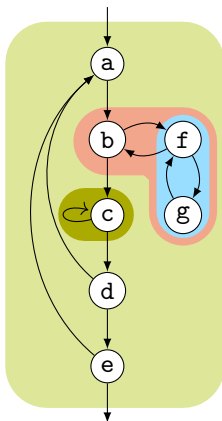
CFG is reducible iff all loops are reducible

**[Slide 162] Finding Natural Loops**

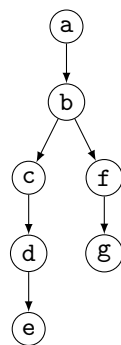
- Modified version¹⁰ of Tarjan's algorithm¹¹
- Iterate over dominator tree in post order
- Each block: find predecessors dominated by the block
 - None \rightsquigarrow no loop header, continue
 - Any \rightsquigarrow loop header, these edges *must* be backedges
- Walk through predecessors until reaching header again
 - All blocks on the way must be part of the loop body
 - Might encounter nested loops, update loop parent

[Slide 163] 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

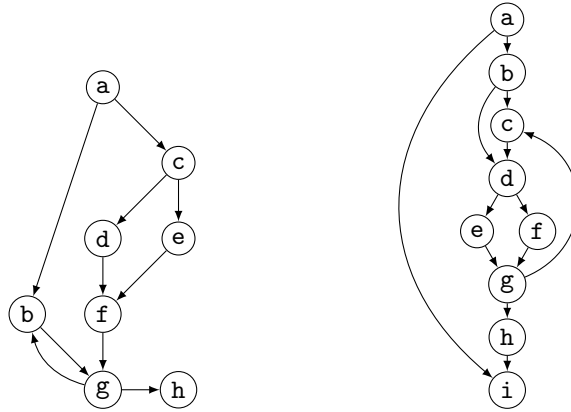
¹⁰G Ramalingam. "Identifying loops in almost linear time". In: *TOPLAS* 21.2 (1999), pp. 175–188. URL: <https://dl.acm.org/doi/pdf/10.1145/316686.316687>.

¹¹R Tarjan. "Testing flow graph reducibility". In: *STOC*. 1973, pp. 96–107. URL: <https://dl.acm.org/doi/pdf/10.1145/800125.804040>.

[Slide 164] Loop Analysis – Example

In-Class Exercise:

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

**[Slide 165] Loop Invariant Code Motion (LICM)**

- Analyze loops, iterate over loop tree in post-order
 - I.e., visit inner loops first
- ↑ Hoist:¹² 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)

5.7. LLVM Passes

[Slide 166] Transformations and Analyses in LLVM: Passes

- Transformations and analyses organized in *passes*
- Pass can operate on Module/(CGSCC)/Function/Loop
- Analysis pass: takes input IR and returns analysis result
 - May also use results of other analyses; results are cached
- Transformation pass: takes input IR and returns preserved analyses
 - Can use analyses, which are re-run when outdated
- Pass manager executes passes on same granularity

¹²<https://github.com/bytecodealliance/wasmtime/blob/bd6fe11/cranelift/codegen/src/licm.rs>

- Otherwise, use adaptor: `createFunctionToLoopPassAdaptor` (and preferably combine multiple smaller passes into a separate pass manager)

[Slide 167] Using LLVM (New) Pass Manager

```
void optimize(llvm::Function* fn) {
    llvm::PassBuilder pb;
    llvm::LoopAnalysisManager lam{};
    llvm::FunctionAnalysisManager fam{};
    llvm::CGSCCAnalysisManager cgam{};
    llvm::ModuleAnalysisManager mam{};
    pb.registerModuleAnalyses(mam);
    pb.registerCGSCCAnalyses(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);
}
```

[Slide 168] 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();
    }
};
// ...
```

[Slide 169] 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 -lLLVM -fPIC  
opt -S -load-pass-plugin=$PWD/testpass.so -passes=testpass input.ll
```

[Slide 170] 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, common sub-expression elimination, loop-invariant code motion
- Compilers often implement transformations as passes
- Analyses may be invalidated by transformations, needs tracking

[Slide 171] 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?
- 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?

6. Vectorization

[Slide 173] Parallel Data Processing

- Sequential execution has inherently limited performance
 - Clock rate, energy consumption/cooling, data path lengths, speed of light, ...

The speed of light is around $3 \cdot 10^8 \frac{m}{s}$. For a 4 GHz processor, during one clock cycle light and therefore also the electricity charges in the circuit can travel *at most* 75 cm. This doesn't consider the latencies of transistors, resistance, and other physical properties, which further restrict the path length.

- 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)

[Slide 174] Single Instruction, Multiple Data (SIMD)

- Idea: perform same operations on multiple data in parallel
- First computer with SIMD operations: MIT Lincoln Labs TX-2, 1957¹

The MIT TX-2 supported computations on one 36-bit integer, two 17-bit integers, four 9-bit integers, and the pair of one 27-bit and one 9-bit integer.

- 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

¹W Clark et al. *The Lincoln TX-2 Computer*. Apr. 1957. URL: http://www.bitsavers.org/pdf/mit/tx-2/TX-2_Papers_WJCC_57.pdf.

[Slide 175] 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

[Slide 176] SIMD ISAs: Design

- Vectors are often implemented as fixed-size wide registers
 - Examples: ARM NEON 32×128-bit, Power QPX 32×256-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

[Slide 177] Historical Development of SIMD Extensions

See Figure 6.1.

[Slide 178] 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

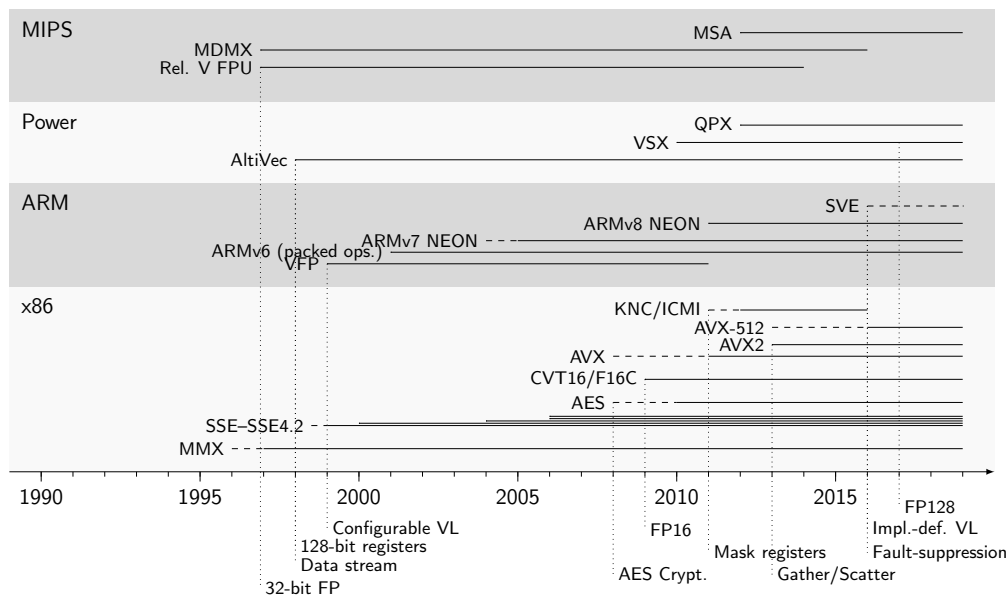


Figure 6.1.: Historical Development of SIMD extensions to contemporary ISAs.

- Cryptography

[Slide 179] 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)

Large vectors (e.g., in AVX-512) cause a high space and power usage in hardware. They also need a higher memory bandwidth – on x86 systems, loading one AVX-512 register means copying an entire cache line. The performance gains of doubling the vector size are smaller than 2x for many applications, and some CPUs cannot sustain executing 512-bit operations at the maximum clock frequency due to power consumption and cooling.

- 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

[Slide 180] 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

[Slide 181] 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

[Slide 182] 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 & ~3ul); i += 4) {
        __m128 x1 = _mm_loadu_ps(&x[i]);
        __m128 y1 = _mm_loadu_ps(&y[i]);
        sum = _mm_add_ps(sum, _mm_mul_ps(x1, y1));
    }
    // ... take care of tail (i..<n) ...
}
```

[Slide 183] 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

[Slide 184] 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

[Slide 185] GCC Vector Extensions

In-Class Exercise:

Compile^a 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;  
}
```

Solution on page 165.

^a<https://godbolt.org/z/43v98rdno>

[Slide 186] LLVM-IR: Vectors

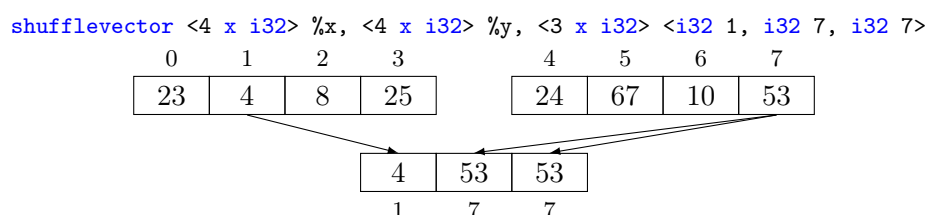
- `<N x ty>` – fixed-size vector type, e.g. `<4 x i32>`
 - Valid element type: integer, floating-point, pointers
 - Memory layout: densely packed (i.e., `<8 x i2> ≈ i16`)
- `<vscale x N x ty>` – scalable vector, e.g. `<vscale x 4 x i32>`
 - Vector with a multiple of N elements
 - Intrinsic `@llvm.vscale.i32()` – 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

Note, however, that few ISAs actually support vector indexing with a dynamic index. The two most widely used architectures, x86-64 and

AArch64, do not. During instruction selection, such IR operations will get lowered to a stack spill and a load of the element from memory.

[Slide 187] 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.. n refer to %x, n .. $2n$ to %y)
 - Result is of type `<m x ty>`



[Slide 188] shufflevector: Examples

In-Class Exercise:

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>`

Solution on page 165.

