Code Generation for Data Processing

Alexis Engelke

Winter 2025/26

Contents

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/ws2526/codegen/`

Exam

- Written exam, 90 minutes, **no retake**, 2026-02-27 14:00
- (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

> - Grade $*$: excellent submission, goes beyond expectations
> - Grade $+$: good submission, meets expectations
> - Grade $\sim$: acceptable submission, but is below expectations
> - Grade $-$: unacceptable submission, too far below expectations

- Exercises integrated into lecture
- Hands-on programming or analysis of systems (needs laptop)
- Occasionally: present and discuss homework solutions

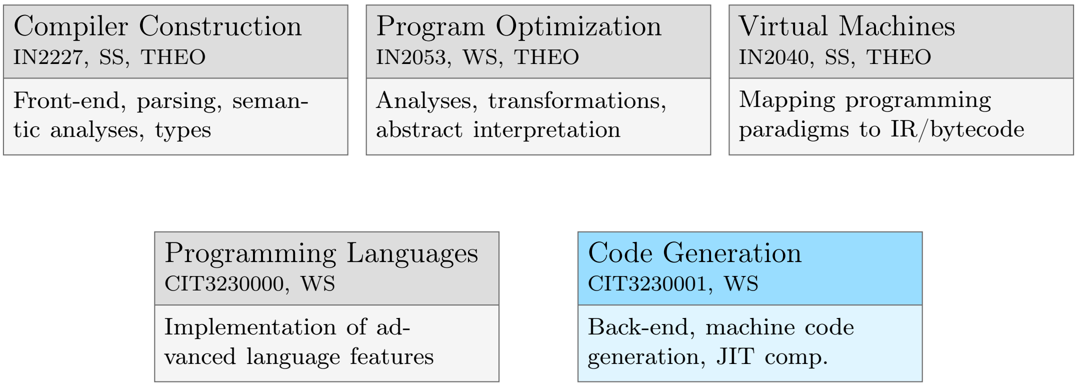
Grade Bonus

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

[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
Front-end, parsing, semantic analyses, types

Program Optimization
IN2053, WS, THEO
Analyses, transformations, abstract interpretation

Virtual Machines
IN2040, SS, THEO
Mapping programming paradigms to IR/bytecode

Programming Languages
CIT3230000, WS
Implementation of advanced language features

Code Generation
CIT3230001, WS
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

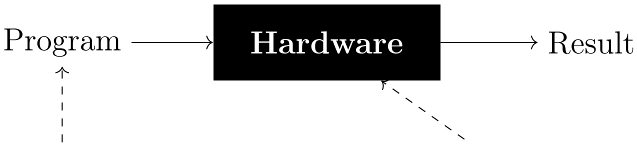
[Slide 10] Motivational Example: Brainfuck

- 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



Program

Hardware

Result

Programs

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

Hardware/ISA

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

[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

Program → Compiler → Program

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

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

Interpreter

Program → Interpreter → Result

- Directly execute program
- Computes program result

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

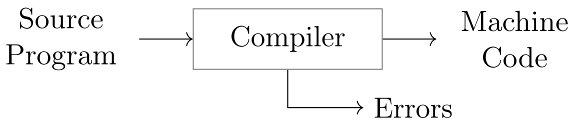
Multiple compilation steps can precede the "final interpretation"

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 → machine code
  - JIT compilation of JavaScript, WebAssembly, Java bytecode, . . .
  - Database query compilation
  - ISA emulation/binary translation

[Slide 15] Compiler Structure: Monolithic

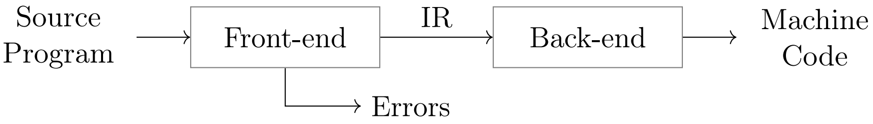

Source Program → Compiler → Machine Code

Errors

- 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

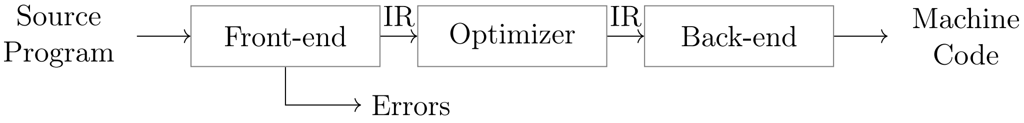
- Parses source code
- Detect syntax/semantical errors
- Emit *intermediate representation* encode semantics/knowledge
- Typically: $\mathcal{O}(n)$ or $\mathcal{O}(n \log n)$

Back-end

- Translate IR to target architecture
- Can assume valid IR ($\rightsquigarrow$ 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. $\mathcal{R}e$
   - Example: `a+b*c` → `ID(a) PLUS ID(b) TIMES ID(c)`
2. Parser: build (abstract) syntax tree, check for syntax errors $\mathcal{CFG}$
   - Syntax Tree: describe grammatical structure of complete program Example: `expr("a", op("+"), expr("b", op("*"), expr("c"))`
   - Abstract Syntax Tree: only relevant information, more concise Example: `plus("a", times("b", "c"))`
3. Semantic Analysis: check types, variable existence, etc.
4. IR Generator: produce IR for next stage
   - This might be the AST itself

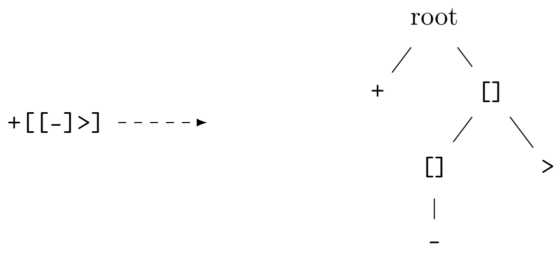
[Slide 19] Compiler Back-end

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?

Solution on page 191.