[Slide 189] LLVM-IR with Vector Instructions: Example

In-Class Exercise:

- 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
}

```

Solution on page 165.

[Slide 190] 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)

```

[Slide 191] 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)

[Slide 192] 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:
    padd xmm0, xmm1
    ret
```

[Slide 193] 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
    padd xmm0, xmm2
    ret
```

[Slide 194] 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
    pcmpgtd xmm0, xmm1
    psubd xmm1, xmm2
    pblendvb xmm1, xmm2, xmm0
    movdqa xmm0, xmm1
    ret
```

[Slide 195] 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

[Slide 196] 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 handling²

[Slide 197] SPMD Implementations on CPUs

- OpenMP SIMD functions
 - Need to be combined with `#pragma omp simd` loops
- Intel ispc³ (Implicit SPMD Program Compiler)
 - Extension of C with keywords `uniform`, `varying`
 - Still active and interesting history⁴
- OpenCL on CPU
 - Very similar programming model
 - But: higher complexity for communicating with rest of application

[Slide 198] 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

²R Karrenberg and S Hack. “Whole-function vectorization”. In: *CGO*. 2011, pp. 141–150.

³M Pharr and WR Mark. “ispc: A SPMD compiler for high-performance CPU programming”. In: *InPar*. 2012, pp. 1–13.

⁴<https://pharr.org/matt/blog/2018/04/30/ispc-all>

[Slide 199] 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

[Slide 200] 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

[Slide 201] 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

[Slide 202] 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

[Slide 203] 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

[Slide 204] Auto-Vectorization: Examples

In-Class Exercise:

Compile^a the functions from `ex06.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?

^a<https://godbolt.org/z/aqbb937cb>

[Slide 205] 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

[Slide 206] 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?

7. Instruction Selection

7.1. Overview

[Slide 208] Code Generation – Overview

- Instruction Selection
 - Map IR to assembly
 - Keep code shape and storage; change operations
- Instruction Scheduling
 - Optimize order to hide latencies
 - Keep operations, may increase demand for registers
- Register Allocation
 - Map virtual to architectural registers and stack
 - Adds operations (spilling), changes storage

[Slide 209] 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]
```

[Slide 210] 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

RISC-V, for example, is an architecture which specifies several instruction sequences that are intended to be macro-fused by the processor for improved performance.

[Slide 211] 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**

The halting problem can be reduced to optimal code generation: if a program continues forever, the program has to be optimized to a simple endless loop to be optimal.

- Alternative: optimal *tiling* of IR with machine code instructions
 - IR as dataflow graph, instr. tiles to optimally cover graph
 - \mathcal{NP} -complete¹

The cited paper shows how SAT can be reduced to optimal tiling of a DAG.

- Additional complication: many different ways to express same computation

[Slide 212] Avoiding ISel Altogether

Use an interpreter

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

Interpreters are in many cases a good alternative to writing code generators — the latter are highly platform-dependent and involve several hard problem, while the former are easy to write in portable languages.

7.2. Macro Expansion

[Slide 213] Macro Expansion

- Expand each IR operation with corresponding machine instrs

¹DR Koes and SC Goldstein. “Near-optimal instruction selection on DAGs”. In: *CGO*. 2008, pp. 45–54.
URL: <http://llvm.org/pubs/2008-CGO-DagISel.pdf>.

%5 = add %1, 12345	→	%5a = movz 12345 %5 = add %1, %5a
%6 = and %2, 7	→	%6 = and %2, 7
%7 = shl %5, %6	→	%7a = lsl %5, %6 %7b = cmp %6, 64 %7 = csel %7a, xzr, %7b, lo

[Slide 214] 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

[Slide 215] Peephole Optimization

- Plain macro expansion leads to suboptimal results
- Idea: replace inefficient instruction sequences²
- Originally: physical window over assembly code
 - Replace with more efficient instructions having same effects
 - Possibly with allocated registers
- Extension: do expansion before register allocation³
 - Expand IR into Register Transfer Lists (RTL) with temporary registers
 - While *combining*, ensure that each RTL can be implemented as single instr.

GCC's RTL is heavily inspired by the original idea of register transfer lists. However, due to severe limitations when optimizing code, the implementation was later expanded and some RTL passes also make use of SSA form.

[Slide 216] 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

²WM McKeeman. “Peephole optimization”. In: *CACM* 8.7 (1965), pp. 443–444. URL: <https://dl.acm.org/doi/pdf/10.1145/364995.365000>.

³JW Davidson and CW Fraser. “Code selection through object code optimization”. In: *TOPLAS* 6.4 (1984), pp. 505–526. URL: <https://dl.acm.org/doi/pdf/10.1145/1780.1783>.

7.3. Tree Covering

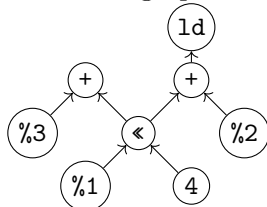
[Slide 217] ISel as Graph Covering – High-level Intuition

- 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

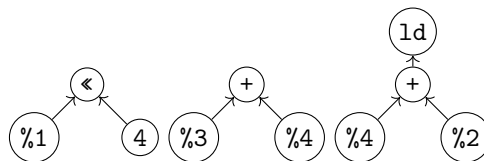
[Slide 218] Tree Covering: Converting SSA into Trees

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

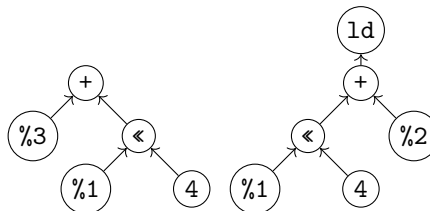
- Data flow graph:



- Method 1: Edge Splitting



- Method 2: Node Duplication



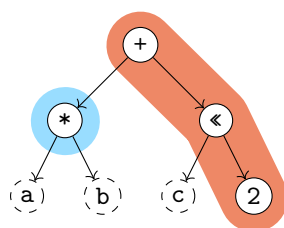
[Slide 219] Tree Covering: Patterns

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

[Slide 220] 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

[Slide 221] Tree Covering: Greedy/Maximal Munch – Example

Matching Patterns:

- $+$: P_1 – cost 1 – covered nodes: 1
- $+$: P_2 – cost 2 – covered nodes: 3 –beamer|beamer: best
- $+$: 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
```

[Slide 222] Tree Covering: with LR-Parsing?

- Can we use (LR-)parsing for instruction selection? Yes!⁴
 - Pattern set = grammar; IR (in prefix notation) = input

Advantages

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

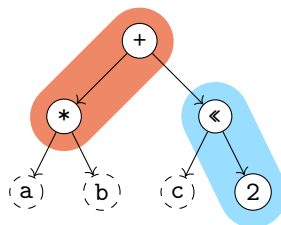
Disadvantages

- 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

[Slide 223] Tree Covering: Dynamic Programming⁵

- 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

[Slide 224] Tree Covering: Dynamic Programming – Example



⁴RS Glanville and SL Graham. “A new method for compiler code generation”. In: *POPL*. 1978, pp. 231–254. URL: <https://dl.acm.org/doi/pdf/10.1145/512760.512785>.

⁵AV Aho, M Ganapathi, and SWK Tjiang. “Code generation using tree matching and dynamic programming”. In: *TOPLAS* 11.4 (1989), pp. 491–516. URL: <https://dl.acm.org/doi/pdf/10.1145/69558.75700>.

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}

[Slide 225] Tree Covering: Dynamic Programming – Off-line Analysis

- Cost analysis can actually be *precomputed*⁶
- 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 tables⁷

[Slide 226] 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`)

7.4. DAG Covering

[Slide 227] 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} -complete⁸

⁶A 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.

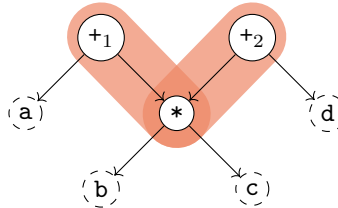
⁷MA Ertl, K Casey, and D Gregg. “Fast and flexible instruction selection with on-demand tree-parsing automata”. In: *PLDI* 41.6 (2006), pp. 52–60.

⁸DR Koes and SC Goldstein. “Near-optimal instruction selection on DAGs”. In: *CGO*. 2008, pp. 45–54. URL: <http://llvm.org/pubs/2008-CGO-DagISel.pdf>.

[Slide 228] DAG Covering: Adapting Dynamic Programming I⁹

- 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

“Roots” in this context refers to all values that are required by other basic blocks and therefore must be computed and stored in some register.

[Slide 229] 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

[Slide 230] DAG Covering: Adapting Dynamic Programming II¹⁰

- 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

⁹MA Ertl. “Optimal code selection in DAGs”. In: *POPL*. 1999, pp. 242–249. URL: <https://dl.acm.org/doi/pdf/10.1145/292540.292562>.

¹⁰DR Koes and SC Goldstein. “Near-optimal instruction selection on DAGs”. In: *CGO*. 2008, pp. 45–54. URL: <http://llvm.org/pubs/2008-CGO-DagISel.pdf>.

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

[Slide 231] DAG Covering: ILP¹¹

- 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

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

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)

[Slide 232] 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

7.5. Graph Covering**[Slide 233] 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)

¹¹DR Koes and SC Goldstein. “Near-optimal instruction selection on DAGs”. In: *CGO*. 2008, pp. 45–54.
URL: <http://llvm.org/pubs/2008-CGO-DagISel.pdf>.

7.6. ISel in Practice

[Slide 234] 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 ~> much slower code!

[Slide 235] 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 artificial¹²

Problem in practice: implementing the huge amount of required patterns

- LLVM X86 back-end has 60k 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

To illustrate the last point: x86 has no instruction to reverse bits of an integer. However, with the GFNI extension, the instruction `vqf2p8affineqb` was introduced, which can be configured in a way that it performs a bit-reverse operation^a. (Would you come up with this pattern?)

^a<https://github.com/llvm/llvm-project/pull/81764>

7.7. LLVM Instruction Selection

[Slide 236] 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

¹²My personal opinion.

[Slide 237] 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, $xzr
    %4:gpr64 = ADDXrs killed %3, %2, 2
    $x0 = COPY %4
    RET_ReallyLR implicit $x0
llc -march=aarch64 -stop-after=finalize-isel
```

[Slide 238] LLVM MIR Example

In-Class Exercise:

Analyze the Machine IR of the following code. (Also consult the reference^a.)

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

```
// clang --target=aarch64 -c -mllvm -stop-after=finalize-isel -O1 -o -
int foo(int n) {
  int r = 1;
  while (n) { r *= n << n; n--; }
  return r;
}
```

Also try `-O0`, `-O2`, `-g`, and `-target=x86_64`.

^a<https://llvm.org/docs/MIRLangRef.html>

[Slide 239] LLVM: Instruction Selectors**FastISel**

- Uses macro expansion
- Low compile-time
- Code quality poor
- Only common cases
- Otherwise: fallback to SelectionDAG
- Default for `-O0`

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 -O0

[Slide 240] 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

See Figure 7.1.

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

Note: needs LLVM debug build

[Slide 241] 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

See Figure 7.2.

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

Note: needs LLVM debug build

[Slide 242] Instruction Selection – Summary

- Instruction Selection: transform generic into arch-specific instructions

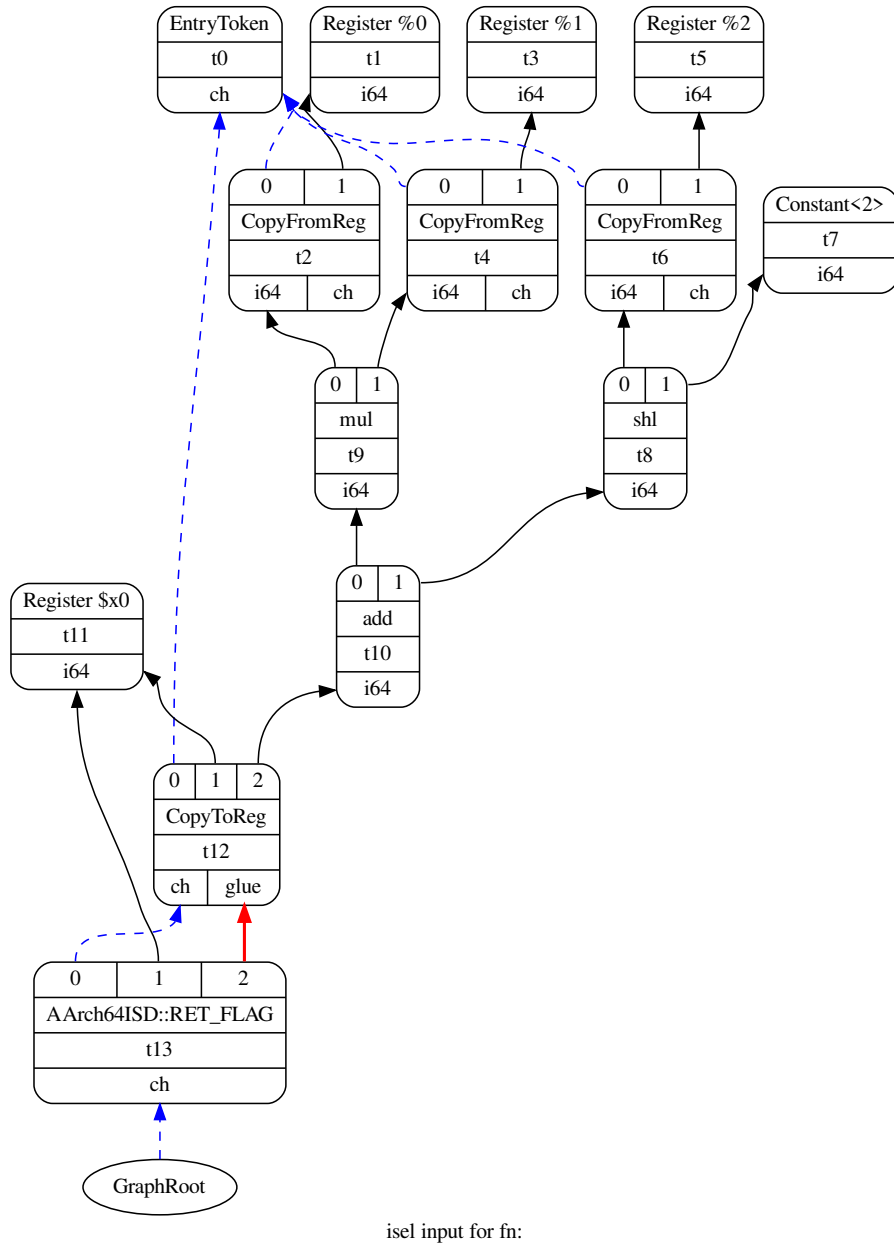


Figure 7.1.: ISel DAG right before instruction selection.

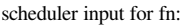


Figure 7.2.: X86 DAG after Instruction Selection

- 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 Covering: support for multi-res instrs., but \mathcal{NP} -complete
- Graph Covering: mightiest, but also most complex, rarely used

[Slide 243] 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?

8. Register Allocation

[Slide 245] Register Allocation

- Map unlimited/virtual registers to limited/architectural registers
- Assign a register to every value
 - Outputs get a (new) register, input operands often require registers
- When running out of registers, move values to stack
 - Stack *spilling* – save value register from to stack memory
- ϕ -nodes: ensure all inputs are assigned to same location
- Goal: produce correct code, minimize extra load/stores
 - Regalloc affects performance in orders of magnitude

[Slide 246] Register Allocation: Overview Example

```
gauss(%0) {  
    %2 = SUBXri %0, 1  
    %3 = MADDXrrr %0, %2, 0  
    %4 = MOVXconst 2  
    %5 = SDIVrr %3, %4  
    ret %5  
}  
gauss(%0 : X0) {  
    %2 = SUBXri %0, 1 : X1  
    %3 = MADDXrrr %0, %2, 0 : X0  
    %4 = MOVXconst 2 : X1  
    %5 = SDIVrr %3, %4 : X0  
    ret %5  
}
```

- May also insert copy and stack spilling instructions

8.1. Avoiding Register Allocation

[Slide 247] Simplest thing that could possibly work

- Idea: allocate a one stack slot for every SSA variable/argument
- Load all instruction operands into registers right before
- Perform instruction
- Write result back to stack slot for that SSA variable

+ Simple, always works, debugging easy
– *Extremely* inefficient in time and space

[Slide 248] Regalloc Example 1

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

gauss(%0 : X0)
    %spills = alloca 40
    STRXi %0, %spills, 0
    %l0 = LDRXi %spills, 0 : X0
    %2 = SUBXri %0%l0, 1 : X0
    STRXi %2, %spills, 8
    %l1 = LDRXi %spills, 0 : X0
    %l2 = LDRXi %spills, 8 : X1
    %3 = MADDXrrr %l1, %l2, 0 : X0
    STRXi %3, %spills, 16
    %4 = MOVXconst 2 : X0
    STRXi %4,i %spills, 24
    %l3 = LDRXi %spills, 16 : X0
    %l4 = LDRXi %spills, 24 : X1
    %5 = SDIVrr %l3, %l4 : X0
    STRXi %5, %spills, 32
    %l5 = LDRXi %spills, 32 : X0
    ret %l5
```

8.2. Handling PHI Nodes

[Slide 249] Handling PHI Nodes

- ϕ -node needs to become register or stack slot
 - Simplest thing that could possibly work: PHI becomes stack slot
- Remember: ϕ -nodes are executed on the edge
- Idea: predecessors write their value to that location at the end
 - First pass: define/allocate storage for ϕ -node, but ignore inputs
 - Second pass: insert move operations at end of predecessors

[Slide 250] Regalloc Example 2

```
identity(%0)
    br %2
2:
    %3 = phi [ 0, %1 ], [ %4, %2 ]
    %4 = ADDXri %3, 1
    %5 = CMPXrr_BLS %4, %0
    br %5, %2, %6
6:
    ret %3
```

Pass 12

- Original value lost in %6!

```
identity(%0 : X0)
  %spills = alloca 24
  STRXi %0, %spills, 0
  %c0 = MOVXconst 0 : X0
  STRXi %c, %spills, 8
  br %2
2: %3 = phi [ 0, %1 ], [ %4, %2 ]
  %l0 = LDRXi %spills, 8 : X0
  %4 = ADDXri %l0, 1 : X0
  STRXi %4, %spills, 16
  %l4 = LDRXi %spills, 16 : X0
  STRXi %l4, %spills, 8
  %l1 = LDRXi %spills, 16 : X0
  %l2 = LDRXi %spills, 0 : X1
  %5 = CMPXrr_BLS %l1, %l2
  br %5, %2, %6
6: %l3 = LDRXi %spills, 8 : X0
  ret %l3
```

[Slide 251] Critical Edges

- 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 \Rightarrow unnecessary and – worse – incorrect



- *Break* critical edges: insert an empty block

[Slide 252] Regalloc Example 2 – Attempt 2

```
identity(%0)
  br %2
2:
  %3 = phi [ 0, %1 ], [ %4, %6 ]
  %4 = ADDXri %3, 1
  %5 = CMPXrr_BLS %4, %0
  br %5, %6, %7
6:
  br %2
7:
  ret %3
```

Pass 12

```

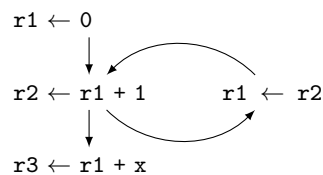
identity(%0 : X0)
    %spills = alloca 24
    STRXi %0, %spills, 0
    %c0 = MOVXconst 0 : X0
    STRXi %c, %spills, 8
    br %2
2: %3 = phi [ 0, %1 ], [ %4, %6 ]
    %l0 = LDRXi %spills, 8 : X0
    %4 = ADDXri %l0, 1 : X0
    STRXi %4, %spills, 16
    %l1 = LDRXi %spills, 16 : X0
    %l2 = LDRXi %spills, 0 : X1
    %5 = CMPXrr_BLS %l1, %l2
    br %5, %6, %7
6: %l4 = LDRXi %spills, 16 : X0
    STRXi %l4, %spills, 8
    br %2
7: %l3 = LDRXi %spills, 8 : X0
    ret %l3

```

[Slide 253] Handling Critical Edges

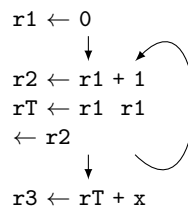
Breaking Edges

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



Copy Used Values

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



[Slide 254] Regalloc Example 3