[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 ⇝ 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] Threaded Interpretation[1]

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

```c
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 branches and use this for better prediction. On such processors, threaded interpretation might not improve performance (or gains might be lower).

[1] MA Ertl and D Gregg. "The structure and performance of efficient interpreters". In: *JILP* 5 (2003), pp. 1–25. URL: http://www.jilp.org/vol5/v5paper12.pdf.

[Slide 26] Threaded Interpretation with Tail Calls

- Threaded interpretation can also be implemented with tails calls

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

> Computed `goto` is a GNU extension and similar features are not available in many other languages. However, it is possible to achieve a similar effect through indirect tail calls[a].
>
> In this example[b], the resulting assembly is almost identical for x86-64, but for AArch64, some more engineering effort would be required.
>
> The code as written has another problem: At `-O0`, no tail call optimization will be performed. Some compilers like Clang provide a way to force tail calls in C/C++ code:
>
> ```
> void tcInterp(struct op* pc, struct state* s) {
>   // ... (and adapt individual functions likewise)
>   [[clang::musttail]] return fns[pc->op](pc, s);
> }
> ```
>
> In the context of functional programming, this style is also referred to as *continuation-passing style*.
>
> [a] A tail call is a call at the end of a function without adding a new entry to the stack frame.
>
> [b] `https://godbolt.org/z/E5drG61aK`

[Slide 27] Direct Threading[2]

- Use function pointer as operation to avoid indirection

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

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

[2] JR Bell. “Threaded Code”. In: *CACM* 16.6 (1973), pp. 370–372. URL: `https://dl.acm.org/doi/pdf/10.1145/362248.362270`.

> While there might be a benefit of reducing the instruction count, the effect is typically insignificant on out-of-order CPUs, as the extra load instruction is typically a cache hit and the latency is hidden by other operations.

[Slide 28] Threaded Interpretation — Comparison

In-Class Exercise:

What are benefits/drawbacks of the three threading approaches[a]?

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

Solution on page 191.

[a] `https://godbolt.org/z/4rcEv4sqj`

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

[Slide 29] 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 ⇝ fewer instructions
  - Larger operations preferable, but not too many (prediction)
- Perhaps optimize – if it's worth the benefit
  - Fold constants, combine instructions, …
  - Consider superinstructions for common sequences
- For very cold code: avoid transformations at all

[Slide 30] Interpretation vs. Compilation

Fundamental benefits of compilation:

- Elimination of interpreter dispatch
  - Can be significant for bytecodes with many small operations
- Use of CPU registers for values across bytecode instructions
  - Interpreter: most values must be stored in memory
  - Register allocation can bring large improvements

  > This is a major driver for performance even on modern CPUs, which have two related optimizations: Store-to-load forwarding, where the value from a store will be forwarded to a later load from the same address; and memory renaming,

> where the physical register of a stored value is kept and a subsequent load with a similar address operand speculatively reuses that register. However, CPU resources for these optimizations are limited and using registers directly is much more efficient.

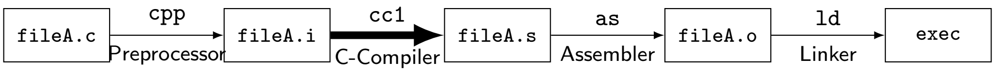
- No fundamental benefit: optimizations
  - Many optimizations can also be applied on bytecode

Fundamental benefits of interpretation: simple, portable

1.5. Context of Compilation

[Slide 31] 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 32] 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 33] 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?

Solution on page 191.

[Slide 34] 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 35] Introduction and Interpretation – Summary

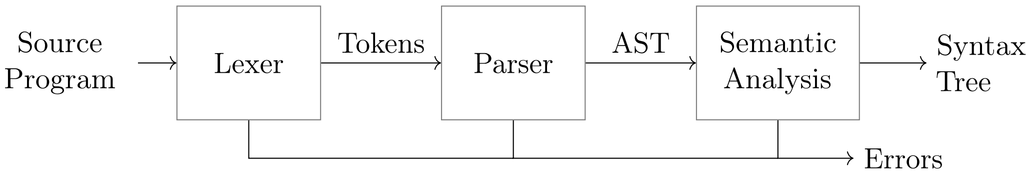
- Compilation vs. interpretation and combinations
- Compilers are key to usable/performant languages
- Target language typically machine code or bytecode
- Three-phase architecture widely used
- Interpretation techniques: bytecode, threaded interpretation, ...
- JIT compilation imposes different constraints

[Slide 36] Introduction and Interpretation – Questions

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

2. Compiler Front-end

[Slide 38] Compiler Front-end


Source Program

Lexer

Tokens

Parser

AST

Semantic Analysis

Syntax Tree

Errors

- 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 39] 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 40] 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 41] 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 identifer, 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 42] 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 43] Lexer Implementation

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

> 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 44] Parsing

- Convert stream of tokens into (abstract) syntax tree
- Most programming languages are context-sensitive
  - Variable declarations, argument count, type match, etc. ⇝ separated into semantic analysis

  Syntactically valid: `void foo = doesntExist / "abc";`
- Grammar usually specified as CFG

[Slide 45] Context-Free Grammar (CFG)

- Terminals: basic symbols/tokens
- Non-terminals: syntactic variables

- Start symbol: non-terminal defining language
- Productions: non-terminal → series of (non-)terminals

$$\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \mathit{whileStmt} \mid \mathit{breakStmt} \mid \mathit{exprStmt} \\
\mathit{whileStmt} & \rightarrow & \textbf{while}\ \textbf{(}\ \mathit{expr}\ \textbf{)}\ \mathit{stmt} \\
\mathit{breakStmt} & \rightarrow & \textbf{break}\ \textbf{;} \\
\mathit{exprStmt} & \rightarrow & \mathit{expr}\ \textbf{;} \\
\mathit{expr} & \rightarrow & \mathit{expr} + \mathit{expr} \mid \mathit{expr} * \mathit{expr} \mid \mathit{expr} = \mathit{expr} \mid \textbf{(}\ \mathit{expr}\ \textbf{)} \mid \textbf{number}
\end{array}$$

[Slide 46] 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 47] Hand-written Parsing – Second Try

- Need look-ahead to distinguish production rules
- Consequences for grammar:
  - No left-recursion
  - First n terminals must allow distinguishing rules
  - $LL(n)$ grammar; n typically 1

  ⇒ Not all CFGs (easily) parseable (but most programming langs. are)
- Now... expressions

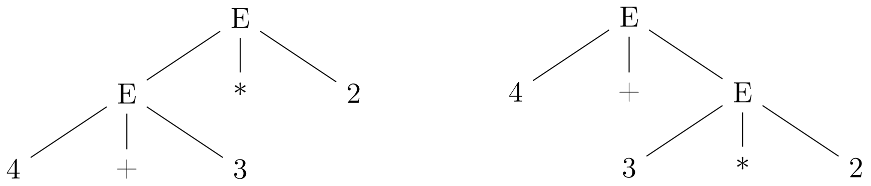
```
NodePtr parseBreakStmt() { /*...*/ }
NodePtr parseWhileStmt() { /*...*/ }

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

[Slide 48] Ambiguity

$$expr \rightarrow expr + expr \mid expr \text{ * } expr \mid expr = expr \mid (\ expr\ ) \mid \textbf{number}$$

Input: $4 + 3 * 2$



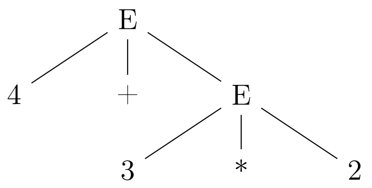
> The grammar, as specified, is ambiguous, there are two possible ways to parse the input.