```

    br %2
2:
    %3 = phi [ %0, %1 ], [ %8, %7 ]

```

```

%4 = phi [ 1, %1 ], [ %5, %7 ]
%5 = phi [ 0, %1 ], [ %4, %7 ]
%6 = CBNZXX(%3)
br %6, %7, %9
7:
%8 = SUBXri %3, 1
br %2
9:
ret %4

```

- Value of ϕ node lost!

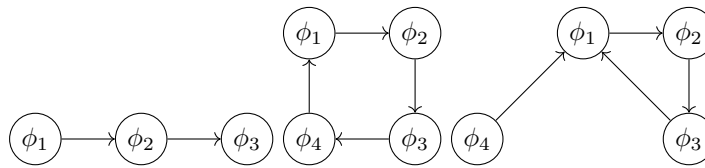
```

odd(%0 : X0)
%spills = alloca 40
STRXi %0, %spills, 0
%l3 = LDRXi %spills, 0 : X0; STRXi %l3, %spills, 8
%c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16
%c1 = MOVXconst 0 : X0; STRXi %c1, %spills, 24
br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
%4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
%5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
%l0 = LDRXi %spills, 8 : X0
%6 = CBNZXX(%l0)
br %6, %7, %9
7:%l1 = LDRXi %spills, 8 : X0
%8 = SUBXri %l1, 1 : X0; STRXi %8, %spills, 32
%l4 = LDRXi %spills, 32 : X0; STRXi %l4, %spills, 8
%l5 = LDRXi %spills, 24 : X0; STRXi %l5, %spills, 16
%l6 = LDRXi %spills, 16 : X0; STRXi %l6, %spills, 24
br %2
9:%l2 = LDRXi %spills, 24 : X0
ret %l2

```

[Slide 255] PHI Cycles

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



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

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

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

[Slide 256] Handling PHI Cycles

1. Compute number of other ϕ nodes reading other ϕ on same edge
2. For each ϕ with 0 readers: handle node/chain
 - No readers \rightsquigarrow start of chain
 - Handling node may unblock next element in chain
3. For all remaining ϕ -nodes: must be cycles, reader count always 1
 - Choose arbitrary node, load to temporary register, unblock value
 - Handle just-created chain
 - Write temporary register to target

Resolving ϕ cycles requires an extra register (or stack slot)

[Slide 257] Regalloc Example 3 – Attempt 2

Edge %1 \rightarrow %2 Edge %7 \rightarrow %2

Critical ϕ :

- %4 #readers: 10 – broken
- %5 #readers: 10

Action: break %4

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %l3 = LDRXi %spills, 0 : X0; STRXi %l3, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16
  %c1 = MOVXconst 0 : X0; STRXi %c1, %spills, 24
  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %l0 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%l0)
  br %6, %7, %9
7:%l1 = LDRXi %spills, 8 : X0
  %8 = SUBXri %l1, 1 : X0; STRXi %8, %spills, 32
  %l4 = LDRXi %spills, 32 : X0; STRXi %l4, %spills, 8
  %l5 = LDRXi %spills, 24 : X1
  %l6 = LDRXi %spills, 16 : X0; STRXi %l6, %spills, 24
  STRXi %l5, %spills, 16
  br %2
9:%l2 = LDRXi %spills, 24 : X0
  ret %l2
```


[Slide 258] SSA Destruction

In-Class Exercise:

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

1. Dependencies between ϕ -nodes?
 2. Critical Edges? (Draw CFG)
 3. Destruct SSA into form with unlimited registers.
 - a) ... by breaking critical edges
 - b) ... by copying used values
-

8.3. Better Register Allocation

[Slide 259] Better Register Allocation

- Goal: keep as many values in registers as possible
 - Less stack spilling \Rightarrow better performance
- Problem: register count (severely) limited
- \rightsquigarrow Are there enough registers? (otherwise: spilling)
- \rightsquigarrow Which register to choose?
- \rightsquigarrow Which register to kill and put on the stack?
- Needs information when value is actually needed

[Slide 260] Interlude: Register Allocation Research – Executive Summary

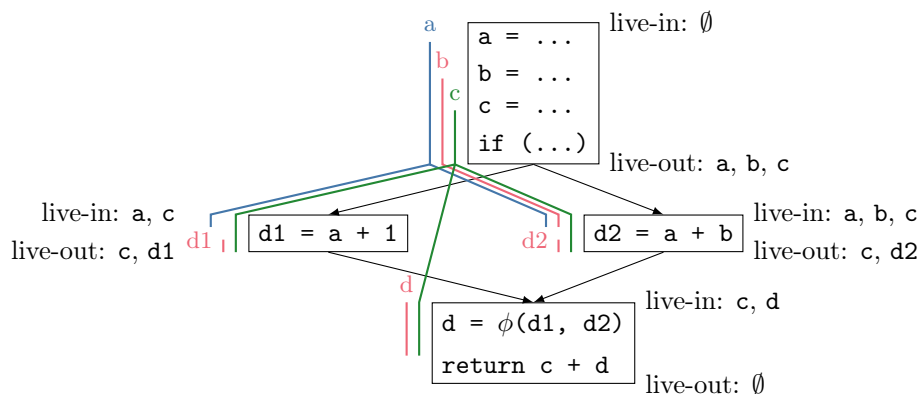
- *Tons* of papers exist
- Papers often skip over important details
 - E.g., when spilling – using the value needs another register
 - E.g., temporary register for shuffling values
- Additional (ISA) constraints in practice: (incomplete list)

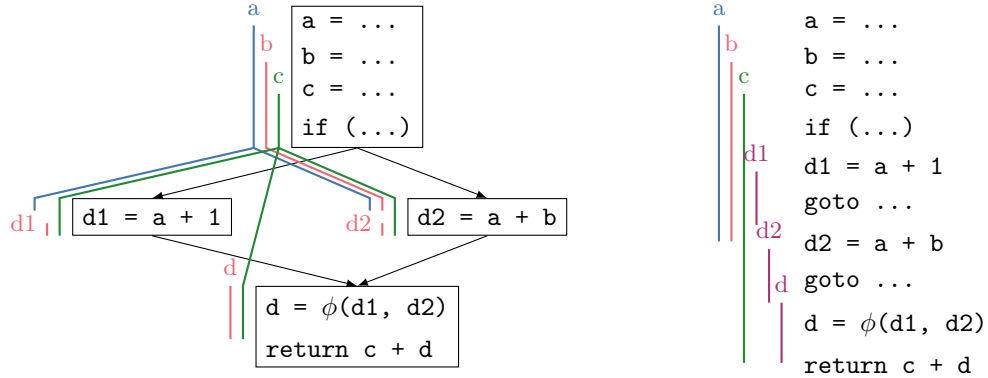
- 2-address instructions with destructive source
- Fixed registers for specific instructions
- Computing the stack address may need yet another register
- Different register classes, often just handled independently
- Implementations even of simple algorithms tend to be large and complex

[Slide 261] Liveness Analysis – Definitions

- *Live*: value still used afterwards
 - 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 interval*: over-approximation of live ranges without holes
 - Depends on block order, reverse post-order often a good choice
- *Live-in/Live-out*: values live at begin/end of basic block
 - For ϕ nodes: ϕ is live-in, operands are live-out in predecessors (Note: different literature uses different definitions)

[Slide 262] Liveness Analysis – Example



[Slide 263] Liveness Analysis – Example – Live Ranges vs. Live Intervals

- Live intervals are substantially worse, but easier to compute

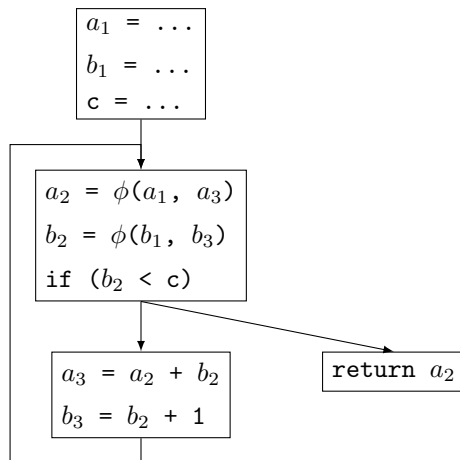
[Slide 264] Liveness Analysis – Algorithm¹

- Iterate over blocks in post-order
 - $live \leftarrow \cup s.liveIn \setminus s.phis, s \in b.successors$
 - $live \leftarrow live \cup \{\phi.input(b) \mid \phi \in b.successors.phis\}$
 - $b.liveOut \leftarrow live$
 - $\forall v \in live : ranges[v].add(b.start, b.end)$
 - For each non- ϕ instruction $inst$ in reverse order
 - * $live \leftarrow (live \cup inst.ops) \setminus \{inst\}$
 - * $ranges[inst].setStart(inst)$
 - * $\forall op \in inst.ops : ranges[op].add(b.start, inst)$
 - $b.liveIn \leftarrow live \cup b.phis$
- Repeat until convergence²

¹Adapted from C Wimmer and M Franz. “Linear scan register allocation on SSA form”. In: *CGO*. 2010, pp. 170–179.

²Reducible graphs: expanding $liveIn$ of loop headers to the entire loop suffices

[Slide 265] Liveness Analysis – Example



[Slide 266] Liveness Analysis – Example

In-Class Exercise:

```

fn(%0, %1) {
b1:
  %2 = add %0, %1
  br %b2
b2:
  %3 = phi [%b1: %1], [%b3: %4]
  %4 = phi [%b1: %0], [%b3: %3]
  %5 = phi [%b1: %2], [%b3: %3]
  %6 = phi [%b1: 0], [%b3: %8]
  %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 [%b2: %4], [%b3: %3]
  %11 = phi [%b2: %5], [%b3: %8]
  %12 = add %10, %11
  %13 = add %12, %2
  ret %13
}
  
```

1. Compute live ranges of every SSA value
-

[Slide 267] Register Allocation Decisions (Outline)

- 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.

[Slide 268] Register Allocation Strategies**Scan-based**

- Iterate over the program
- Decide locally what to do
- Greedily assign registers

Graph-based

- Compute *interference graph*
 - Nodes are values
 - Edge \Rightarrow live ranges overlap
 - Holistic approach
- + Fast, good for straight code
– Code quality often bad
- Used for -O0 and JIT comp.
- + Often generate good code
– Expensive, superlinear run-time
- Used for optimized code

[Slide 269] Linear Scan Register Allocation³

- Idea: treat whole function as single block
 - Block order affects quality (but not correctness)
 - Only consider live intervals without holes
- Iterate over instructions from top to bottom
- For operands of instruction in their last use: mark register as free
- Assign instruction result to new free register
 - If no free register available: move some value to the stack
 - Heuristic: value whose liveness ends furthest in future

[Slide 270] 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

³M Poletto and V Sarkar. “Linear scan register allocation”. In: *TOPLAS* 21.5 (1999), pp. 895–913.

- 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 splitting⁴
- Function calls: clobber lots of registers

[Slide 271] Scan-based Register Allocation⁵

Iterate over basic blocks⁶

- Start with register assignment from predecessor
 - Multiple predecessors: choose assignment from any one
 - ϕ -nodes can either reside in registers or on the stack
- 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 them⁷

[Slide 272] Scan-based Register Allocation – Spilling

What to spill?

- Spill value with furthest use in future⁸
 - 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 fine⁹
 - Downside: superlinear run-time

Where to store?

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

[Slide 273] Scan-based Register Allocation – Spilling

Where to insert store?

⁴O Traub, G Holloway, and MD Smith. “Quality and speed in linear-scan register allocation”. In: *SIGPLAN* 33.5 (1998), pp. 142–151. URL: <https://dl.acm.org/doi/pdf/10.1145/277652.277714>.

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

⁶Typically: reverse post-order, so most predecessors are seen before successors, except for loops.

⁷Without critical edges, only relevant for blocks with one successor — others are visited afterwards by RPO definition.

⁸C Wimmer and H Mössenböck. “Optimized interval splitting in a linear scan register allocator”. In: *VEE*. 2005, pp. 132–141.

⁹Intel Optimization Reference Manual (Aug. 2023), Assembly/Compiler Coding Rules 38 and 45

- 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

[Slide 274] 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

[Slide 275] Graph Coloring Approaches

- + Considerably better results than greedy algorithms
- High run-time, even with heuristics
- Graph coloring in general is \mathcal{NP} -complete
- Often used in compilers (e.g., GCC, WebKit)

AD IN2053 “Program Optimization” covers this more formally

8.4. Generating Assembly

[Slide 276] 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 `alloca`s, 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*

[Slide 277] 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

8.5. Summary

[Slide 278] Register Allocation – Summary

- Map unlimited virtual registers to restricted register set
- Responsible for:
 - Assigning registers to values
 - Deciding which registers to spill to stack
 - Deciding when to spill/unspill values
- ϕ -nodes require extra care, esp. for chains and cycles
- Liveness information is key information for register allocation
- Scan-based approaches are fast, but lead to suboptimal code
- Graph coloring yields better results, but is much slower
- Register allocation/spilling heavily relies on heuristics in practice

[Slide 279] Register Allocation – Questions

- Why is register allocation a difficult problem?
- How are ϕ -nodes handled during register allocation?
- What are the two main problems when destructing ϕ -nodes?
- Why are critical edges problematic and how to deal with them?
- What are practical constraints for register allocation?
- How to detect whether a value is still needed at some point?
- How to compute the live ranges of values in an SSA-based IR?
- What is the idea of linear scan and what are its practical problems?

9. Object Files, Linker, and Loader

[Slide 281] 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

[Slide 282] 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

The small code model allows to use relative jumps/calls to other functions and relative (or 32-bit absolute, directly encodeable in memory operands) accesses to global variables. In the medium code model, this only holds for code references, but all large data sections need to be accessed more indirectly (on x86-64, this requires a 64-bit move-immediate instruction). In the large code model, relative calls can no longer be used, as the function might be more than 2 GiB away — for every function call, the address needs to be materialized with a 64-bit move followed by an indirect call.

Some ABIs define other code models, e.g., AArch64 has a tiny code model that restricts the code/data size of the binary to 1 MiB. This allows to use the `adr` instruction (21-bit offset) to reference global variables instead of the typical `adrp/adr` pair. Additionally, conditional branches to other functions are possible in this code model.

As a general rule of thumb: the smaller the code model, the more efficient is the

resulting code.

- 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)

Although required for shared libraries and increasingly used for binaries as a security measure, position-independent code (PIC) can have a significant performance impact, as an extra indirection step is needed whenever an absolute address is required.

Compiler needs to know code model

9.1. Object Files

[Slide 284] Executable and Linkable Format (ELF)

- Widely used format for code
 - REL: relocatable/object file
 - EXEC: executable (non-PIE)
 - DYN: shared library/PIE
 - CORE: coredump
- ELF header: general information
- Program headers: used for execution
- Section headers: used for linking

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

[Slide 285] 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;
```

[Slide 286] 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
- `.symtab` – symbol table, references string table for names
- `.shstrtab` – string table for section header names

[Slide 287] 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 1 "varname" String 4 "name" String 9 "foo"
```

[Slide 288] 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
```

[Slide 289] 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	00		0	0	0
[1]	.text	PROGBITS	00	AX	0	0	1
[2]	.rela.text	RELA	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	RELA	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

[Slide 290] 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;
```

Not all global variables or functions need to show up in the symbol table (e.g., global objects with the `private` linkage in LLVM are omitted from the symbol tables, as are local labels from the assembler).

[Slide 291] 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

[Slide 292] Example: Writing Code to .text

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

- Symbol may be unknown
- Linker needs to resolve offset later

↪ Relocations

```
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_PC32a  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
```

^aRecent GAS emits R_X86_64_PLT32, which is equivalent for local symbols.

[Slide 293] Relocations

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

Static Relocation

ET_REL

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

Dynamic Relocation

ET_EXEC/ET_DYN

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

[Slide 294] Relocation Types

- Types and meaning defined by psABI¹

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	$Z + A$

[Slide 295] 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 symtab -----/  |
      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
000000000000000c 0000000400000002 R_X86_64_PC32  foo - 4
0000000000000011 0000000600000004 R_X86_64_PLT32  external - 4
```

¹x86-64: HJ Lu et al. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*. 2022. URL: <https://gitlab.com/x86-psABIs/x86-64-ABI/-/jobs/artifacts/master/raw/x86-64-ABI/abi.pdf?job=build>.

[Slide 296] Relocations on RISC Architectures

- RISC architectures typically have *more* relocation types
 - Example: AArch64² 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 ± 4 GiB range, `add` or `load` offset for low bits
 - Scaled load offsets require different relocations for each scale

[Slide 297] Branch Relocations

- Branches (often) have limited range; compiler must assume max. distance
- x86-64: ± 2 GiB range, if larger use `mov` and indirect jump
- AArch64: ± 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*: ± 1 MiB range \rightsquigarrow needs ind. jump *often*

[Slide 298] Branch Relocations on RISC-V

In-Class Exercise:

1. Compile the code 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?
 3. Link the file: `ld.lld -shared -o rv.so rv.o` and disassemble `rv.so`.
What is different now?
-

²Arm Ltd. *ELF for the Arm 64-bit Architecture (AArch64)*. URL: <https://github.com/ARM-software/abi-aa/blob/main/aaelf64/aaelf64.rst> (visited on 11/21/2022).

9.2. Executable Files

[Slide 300] Linker³

- 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.
 7. Profit!

[Slide 301] ELF Executable File

- Entry in ELF header: entry address of the program
 - Typically provided by libc to call `__libc_start_main`
- Program headers: instructions for loading the program
- PT_PHDR: described program headers
- PT_LOAD: loadable segment
 - Specifies virtual address, file offset, file size/memory size, permission
 - `vaddr & (pagesize-1) == offset & (pagesize-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

[Slide 302] Example: Program Headers

Program Headers:

Type	Offset	VirtAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x000000	0x00400000	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	0x000000	0x00000000	0x000000	0x000000	RW	0x10

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

³Interesting blog on LLD: F Song. *Personal Blog*. URL: <https://maskray.me/> (visited on 11/21/2022).

[Slide 303] Loading a Binary to Memory

- Load ELF header and program header
 - If ET_DYN (\leadsto 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

9.3. Linker Optimizations

[Slide 305] 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 \leadsto foobar\0`
 - Sort strings from tail (e.g., radix sort), deduplicate neighbors

[Slide 306] COMDAT Groups

In-Class Exercise:

```
//--- 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.o inline{1,2}.o
```

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

[Slide 307] 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`

[Slide 308] 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

[Slide 309] Link-Time Optimization

- Problem: Compilers still suck 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 (`-flto`)
 - Done as very first step at link-time
- LTO is widely used and highly effective

9.4. Static Libraries

[Slide 311] 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

9.5. Shared Libraries

[Slide 313] 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

[Slide 314] Shared Libraries: Changes in Compiler

None ☺ (almost)

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

[Slide 315] 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, symtab, strtab, ...

[Slide 316] 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

[Slide 317] 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

[Slide 318] 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 sparse⁴

⁴A Roenky. *ELF: better symbol lookup via DT_GNU_HASH*. URL: <https://flapenguin.me/elf-dt-gnu-hash> (visited on 12/14/2022)

[Slide 319] Miscellaneous Things

- Purpose of all these dynamic entries
- Symbols: versioning and visibility
- 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`)

[Slide 320] 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
- Linker can perform further optimizations

[Slide 321] 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?
- 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?

10. Unwinding and Debuginfo

[Slide 323] 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: coredumps, debuggers, profilers
- Debugging experience enhanced by variables, files, lines, statements, etc.

[Slide 324] 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

10.1. Stack Unwinding

[Slide 325] 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`

The two-phase separation is not strictly necessary for languages like C++. A primary benefit is that the cleanup can be avoided if no handler is found, in which case the program can abort directly. Another benefit is that the separation allows for resumptive exceptions handling, i.e. where the error condition is corrected and the program resumes at the point where the exception was raised.

Language-specific data gives the personality function information about the structure of the function, the location of exception handlers, and the types of handled exceptions. Note that in C++ exception handlers not just correspond to `catch` handlers, but also to destructors that need to be executed during unwinding.

Forced unwinding is slightly different and skips the search phase.

[Slide 326] 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
 - Update some registers with exception data

[Slide 327] 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
- High overhead in non-exceptional case

[Slide 328] 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
- Still needs to find meta-information