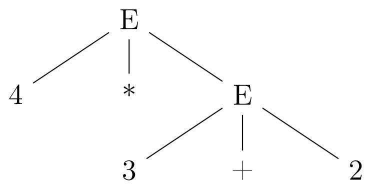
[Slide 49] Ambiguity – Rewrite Grammar?

$$\begin{aligned} primary &\rightarrow (\ expr\ ) \mid \textbf{number} \\ expr &\rightarrow primary + expr \mid primary \text{ * } expr \mid primary = expr \mid primary \end{aligned}$$

Input: $4 + 3 * 2$

Input: $4 * 3 + 2$

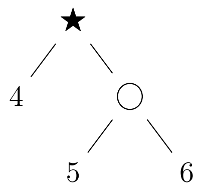


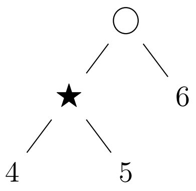


> The grammar is no longer ambiguous, but the result might not be expected, conventionally, multiplication has a stronger binding than addition.

[Slide 50] Ambiguity – Precedence

Input: $4 \bigstar 5 \bigcirc 6$





If $\mathit{prec}(\bigcirc) > \mathit{prec}(\bigstar)$ or equal prec. and ★ is right-assoc.

> Examples:
>
> - $4 + 5 \cdot 6$ ($\mathit{prec}(\cdot) > \mathit{prec}(+)$)
> - $a = b = c$ ($=$ is right-assoc.)
> $b = c$ should be executed first

If $\mathit{prec}(\bigcirc) < \mathit{prec}(\bigstar)$ or equal prec. and ★ is left-assoc.

> Examples:
>
> - $4+5 < 6$ ($\mathit{prec}(<) < \mathit{prec}(+)$)
> - $a + b - c$ ($+$ is left-assoc.)
> $a + b$ should be executed first

[Slide 51] Hand-written Parsing – Expression Parsing

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

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

[Slide 52] Hand-written Parsing – Expression Parsing

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

```
OpDesc OPS[] = { // {prec, rassoc}
  [Token::MUL] = {12, false},
  [Token::ADD] = {11, false},
  [Token::EQ] = {2, true},
  [Token::QUEST] = {3, true}, // ?:
}
NodePtr parseExpr(unsigned minPrec=1) {
  auto lhs = parsePrimaryExpr();
  while (auto op = OPS[next().kind];
        op.prec >= minPrec) {
    // ... handle (, [, ?: ...
    auto newPrec = op.rassoc ?
      op.prec : op.prec + 1;
    auto rhs = parseExpr(newPrec);
    lhs = makeNode(op.nodeKind,
      {lhs, rhs});
  }
  return lhs;
}
```

In-Class Exercise:

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

Solution on page 192.

[Slide 53] 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 54] 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 55] 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 56] 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 he parser.

[Slide 57] 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) ompilation.
>
> [a] `https://notes.eatonphil.com/parser-generators-vs-handwritten-parsers-survey-2021.html`

[Slide 58] Parsing C++

- C++ is not context-free (inherited from C): `T * a;`
- C++ is ambiguous: `Type (a), b;`
  - Can be a declaration or a comma expression
- C++ templates are Turing-complete[1]
- C++ *parsing* is hence *undecidable*[2]
  - Template instantiation combined with C `T * a` ambiguity

2.3. Semantic Analysis

[Slide 59] Semantic Analysis

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

[Slide 60] 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?

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

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

Solution on page 192.

- 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

[Slide 61] 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 62] 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 63] 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 64] 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 65] Recommended Lectures

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

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

[Slide 66] 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 67] 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

3.1. Motivation

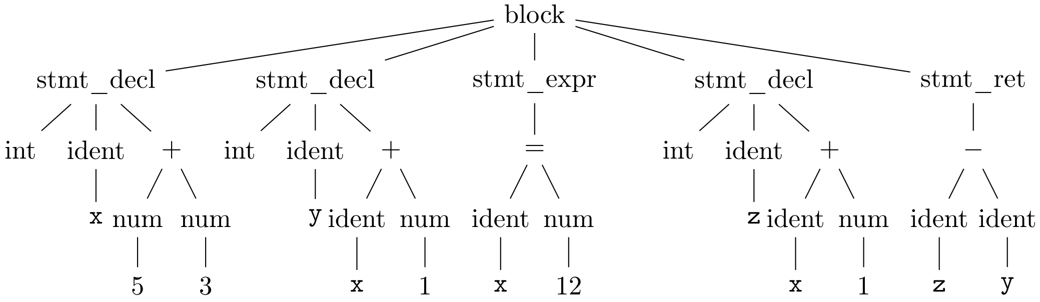
[Slide 69] Intermediate Representations: Motivation

- 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 70] Intermediate Representations: Motivation



Question: how to optimize? Is `x+1` redundant? ⇝ 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 71] Intermediate Representations: Motivation

$$
\begin{array}{llccc}
x_1 & \leftarrow & 5 & + & 3 \\
y_1 & \leftarrow & x_1 & + & 1 \\
x_2 & \leftarrow & 12 & & \\
z_1 & \leftarrow & x_2 & + & 1 \\
tmp_1 & \leftarrow & z_1 & - & y_1 \\
\text{return} & & tmp_1 & &
\end{array}
$$

Question: how to optimize? Is `x+1` redundant? ⇝ No! ☺

[Slide 72] 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 73] 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 74] Compiler Design: Effect of Languages – Declarative