```
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
```

[Slide 329] Stack Unwinding: Without Frame Pointer

- Definition: *canonical frame address (CFA)* is **sp** at the function call
 - For x86-64, where **call** modifies the frame pointer, this refers to the **rsp** *before* the call, so that the return address is stored at the address $CFA - 8$.
 - Given: **pc** and **sp** (bottom of stack frame/call frame)
 - In parent frames: $retaddr - 1 \sim pc$ and $CFA \sim sp$
 - We cannot take *retaddr* as previous program counter: at that point, the stack frame might have a different layout, e.g. for ABIs where the callee is responsible for removing stack arguments or in case of a **noreturn** function. It is a safe assumption that every call instruction has a size of at least one byte.
 - Need to map **pc** to stack frame size
 - $sp + framesize = CFA$
 - Stack frame size 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
- Other reg. can act as frame pointer, register saved in other register, ...
 - Need to know where return address is stored

10.2. Call Frame Information

[Slide 330] 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

[Slide 331] Call Frame Information – Example 1

	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	

The table describes how to reach the top of the stack frame (the CFA) and the location of saved registers within the stack frame for every instruction: each row describes the state when the program counter points to the instruction (thus, the row describes the state *before* the instruction is executed).

[Slide 332] 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	

[Slide 333] 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	

[Slide 334] Call Frame Information – Exercise

In-Class Exercise:

- Download `ex10.txt` from the course website
- Construct the CFI tables for both functions (you can omit lines that don't change)

Solution on page 166.

[Slide 335] 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

Note that the bytecode is just a compression of the table, which describes a mapping from a program counter to a stack frame layout. This is completely independent from the dynamic execution trace that was executed — the unwinder does not know which instructions were executed previously or through which code path the current instruction was reached.

Important operations (see DWARF standard^a section 6.4.2 for a complete list):

- `DW_CFA_def_cfa`: define CFA register and offset
- `DW_CFA_def_cfa_offset`: define CFA offset, keep register unchanged
- `DW_CFA_def_cfa_register`: define CFA register, keep offset unchanged
- `DW_CFA_offset`: register is stored at CFA plus offset
- `DW_CFA_restore`: register has same value as at function entry (i.e., same value as specified at the end of the bytecode of the Common Information Entry (CIE, see below))
- `DW_CFA_undefined`: register is undefined/not recoverable
- `DW_CFA_same_value`: register is unmodified
- `DW_CFA_advance_loc`: create new row and advance machine code offset

^aDWARF Debugging Information Committee. *DWARF Debugging Information Format Version 5*. Feb. 2017. URL: <http://dwarfstd.org/doc/DWARF5.pdf>.

[Slide 336] Call Frame Information: Bytecode – Example 1

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

[Slide 337] Call Frame Information: Bytecode – Example 2

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

[Slide 338] 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: {}

[Slide 339] Call Frame Information: Bytecode – Exercise

In-Class Exercise:

- For the functions in `ex10.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?

Solution on page 166.

[Slide 340] Call Frame Information: Bytecode

- DWARF¹ 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>`

[Slide 341] Call Frame Information: `.eh_frame_hdr`²

- 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

[Slide 342] 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`

[Slide 343] Call Frame Information: Assembler Directives – Example

```
int bar(int*);
int foo(unsigned long x) {
    int arr[x * 4];
```

¹DWARF Debugging Information Committee. *DWARF Debugging Information Format Version 5*. Feb. 2017. URL: <http://dwarfstd.org/doc/DWARF5.pdf>.

²https://refspecs.linuxfoundation.org/LSB_4.1.0/LSB-Core-generic/LSB-Core-generic/ehframechpt.html

```
    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
```

[Slide 344] 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

[Slide 345] Unwinding: Other Platforms

- Unwinding depends *strongly* on OS and architecture
- GNU/Linux uses DWARF
- Apple has modified compact version
- Windows has SEH with kernel-support for unwinding
- IBM AIX has their own format
- AArch32 has another custom format
- Additionally: minor differences for return address, stack handling, ...

Needs to work reliably for exception handling

10.3. Debug Information

[Slide 346] 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

[Slide 347] Debug Frame Information

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

[Slide 348] 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`

[Slide 350] DWARF: Hierarchical Program Description

- Extensible, flexible, Turing-complete³ 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, ...

[Slide 351] DWARF: Data Types

```
DW_TAG_structure_type [0x2e]
  DW_AT_byte_size (0x08)
  DW_AT_sibling (0x4a)
  DW_TAG_member [0x37]
```

³J Oakley and S Bratus. “Exploiting the Hard-Working DWARF: Trojan and Exploit Techniques with No Native Executable Code”. In: *WOOT*. 2011. URL: https://www.usenix.org/events/woot11/tech/final_files/Oakley.pdf.

```

    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")

```

[Slide 352] 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")
    // ...

```

[Slide 353] 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, ...

[Slide 354] 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

[Slide 356] 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

[Slide 357] 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

[Slide 358] 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?

11. JIT Compilation and Sandboxing

11.1. JIT Compilation

[Slide 360] 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 PGO¹

[Slide 361] 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

This approach *is* used in practice, as it does not require further knowledge in compilers or operating systems.

[Slide 362] 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

¹Profile-Guided Optimization; GCC: `-fprofile-generate` to store information about branches/values; `-fprofile-use` to use it

- Before execution: change protection to (read-)execute

[Slide 363] JIT: Making Code Executable

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

[Slide 364] 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-V² (Linux), all non-x86 (Darwin): system call
- Skipping ICache flush: spurious, hard-to-debug problems

[Slide 365] Code Generation: Differences AoT vs. JIT

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

- JIT compiler and linker are often merged

[Slide 366] 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

²RISC-V has user `fence.i`, but only affects current core

- 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)

In practice, the performance difference between the large and Small-PIC model is often negligible. Small-PIC tends to be slightly more efficient when there are many references to globals from the same JIT-compiled object file.

The other consideration is compile-time: Compiler frameworks like LLVM are optimized for the small code model; using the large code model often causes fallbacks to a slower/less optimized code path, which can lead to a substantial compile-time increase.

[Slide 367] 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`

[Slide 368] 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

[Slide 369] 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
- libc provides `__register_frame` and `__deregister_frame`
 - Call with address of first Frame Description Entry (FDE)
 - Historically also called by init code

[Slide 370] 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

[Slide 371] 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`³
 - Text file; format: `startaddr size name\n`
- Approach 2: *needs an extra slide*

[Slide 372] JIT: Linux perf JITDUMP format (optional)

- JIT-compiler dumps function name/address/size/code⁴
 - 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 \rightsquigarrow 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!

[Slide 373] Compilation Time

- Problem: code generation takes time

³<https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/perf/Documentation/jit-interface.txt>

⁴<https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/perf/Documentation/jitdump-specification.txt>

- 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

[Slide 374] Compilation Time: Simple Approach

Caching

- Doesn't work on first execution

Although caching is frequently suggested as solution for addressing long compile times, it doesn't really solve the problem. While cache hits can be quite likely on repeated execution, cache misses *do* happen and in these cases, compile times can matter.

[Slide 375] 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

[Slide 376] 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

[Slide 377] 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”

11.2. Sandboxing

[Slide 378] 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

[Slide 379] 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)

[Slide 380] 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

[Slide 381] 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

[Slide 382] Approach: Sandbox Native Code with Modification

- 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
- Google Native Client⁵, originally x86-32, ported to x86-64 and ARM
- Designed as browser extension
- Native code shipped to browser, executed after validation

[Slide 383] 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

⁵B Yee et al. “Native client: A sandbox for portable, untrusted x86 native code”. In: *SP*. 2009, pp. 79–93.

[Slide 384] NaCl on non-i386 Systems

- Other architectures⁶ use base register instead of segment offsets
 - Additional verification required
- Deprecated in 2017 in favor of WebAssembly
- + 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

[Slide 385] 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

[Slide 386] 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.js⁷: 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)`

[Slide 387] 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
```

⁶D Sehr et al. “Adapting Software Fault Isolation to Contemporary {CPU} Architectures”. In: *19th USENIX Security Symposium (USENIX Security 10)*. 2010.

⁷D Herman, L Wagner, and A Zakai. *asm.js*. 2014. URL: <http://asmjs.org/spec/latest/>.

```
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

[Slide 388] 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

[Slide 389] 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! \rightsquigarrow 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

[Slide 390] 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

[Slide 391] 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

[Slide 392] 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?

12. Binary Translation

12.1. Overview

[Slide 394] Motivation

- Run program on other architecture
- Use-case: application compatibility
 - Other architecture with incompatible instruction encoding
 - Applications using unavailable ISA extensions¹
- Use-case: architecture research
 - Development of new ISA extensions without existing hardware

[Slide 395] 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

12.2. Binary Translation

[Slide 396] 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.

¹Exception-based implementation possible, but slow.

[Slide 397] 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

[Slide 398] 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

[Slide 399] Dynamic Binary Translation

See Figure 12.1.

- 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

[Slide 400] Dynamic Binary Translation: Code Fragment

RISC-V Code

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

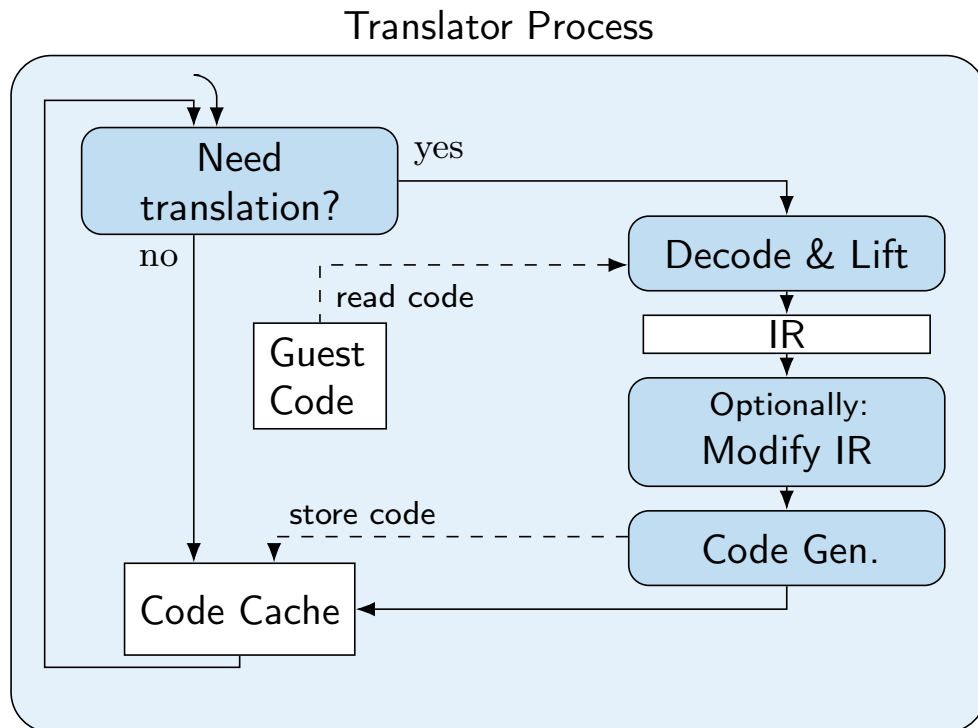


Figure 12.1.: Overview of a dynamic binary translator.

Translation Engine

```

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

```

Semantical representation

```

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

```

Every block of guest code is translated into a function that modifies the emulated CPU state and returns the address of the next block (which can be dynamic in case of a conditional or indirect jump).

The translation engine merely drives the translation and execution. Instead of returning to the translator for every dispatch, this can also be implemented through

| tail calls in every translated block.