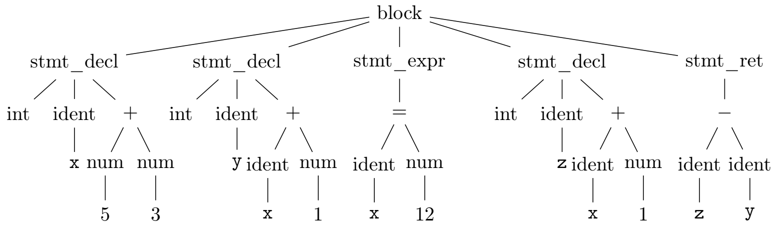
- Describes execution target
- Compiler has to derive good mapping to imperative hardware

- 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 75] Graph IRs: Abstract Syntax Tree (AST)

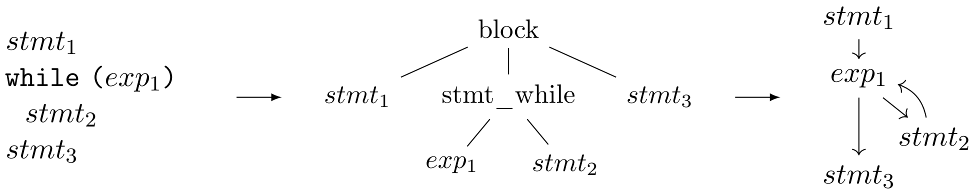
- Code representation close to the source
- Representation of types, constants, etc. might differ
- Storage might be problematic for large inputs



3.2. Control Flow Graph

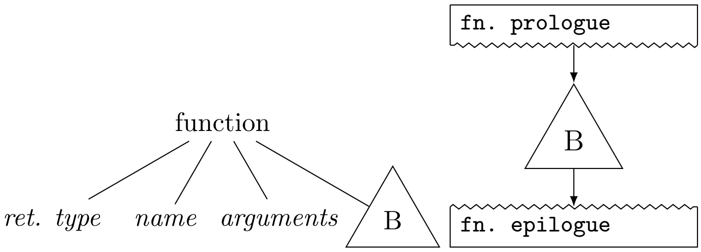
[Slide 76] Graph IRs: Control Flow Graph (CFG)

- 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

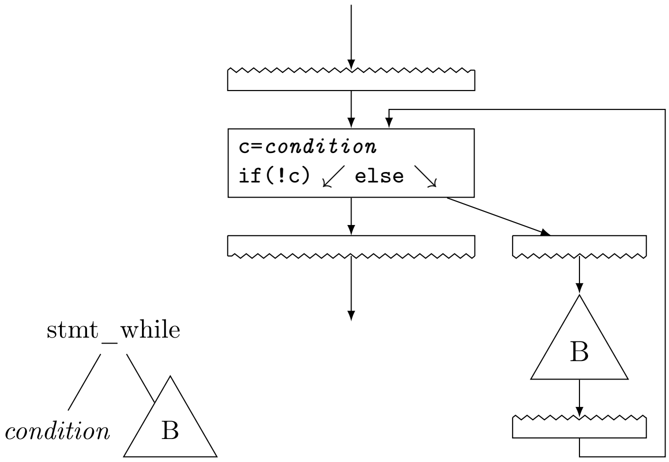


[Slide 77] Build CFG from AST – Function

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



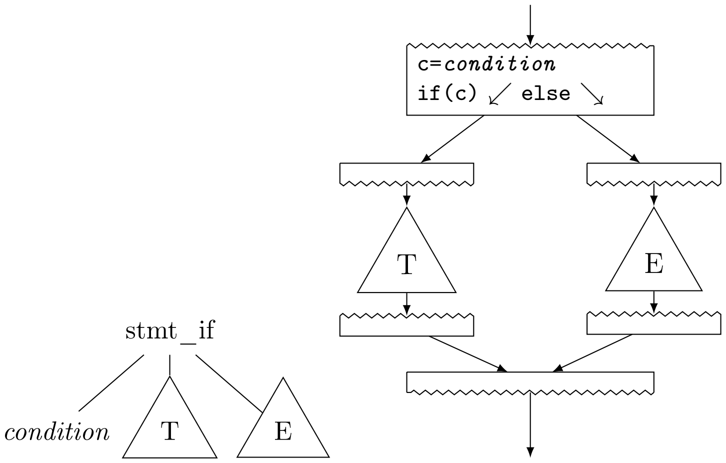
[Slide 78] Build CFG from AST – While Loop



> 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 79] Build CFG from AST – If Condition



[Slide 80] Build CFG from AST: Switch

Linear search

```
t ← exp
if t == 3: goto $B_3$
if t == 4: goto $B_4$
if t == 7: goto $B_7$
if t == 9: goto $B_9$
goto $B_D$
```

+ Trivial
– Slow, lot of code

Binary search

```
t ← exp
if t == 7: goto $B_7$
elif t > 7:
  if t == 9: goto $B_9$
else:
  if t == 3: goto $B_3$
  if t == 4: goto $B_4$
goto $B_D$
```

+ Good: sparse values
– Even more code

Jump table

```
t ← exp
if $0 \le$ t $< 10$:
  goto table[t]
goto $B_D$

table = {
  $B_D$, $B_D$, $B_D$, $B_3$,
  $B_4$, $B_D$, ... }
```

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

> Typically, it is beneficial to keep switches in a high-level form during optimizations, as they are more expressive than, e.g., a corresponding search tree. The corresponding node in the CFG has then one outgoing edge per case plus the edge for the default target.

[Slide 81] Build CFG from AST: Break, Continue, Goto, Computed Goto

- break/continue: trivial

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

> Very dense CFGs are problematic for the runtime of many graph algorithms — many algorithms need at the very least to consider every edge, so the runtime of such algorithms grows by at least $\Omega(|N|^2)$, often more. Typically, therefore, only a single computed `goto` is emitted into the IR as a separate basic block; all computed `goto`s in the code are re-routed to go through this new block. Only very late in the code generation pipeline, this block is duplicated again into all its predecessors to produce the expected behavior. (Note that, due to compiler bugs, this might not always happen.)

[Slide 82] 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 83] 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();
  }
  return g();
}
```

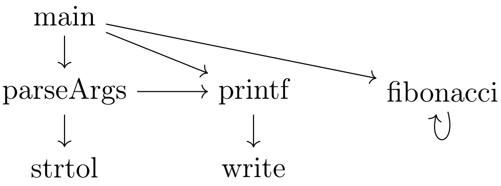
```
int fn2() {
  a();
  do switch (c()) {
  case 1:
    while (d()) {
      e();
    case 2:
      f();
    }
  default:
    g();
  } while (h());
  return b();
}
```

Solution on page 193.

3.3. Other Graph IRs

[Slide 84] Graph IRs: Call Graph

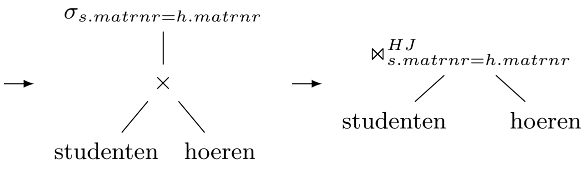
- Graph showing (possible) call relations between functions
- Useful for interprocedural optimizations
  - Function ordering
  - Stack depth estimation
  - ...



[Slide 85] 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.

```
SELECT s.name, h.vorlnr
FROM studenten s, hoeren h
WHERE s.matrnr = h.matrnr
```


$\sigma_{s.matrnr=h.matrnr}$
×
studenten
hoeren
$\bowtie^{HJ}_{s.matrnr=h.matrnr}$
studenten
hoeren

3.4. Linear IRs

[Slide 86] 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 87] Linear IRs: Register Machines

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

$$\begin{array}{lllll}
x & \leftarrow & 5 & + & 3 \\
y & \leftarrow & x & + & 1 \\
x & \leftarrow & 12 & & \\
z & \leftarrow & x & + & 1 \\
tmp_1 & \leftarrow & z & - & y \\
\text{return} & & tmp_1 & &
\end{array}$$

[Slide 88] 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 89] 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 90] 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 91] 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

$$\begin{array}{lllll}
x & \leftarrow & 5 & + & 3 \\
y & \leftarrow & x & + & 1 \\
x & \leftarrow & 12 & & \\
z & \leftarrow & x & + & 1 \\
tmp_1 & \leftarrow & z & - & y \\
\text{return} & & tmp_1 & &
\end{array}$$

3.5. Single Static Assignment

[Slide 92] 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

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

[Slide 93] 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)

$$
\begin{array}{rl}
\mathit{entry}: & x \;\leftarrow\; \ldots \\
 & \text{if } (x > 2) \text{ goto } \mathit{cont} \\
\mathit{then}: & x \;\leftarrow\; x \,*\, 2 \\
\mathit{cont}: & \text{return}\;\; x
\end{array}
$$

$$\longrightarrow$$

$$
\begin{array}{rl}
\mathit{entry}: & v_1 \;\leftarrow\; \ldots \\
 & \text{if } (v_1 > 2) \text{ goto } \mathit{cont} \\
\mathit{then}: & v_2 \;\leftarrow\; v_1 \,*\, 2 \\
\mathit{cont}: & v_3 \;\leftarrow\; \Phi(\mathit{entry}: v_1, \mathit{then}: v_2) \\
 & \text{return}\;\; v_3
\end{array}
$$

[Slide 94] 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 95] SSA Construction – Local Value Numbering

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

Code

$$
\begin{array}{lllll}
x & \leftarrow & 5 & + & 3 \\
y & \leftarrow & x & + & 1 \\
x & \leftarrow & 12 & & \\
z & \leftarrow & x & + & 1 \\
\mathit{tmp}_1 & \leftarrow & z & - & y \\
\text{return} & & \mathit{tmp}_1 & &
\end{array}
$$

SSA IR

$$
\begin{array}{lll}
v_1 & \leftarrow & \texttt{add } 5,\ 3 \\
v_2 & \leftarrow & \texttt{add } v_1,\ 1 \\
v_3 & \leftarrow & \texttt{const } 12 \\
v_4 & \leftarrow & \texttt{add } v_3,\ 1 \\
v_5 & \leftarrow & \texttt{sub } v_4,\ v_2 \\
 & & \texttt{ret } v_5
\end{array}
$$

Variable Mapping

$$
\begin{array}{rcl}
x & \rightarrow & v_3 \\
y & \rightarrow & v_2 \\
z & \rightarrow & v_4 \\
\mathit{tmp}_1 & \rightarrow & v_5
\end{array}
$$

[Slide 96] 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? ⇒ *don't do this!*
  - *Extremely inefficient, code size explosion, many dead Φ*

[Slide 97] 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 98] SSA Construction – Example

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

[1] 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`.