12.3. Guest State

[Slide 401] 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

[Slide 402] 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

12.4. Fast DBT

[Slide 403] 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

[Slide 404] 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

[Slide 405] 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
// ...
mov rdi, 0x40068c
lea rsi, [rip+1f]
jmp translate_and_dispatch
1:.byte ... // store patch information

// After patching
// ...
jmp trans_40068c
// (garbage remains)
```

[Slide 406] 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

[Slide 407] 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

[Slide 408] 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)

[Slide 409] 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

12.5. Correct DBT

[Slide 410] 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)

[Slide 411] 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, SIGCHLD
 - Delivered externally, e.g. using `kill`
 - Can be delivered to any thread at any time
 - (usually a bad idea to use them)

[Slide 412] 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

[Slide 413] Correct DBT: Memory Accesses

- Option: emulating paging in software (slow, but works)
 - Every memory accesses 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

[Slide 414] 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

[Slide 415] 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

[Slide 416] 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)

[Slide 417] 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?
- `cputid` – how to emulate?
 - Cache sizes, processor model, ...
 - Application can do timing experiment to detect DBT

[Slide 418] 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

[Slide 419] 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?

13. Query Compilation

13.1. Overview

[Slide 421] 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

[Slide 422] 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

[Slide 423] 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.

13.2. Subscripts

[Slide 424] 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: $\bowtie_{s.matrn=r.h.matrn}^{HJ}, \sigma_{a.x < 5 \cdot (b.y - a.z)}$

[Slide 425] Subscripts: Execution

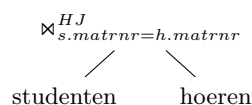
- Option: keep as tree, interpret
 - + Simple, flexible
 - Slow
- Option: compile to bytecode
 - + More efficient
 - More effort to implement, some compile-time
- Option: compile to machine code
 - Code can be complex to accurately represent semantics
 - + Most efficient
 - Most effort to implement, may need short compile-times

[Slide 426] 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

13.3. Query Execution

[Slide 427] 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

[Slide 428] Iterator Model¹

- 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

[Slide 429] 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;
        }
    }
}

```

¹G Graefe. “Volcano—an extensible and parallel query evaluation system”. In: *IEEE Transactions on Knowledge and Data Engineering* 6.1 (1994), pp. 120–135.

```
}  
};
```

- HashJoin builds hash table on first read; materialization might be useful

[Slide 430] 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

[Slide 431] Push-based Model²

- 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

[Slide 432] 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;
```

²T Neumann. “Efficiently compiling efficient query plans for modern hardware”. In: *VLDB* 4.9 (2011), pp. 539–550.

```
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));  
    }  
}  
};
```

[Slide 433] 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

[Slide 434] Pull-based Model vs. Push-based Model³

- 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

[Slide 435] 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

[Slide 436] Code Generation for Push-Based Model

- Inlining code in push-based model yields nice code

³A Shaikhha, M Dashti, and C Koch. “Push versus pull-based loop fusion in query engines”. In: *Journal of Functional Programming* 28 (2018).

- 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)

[Slide 437] Code Generation: Example

$\sigma_{s.matrnur=h.matrnur}$
 |
 ×
 / \
 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_matrnur == t.h_matrnur)
                    out.write(t);
            }
        }
    }
};
  
```

[Slide 438] How to Generate Code

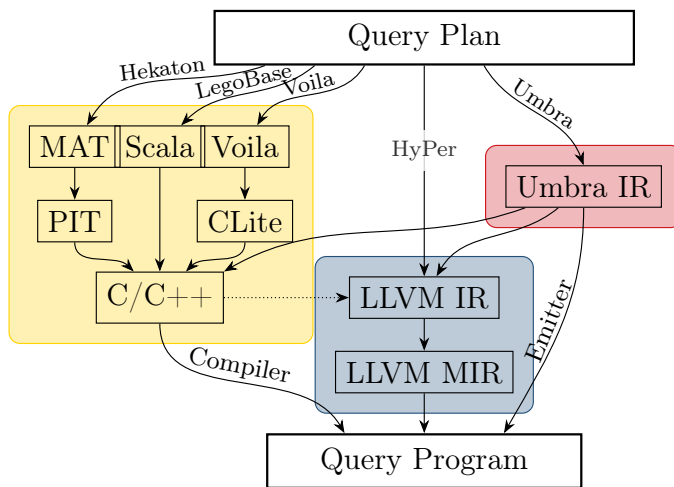
- Code generator executes produce/consume methods
 - Method bodies don't do actual operations, but construct code
 - E.g., call **IRBuilder**
 - Call to helper functions for complex operations e.g. hash table insert/lookup, string operations, memory allocation, etc.
- Resulting code doesn't contain produce/consume 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

[Slide 439] 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)

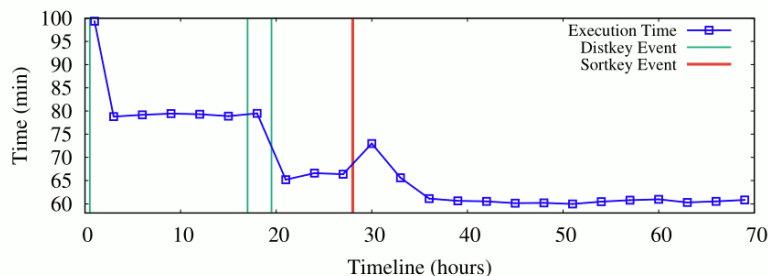
[Slide 440] What to Generate



[Slide 441] Case Study: Amazon Redshift⁴

“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.”

[Slide 442] Case Study: Amazon Redshift⁵

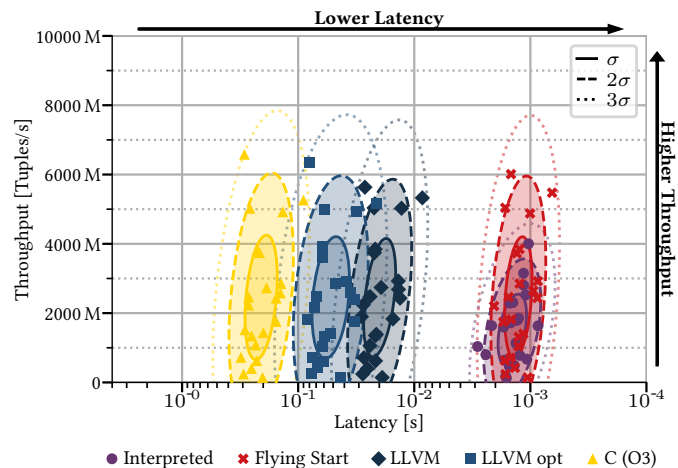


⁴N Armenatzoglou et al. “Amazon Redshift Re-invented”. In: *SIGMOD*. 2022.

⁵N Armenatzoglou et al. “Amazon Redshift Re-invented”. In: *SIGMOD*. 2022.

“Figure 7(a) illustrates [...] from an out-of-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).”

[Slide 443] Compile Times: Umbra



TPC-H sf=30, AMD Epyc 7713 (64 Cores, 1TB RAM)

[Slide 444] 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

[Slide 445] 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

[Slide 446] 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

[Slide 447] 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.

[Slide 448] 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?

A. Exercise Solutions

[Slide 185]

The results diverge strongly depending on whether the operations is natively supported by the architecture. For some operations, the output is as intended:

```
// typedef uint32_t vecty __attribute__((vector_size(16)));
// vecty op(vecty a, vecty b) { return a + b; }
op:
    paddb xmm0, xmm1
    ret
```

For some operations, some legalization needs to be applied, e.g., to widen the elements to the nearest supported size:

```
// typedef uint8_t vecty __attribute__((vector_size(8)));
// vecty op(vecty a, vecty b) { return a * b; }
op:
    punpcklbw xmm0, xmm0 // zero extend elements to 16 bit
    punpcklbw xmm1, xmm1
    pmullw xmm0, xmm1 // 16-bit multiplication
    pand xmm0, xmmword ptr [rip + .LCPI0_0] // {0x00ff, 0x00ff, ...}
    packuswb xmm0, xmm0
    ret
```

Other operations need to be scalarized, because the hardware doesn't support the vectorized operation, e.g. for division. This is of course *less* efficient than directly writing the scalarized code: First, the legalization of vector operations only happens inside the back-end and therefore earlier transformations will only see (and probably ignore) the vectorized code. Second, there needs to be code to extract the elements from the vector and re-insert them at the end.

The performance impact of using this type of vector addition is not visible in the source program; even if it looks optimized, the output might be worse than for unvectorized source code. This is part of the reason why using this style of vector programming is not very popular.

[Slide 188]

1. Broadcast `%x` to all lanes of a `<4 x i32>`.
2. Interleave two vectors into a `<4 x i32>`, taking the elements 0 and 2 from the first vector and elements 1 and 3 from the second vector.
3. Concatenate two vectors into a `<8 x i16>`.

[Slide 189]

The LLVM-IR was generated from this code:

```
// clang-18 -emit-llvm -O1 -g0 -mavx2 -mllvm -force-target-max-vector-interleave=1
void foo(long* restrict a, const long* b) {
#pragma clang loop vectorize(enable) interleave(disable)
    for (long i = 0; i < 2048; i++) {
        a[i] = b[i] < i ? i + b[i] : 0;
    }
}
```

[Slide 334]

func1: There is no line required for offset 9; the value in r12 is still valid afterwards. Reducing the number of lines helps with compression. There is no need to adjust the values of rbp/rbx/r12 after they are reloaded from the stack; the x86-64 SysV ABI guarantees a 128 byte red zone below the stack pointer, which will not be overwritten by signal handlers. On ABIs without a red zone (e.g., AArch64 AAPCS), such an optimization would not be permitted.

	CFA	rip	rbp	rbx	r12	...
0:	rsp+0x08	[CFA-8]				
1:	rsp+0x10	[CFA-8]	[CFA-16]			
7:	rbp+0x10	[CFA-8]	[CFA-16]			
a:	rbp+0x10	[CFA-8]	[CFA-16]	[CFA-32]	[CFA-24]	
43:	rsp+0x08	[CFA-8]	[CFA-16]	[CFA-32]	[CFA-24]	

func2: Similar optimizations can be applied to this function.