```
    n -= 1;
  }
  return res;
}
```

```
            func foo ($v_1$)
  entry:    sealed; varmap: n→ $v_1$, res→ $v_2$
            $v_2 \leftarrow 1$
 header:    sealed; varmap: n→ $\phi_1$, res→ $\phi_2$
            $\phi_1 \leftarrow \phi$(entry: $v_1$, body: $v_6$)
            $\phi_2 \leftarrow \phi$(entry: $v_2$, body: $v_5$)
            $v_3 \leftarrow$ equal $\phi_1$, 0
            br $v_3$, cont, body
   body:    sealed; varmap: n→$v_6$, res→ $v_5$
            $v_4 \leftarrow$ mul $\phi_1$, $\phi_1$
            $v_5 \leftarrow$ mul $\phi_2$, $v_4$
            $v_6 \leftarrow$ sub $\phi_1$, 1
            br header
   cont:    sealed; varmap: res→ $\phi_2$
            ret $\phi_2$
```

> Note: the *sealed* state and the variable map are only used during SSA construction. They can be discarded afterwards and are *not* part of the IR. The information is also not sufficient for debug information, for that purpose as it doesn't sufficiently cover all program points; tracking variable states for debug information typically requires extra instructions or annotations in the IR.

[Slide 99] SSA Construction – Example

In-Class Exercise:

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

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

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

Solution on page 193.

[Slide 100] 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 algorithms[2] or different construction method[3]

AD IN2053 “Program Optimization” covers this more formally

[Slide 101] SSA: Implementation

- Value is often just a reference to the 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

3.6. IR Design

[Slide 103] 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

[2] 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`.

[3] 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`.

[Slide 104] 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 105] 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 106] IR Design: Desired Operations on IR

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

[Slide 107] 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 108] 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 109] Intermediate Representations – Summary

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

[Slide 110] 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 112] LLVM[1]

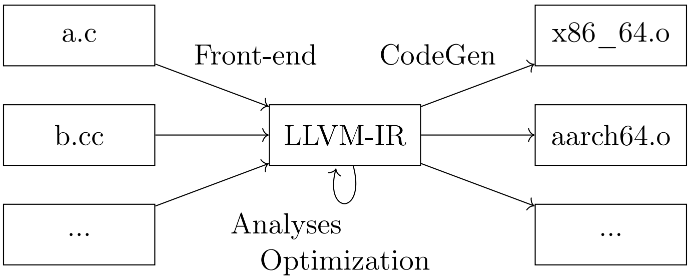
LLVM "Core" Library

- 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 113] 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

[1] 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`.

> 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 114] 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 115] 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 116] 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 117] LLVM-IR: Functions

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

[Slide 118] 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 119] 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 120] 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 121] LLVM-IR: Instructions – Memory and Pointer

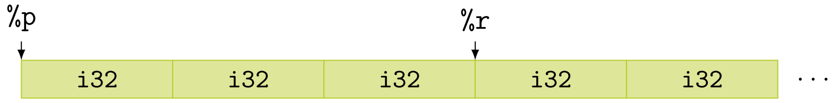
- `alloca <ty>` – allocate addressable stack slot
- `load <ty>, ptr <ptr>`/`store <ty> <val>, ptr <ptr>`
  - May be `volatile` (e.g., MMIO) and/or `atomic`
- `cmpxchg`/`atomicrmw` – similar to hardware operations
- `ptrtoint`/`inttoptr`
- `getelementptr` – address computation on `ptr`/structs/arrays

> `alloca` allocates stack memory whenever it is executed. There is a difference between statically-sized `alloca`s that are in the entry block versus in dynamically-sized `alloca`s or `alloca`s in other blocks:

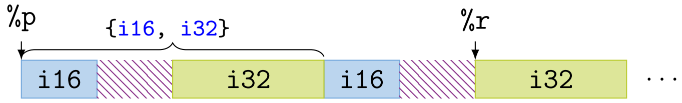
> - Static `alloca`s become part of the fixed-size stack frame.
> - Dynamic `alloca`s require a dynamically-sized stack frame, which is less efficient (use of a frame pointer is required) and can cause unbounded stack growth.

[Slide 122] LLVM-IR: `getelementptr` Examples

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

  Equivalent in C: `&((int*) p)[3]`
- `%r = getelementptr {i16, i32}, ptr %p, i64 1, i32 1`

  Equivalent in C: `&((struct {short _0; int _1;}*) p)[1]._1`
- Also works with nested structs and arrays (and vectors)
- Pointer and array can be dynamic, struct indices must be constant

[Slide 123] 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 124] 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 125] 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

> Note: `-emit-llvm` emits the LLVM IR after optimizations. To get the IR before optimizations, use `-Xclang -disable-llvm-passes`. The LLVM IR generated when compiling with `-O0` is different.

[Slide 126] LLVM-IR: Example

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

[Slide 127] LLVM-IR: Example

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

[Slide 128] LLVM-IR: Example

In-Class Exercise:

```
define i32 @sw(i32 %0) {
  switch i32 %0, label %4 [
    i32 4, label %5
    i32 5, label %2
    i32 8, label %3
    i32 100, label %5
  ]
2: ; preds = %1
  br label %5
3: ; preds = %1
  br label %5
4: ; preds = %1
  br label %5
5: ; preds = %1, %1, %4, %3, %2
```

```
  %6 = phi i32 [ %0, %4 ], [ 9, %3 ], [ 32, %2 ], [ 12, %1 ], [ 12, %1 ]
  ret i32 %6
}
```

Solution on page 194.

[Slide 129] 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
}
```

Solution on page 194.

4.3. API

[Slide 130] 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

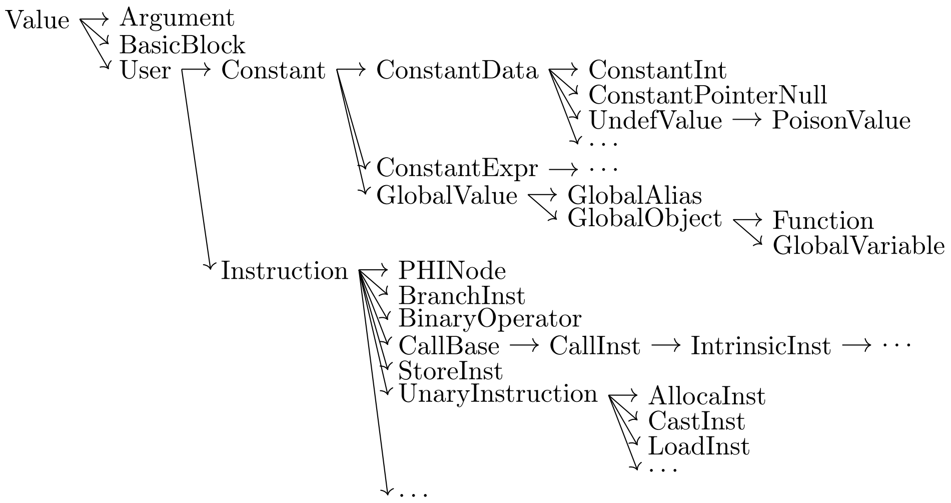
[Slide 131] LLVM-IR C++ API: Basic Example

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



Value
Argument
BasicBlock
User
Constant
ConstantData
ConstantInt
ConstantPointerNull
UndefValue → PoisonValue
...
ConstantExpr → ...
GlobalValue
GlobalAlias
GlobalObject
Function
GlobalVariable
Instruction
PHINode
BranchInst
BinaryOperator
CallBase → CallInst → IntrinsicInst → ...
StoreInst
UnaryInstruction
AllocaInst
CastInst
LoadInst
...
...

> See LLVM Doxygen[a] for a full graph.
>
> [a] https://llvm.org/doxygen/classllvm_1_1Value.html

[Slide 133] LLVM-IR API: Programming Environment

- LLVM implements custom RTTI
  - `isa<>`, `cast<>`, `dyn_cast<>`
- LLVM implements a multitude of specialized data structures
  - E.g.: `SmallVector<T, N>` to keep N elements stack-allocated
  - Custom vectors, sets, maps; see manual[2]
- Preferably uses `ArrayRef`, `StringRef`, `Twine` for references
- LLVM implements custom streams instead of std streams
  - `outs()`, `errs()`, `dbgs()`

[2] https://www.llvm.org/docs/ProgrammersManual.html

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. The sub-class type of the value is stored in an 8-bit integer.

`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 (`realloc`).

`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.

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 134] 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 135] 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

[Slide 136] 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?

5. Analyses and Transformations

5.1. Motivation

[Slide 138] Program Transformation: Motivation

- “User code” is often not very efficient
- Also: no need to, compiler can (often?) optimize better
  - More knowledge: e.g., data layout, constants after inlining, etc.
- Allows for more pragmatic/simple code
- Generating “better” IR code on first attempt is expensive
  - What parts are actually used? How to find out?
- Transformation to “better” code must be done *somewhere*

- Optimization is a misnomer: we don’t know whether it improves code!
  - Many transformations are driven by heuristics
- Many types of optimizations are well-known[1]

5.2. Dead Code Elimination

[Slide 139] 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 140] 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
```

[1] FE 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 141] 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 142] 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 143] Dead Code Elimination

- Problem: unused value cycles

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

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

[Slide 144] 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 pf %b from the previous example. (Refer to the slide deck for the animated version.)

5.3. Sparse Conditional Constant Propagation

[Slide 145] Optimization Example 2

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

[Slide 146] Sparse Conditional Constant Propagation[2]

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

[Slide 147] SCCP Algorithm

State:

- EdgeWorklist
- InstrWorklist
- EdgeExecutable
  - Initialize `false`
- InstrLattice
  - Initialize $\top$

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

[Slide 148] SCCP Algorithm II

visit

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

[2] MN Wegman and FK Zadeck. "Constant propagation with conditional branches". In: *TOPLAS* 13.2 (1991), pp. 181–210. URL: `https://dl.acm.org/doi/pdf/10.1145/103135.103136`.

[Slide 149]

In-Class Exercise:

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

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

In-Class Exercise:

Code attribution: Adapted from `llvm/test/Transforms/SCCP/sccptest.ll` from the LLVM Project, licensed under Apache-2.0 WITH LLVM-Exception.

Solution on page 194.

[Slide 150] SCCP: Misc

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

5.4. Simple Common Subexpression Elimination

[Slide 151] Optimization Example 3

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

[Slide 152] 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?

> This algorithm only works for chains of blocks, but not for merging control flow. It is therefore rarely useful, see later.

[Slide 153] CSE Attempt 1 – Example 1

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

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

[Slide 154] 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!
```

5.5. Dominator Tree

[Slide 155] Domination

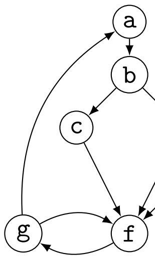
- Remember: CFG $G = (N, E, s)$ with digraph (N, E) and entry $s \in N$
- Dominate: d dom n iff every path from s to n contains d
  - Dominators of n: $DOM(n) = \{d | d \text{ dom } n\}$
- Strictly dominate: d sdom $n \Leftrightarrow d$ dom $n \land d \neq n$
- Immediate dominator: idom $(n) = d : d$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

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

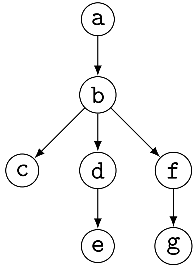
[Slide 156] Dominator Tree

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

Control Flow Graph



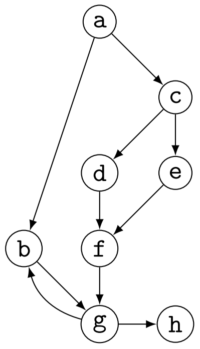
Dominator Tree

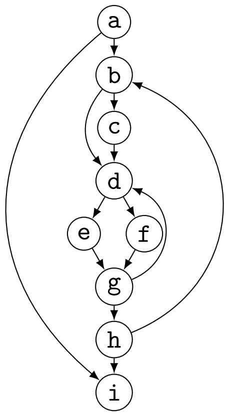


[Slide 157] Dominator Tree – Example

In-Class Exercise:

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





Solution on page 195.

[Slide 158] Dominator Tree: Construction

- Naive: inefficient (but reasonably simple)[3]
  - For each block: find a path from the root – superset of dominators
  - Remove last block on path and check for alternative path
  - If no alternative path exists, last block is idom
- Lengauer–Tarjan: more efficient methods[4]
  - Simple method in $\mathcal{O}(m \log n)$; sophisticated method in $\mathcal{O}(m \cdot \alpha(m, n))$ ($\alpha(m, n)$ is the inverse Ackermann function, grows *extremely* slowly)
  - Used in some compilers[5]
- Semi-NCA: $\mathcal{O}(n^2)$, but lower constant factors[6]

> 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].
>
> [a] 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`.

[Slide 159] 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

[3] 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`.

[4] 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`

[5] Example: `https://github.com/WebKit/WebKit/blob/aabfacb/Source/WTF/wtf/Dominators.h`

[6] L Georgiadis. "Linear-Time Algorithms for Dominators and Related Problems". PhD thesis. Princeton University, Nov. 2005. URL: `https://www.cs.princeton.edu/techreports/2005/737.pdf`

- Check whether a sdom b[7]
  - $a.preNum < b.preNum \;\wedge\; a.postNum > b.postNum$
  - After updates, numbers might be invalid: recompute or walk tree
- Problem: dominance of unreachable blocks ill-defined $\leadsto$ special handling

5.6. Common Subexpression Elimination

[Slide 160] CSE Attempt 2

- Option 1:
  - For identical instructions, store all
  - Add dominance check before replacing
  - Visit nodes in reverse post-order (i.e., topological order)
- Option 2 (Global Value Numbering):[8]
  - Do a DFS over dominator tree
  - Use scoped hashmap to track available values

Does this work? Yes.

[Slide 161] CSE: Hashing an Instruction (and Beyond)

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

[Slide 162] More Complex Global Value Numbering

- Hash-based approach only catches trivially removable duplicates
- Alternative: partition values into *congruence classes*
  - Congruent values are guaranteed to always have the same value
- Optimistic approach: values are congruent unless proven otherwise
- Pessimistic approach: values are not congruent unless proven
- Combinable with: reassociation, DCE, constant folding
- Rather complex, but can be highly beneficial[9]

[7] 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`.

[8] 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`.

[9] K Gargi. "A sparse algorithm for predicated global value numbering". In: *PLDI*. 2002, pp. 45–56.

5.7. Inlining

[Slide 163] Inlining

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

[Slide 164] Inlining: Decision

- First: determine whether function is inlinable

  > Not all functions are inlinable. For example, in LLVM, functions with computed `goto` cannot be inlined, because basic blocks whose address was taken cannot be duplicated. Another example functions compiled from different languages with different exception handling mechanisms (the exception handling routine needs language-specific metadata, which in general cannot be merged; see later).