	CFA	rip	rbp	rbx	r12	r13	r14	r15
0:	rsp+0x08	[CFA-8]						
1:	rsp+0x10	[CFA-8]	[CFA-16]					
4:	rbp+0x10	[CFA-8]	[CFA-16]					
a:	rbp+0x10	[CFA-8]	[CFA-16]			[CFA-40]	[CFA-32]	[CFA-24]
f:	rbp+0x10	[CFA-8]	[CFA-16]		[CFA-48]	[CFA-40]	[CFA-32]	[CFA-24]
13:	rbp+0x10	[CFA-8]	[CFA-56]	[CFA-48]	[CFA-40]	[CFA-32]	[CFA-24]	
77:	rsp+0x08	[CFA-8]	[CFA-56]	[CFA-48]	[CFA-40]	[CFA-32]	[CFA-24]	
80:	rbp+0x10	[CFA-8]	[CFA-56]	[CFA-48]	[CFA-40]	[CFA-32]	[CFA-24]	
ab:	rsp+0x08	[CFA-8]	[CFA-56]	[CFA-48]	[CFA-40]	[CFA-32]	[CFA-24]	

[Slide 339]

```
func1:
    DW_CFA_def_cfa: RSP +8
    DW_CFA_offset: RIP -8
    DW_CFA_advance_loc: 1
    DW_CFA_def_cfa_offset: +16
    DW_CFA_offset: RBP -16
    DW_CFA_advance_loc: 6
    DW_CFA_def_cfa_register: RBP
    DW_CFA_advance_loc: 3
    DW_CFA_offset: R12 -24
    DW_CFA_offset: RBX -32
    DW_CFA_advance_loc: 57
    DW_CFA_def_cfa: RSP +8
```

```
func2:
    DW_CFA_def_cfa: RSP +8
    DW_CFA_offset: RIP -8
    DW_CFA_advance_loc: 1
    DW_CFA_def_cfa_offset: +16
    DW_CFA_offset: RBP -16
    DW_CFA_advance_loc: 3
    DW_CFA_def_cfa_register: RBP
    DW_CFA_advance_loc: 6
    DW_CFA_offset: R15 -24
    DW_CFA_offset: R14 -32
    DW_CFA_offset: R13 -40
    DW_CFA_advance_loc: 5
    DW_CFA_offset: R12 -48
    DW_CFA_advance_loc: 4
    DW_CFA_offset: RBX -56
    DW_CFA_advance_loc1: 100
    DW_CFA_remember_state:
    DW_CFA_def_cfa: RSP +8
    DW_CFA_advance_loc: 9
    DW_CFA_restore_state:
    DW_CFA_advance_loc: 43
    DW_CFA_def_cfa: RSP +8
```


Bibliography

- [AC71] FE Allen and J Cocke. *A catalogue of optimizing transformations*. 1971. URL: <https://www.clear.rice.edu/comp512/Lectures/Papers/1971-allen-catalog.pdf>.
- [AGT89] AV Aho, M Ganapathi, and SWK Tjiang. “Code generation using tree matching and dynamic programming”. In: *TOPLAS* 11.4 (1989), pp. 491–516. URL: <https://dl.acm.org/doi/pdf/10.1145/69558.75700>.
- [Arm] Arm Ltd. *ELF for the Arm 64-bit Architecture (AArch64)*. URL: <https://github.com/ARM-software/abi-aa/blob/main/aaelf64/aaelf64.rst> (visited on 11/21/2022).
- [Arm+22] N Armenatzoglou et al. “Amazon Redshift Re-invented”. In: *SIGMOD*. 2022.
- [BCS97] P Briggs, KD Cooper, and LT Simpson. *Value numbering*. Tech. rep. CRPC-TR94517-S. Rice University, 1997. URL: <https://www.cs.rice.edu/~keith/Promo/CRPC-TR94517.pdf.gz>.
- [BDB90] 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.
- [Bra+13] M Braun et al. “Simple and efficient construction of static single assignment form”. In: *CC*. 2013, pp. 102–122. URL: https://link.springer.com/content/pdf/10.1007/978-3-642-37051-9_6.pdf.
- [Cla+57] W Clark et al. *The Lincoln TX-2 Computer*. Apr. 1957. URL: http://www.bitsavers.org/pdf/mit/tx-2/TX-2_Papers_WJCC_57.pdf.
- [Cyt+91] R Cytron et al. “Efficiently computing static single assignment form and the control dependence graph”. In: *TOPLAS* 13.4 (1991), pp. 451–490. URL: <https://dl.acm.org/doi/pdf/10.1145/115372.115320>.
- [DF84] JW Davidson and CW Fraser. “Code selection through object code optimization”. In: *TOPLAS* 6.4 (1984), pp. 505–526. URL: <https://dl.acm.org/doi/pdf/10.1145/1780.1783>.
- [Die82] PF Dietz. “Maintaining order in a linked list”. In: *STOC*. 1982, pp. 122–127. URL: <https://dl.acm.org/doi/pdf/10.1145/800070.802184>.
- [DWA17] DWARF Debugging Information Committee. *DWARF Debugging Information Format Version 5*. Feb. 2017. URL: <http://dwarfstd.org/doc/DWARF5.pdf>.
- [ECG06] 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.

- [Ert99] MA Ertl. “Optimal code selection in DAGs”. In: *POPL*. 1999, pp. 242–249. URL: <https://dl.acm.org/doi/pdf/10.1145/292540.292562>.
- [Gar02] K Gargi. “A sparse algorithm for predicated global value numbering”. In: *PLDI*. 2002, pp. 45–56.
- [Geo05] L Georgiadis. “Linear-Time Algorithms for Dominators and Related Problems”. PhD thesis. Princeton University, Nov. 2005.
- [GG78] RS Glanville and SL Graham. “A new method for compiler code generation”. In: *POPL*. 1978, pp. 231–254. URL: <https://dl.acm.org/doi/pdf/10.1145/512760.512785>.
- [Gra94] G Graefe. “Volcano—an extensible and parallel query evaluation system”. In: *IEEE Transactions on Knowledge and Data Engineering* 6.1 (1994), pp. 120–135.
- [Hab13] J Haberman. *Parsing C++ is literally undecidable*. 2013. URL: <https://blog.reverberate.org/2013/08/parsing-c-is-literally-undecidable.html>.
- [Hav97] P Havlak. “Nesting of reducible and irreducible loops”. In: *TOPLAS* 19.4 (1997), pp. 557–567. URL: <https://dl.acm.org/doi/pdf/10.1145/262004.262005>.
- [HWZ14] D Herman, L Wagner, and A Zakai. *asm.js*. 2014. URL: <http://asmjs.org/spec/latest/>.
- [KG08] DR Koes and SC Goldstein. “Near-optimal instruction selection on DAGs”. In: *CGO*. 2008, pp. 45–54. URL: <http://llvm.org/pubs/2008-CGO-DagISel.pdf>.
- [KH11] R Karrenberg and S Hack. “Whole-function vectorization”. In: *CGO*. 2011, pp. 141–150.
- [Kud17] J Kuderski. “Dominator Trees and incremental updates that transcend times”. In: *LLVM Dev Meeting*. Oct. 2017. URL: https://llvm.org/devmtg/2017-10/slides/Kuderski-Dominator_Trees.pdf.
- [LA04] C Lattner and V Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *CGO*. 2004, pp. 75–86. URL: <http://www.llvm.org/pubs/2004-01-30-CGO-LLVM.pdf>.
- [LM69] ES Lowry and CW Medlock. “Object code optimization”. In: *CACM* 12.1 (1969), pp. 13–22. URL: <https://dl.acm.org/doi/pdf/10.1145/362835.362838>.
- [LT79] T Lengauer and RE Tarjan. “A fast algorithm for finding dominators in a flowgraph”. In: *TOPLAS* 1.1 (1979), pp. 121–141. URL: <https://dl.acm.org/doi/pdf/10.1145/357062.357071>.
- [Lu+22] HJ Lu et al. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*. 2022. URL: <https://gitlab.com/x86-psABIs/x86-64-ABI/-/jobs/artifacts/master/raw/x86-64-ABI/abi.pdf?job=build>.
- [McK65] WM McKeeman. “Peephole optimization”. In: *CACM* 8.7 (1965), pp. 443–444. URL: <https://dl.acm.org/doi/pdf/10.1145/364995.365000>.

-
- [Neu11] T Neumann. “Efficiently compiling efficient query plans for modern hardware”. In: *VLDB* 4.9 (2011), pp. 539–550.
- [OB11] J Oakley and S Bratus. “Exploiting the Hard-Working DWARF: Trojan and Exploit Techniques with No Native Executable Code”. In: *WOOT*. 2011. URL: https://www.usenix.org/events/woot11/tech/final_files/Oakley.pdf.
- [PM12] M Pharr and WR Mark. “ispc: A SPMD compiler for high-performance CPU programming”. In: *InPar*. 2012, pp. 1–13.
- [PS99] M Poletto and V Sarkar. “Linear scan register allocation”. In: *TOPLAS* 21.5 (1999), pp. 895–913.
- [Ram99] G Ramalingam. “Identifying loops in almost linear time”. In: *TOPLAS* 21.2 (1999), pp. 175–188. URL: <https://dl.acm.org/doi/pdf/10.1145/316686.316687>.
- [Roe] A Roenky. *ELF: better symbol lookup via DT_GNU_HASH*. URL: <https://flapenguin.me/elf-dt-gnu-hash> (visited on 12/14/2022).
- [SDK18] A Shaikhha, M Dashti, and C Koch. “Push versus pull-based loop fusion in query engines”. In: *Journal of Functional Programming* 28 (2018).
- [Seh+10] D Sehr et al. “Adapting Software Fault Isolation to Contemporary {CPU} Architectures”. In: *19th USENIX Security Symposium (USENIX Security 10)*. 2010.
- [Son] F Song. *Personal Blog*. URL: <https://maskray.me/> (visited on 11/21/2022).
- [Tar73] R Tarjan. “Testing flow graph reducibility”. In: *STOC*. 1973, pp. 96–107. URL: <https://dl.acm.org/doi/pdf/10.1145/800125.804040>.
- [THS98] O Traub, G Holloway, and MD Smith. “Quality and speed in linear-scan register allocation”. In: *SIGPLAN* 33.5 (1998), pp. 142–151. URL: <https://dl.acm.org/doi/pdf/10.1145/277652.277714>.
- [Vel03] TL Veldhuizen. *C++ templates are Turing complete*. 2003. URL: <http://port70.net/~nsz/c/c%2B%2B/turing.pdf>.
- [WF10] C Wimmer and M Franz. “Linear scan register allocation on SSA form”. In: *CGO*. 2010, pp. 170–179.
- [WM05] C Wimmer and H Mössenböck. “Optimized interval splitting in a linear scan register allocator”. In: *VEE*. 2005, pp. 132–141.
- [Yee+09] B Yee et al. “Native client: A sandbox for portable, untrusted x86 native code”. In: *SP*. 2009, pp. 79–93.