- Easy way: delegate to programmer (`__attribute__((always_inline))`)

  > This attribute is, in fact, considered as somewhat problematic, because it is not just a very strong inlining hint, but guarantees inlining even with optimizations disabled. Therefore, even at `-O0`, compilers must execute an optimization pass to specifically handle functions annotated with this attribute.

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

> Typical heuristics are based on somewhat arbitrary thresholds, which are typically higher for higher optimization levels. There’re also approaches to use machine learning

> to guide inlining decisions[a].
>
> [a] M Trofin et al. *MLGO: a Machine Learning Guided Compiler Optimizations Framework.* 2021. arXiv: `2101.04808 [cs.PL]`. URL: `https://arxiv.org/abs/2101.04808`.

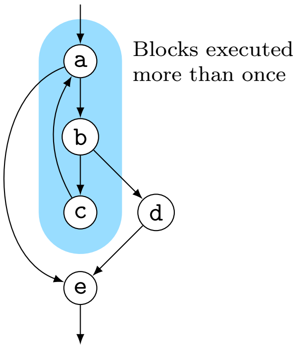
[Slide 165] Inlining Transformation

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

5.8. Loop Analysis

[Slide 166] What is a Loop?

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



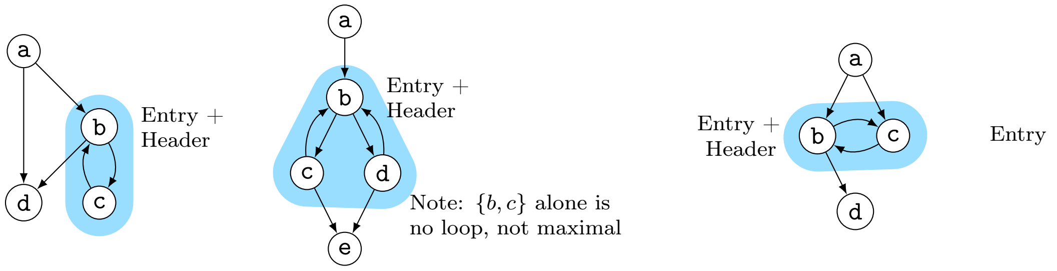
- Loops in source code ≠ loops in CFG
- `d` is *not* part of loop: executed at most once

⇝ Need algorithm to find loops in CFG

[Slide 167] Loops

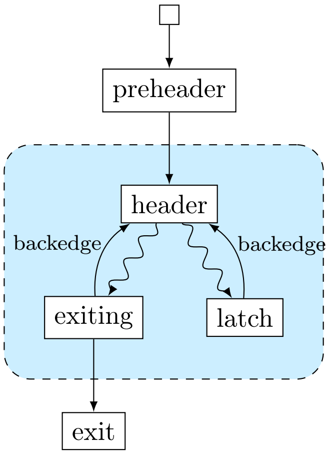
- Loop: maximal SCC L with at least one internal edge[10] (strongly connected component (SCC): all blocks reachable from each other)
  - Entry: block with an edge from outside of L
  - Header h: first entry found (might be ambiguous)
- Loop nested in L: loop in subgraph $L \setminus \{h\}$

[10] 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`.



[Slide 168] Natural Loops

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



Formal Definition

Loop L is reducible iff $\exists h \in L \,.\, \forall n \in L \,.\, h \text{ dom } n$

CFG is reducible iff all loops are reducible

[Slide 169] Finding Natural Loops

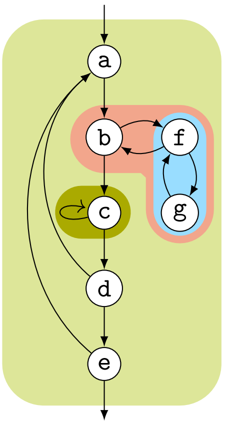
- Modified version[11] of Tarjan's algorithm[12]
- Iterate over dominator tree in post order
- Each block: find predecessors dominated by the block
  - None ⇝ no loop header, continue
  - Any ⇝ loop header, these edges *must* be backedges
- Walk through predecessors until reaching header again
  - All blocks on the way must be part of the loop body
  - Might encounter nested loops, update loop parent

[11] 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`.

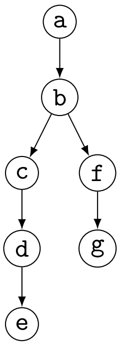
[12] R Tarjan. "Testing flow graph reducibility". In: *STOC*. 1973, pp. 96–107. URL: `https://dl.acm.org/doi/pdf/10.1145/800125.804040`.

[Slide 170] 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

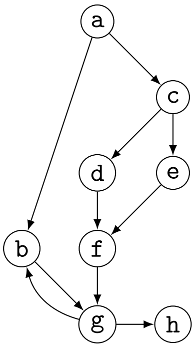
Loop C: {b,f,g}
header: b; parent: D

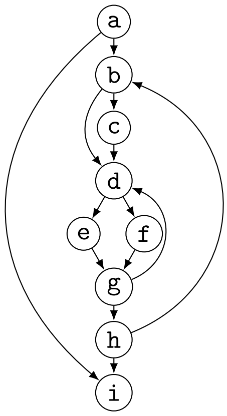
Loop D: {a,b,c,d,e,f,g}
header: a; parent: NULL

[Slide 171] Loop Analysis – Example

In-Class Exercise:

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





Solution on page 195.

5.9. Loop Optimizations

[Slide 172] Loop Invariant Code Motion (LICM)

- Analyze loops, iterate over loop tree in post-order
  - I.e., visit inner loops first

↑ Hoist:[13] iterate over blocks of loop in reverse post-order

[13] https://github.com/bytecodealliance/wasmtime/blob/bd6fe11/cranelift/codegen/src/licm.rs

  - 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)

[Slide 173] Loop Unswitch

- Hoist loop-invariant conditionals outside loop

> After loop-invariant code is moved outside of the loop, it is trivial to determine whether a condition is loop-invariant.

```
while (...)
  if (loopInvariantCond)
    A
  else
    B
  C
```

→

```
if (loopInvariantCond)
  while (...)
    A
    C
else
  while (...)
    B
    C
```

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

> For n independently unswitchable conditions, fully unswitching the loop leads to 2^n loops.

[Slide 174] Loop Rotation

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

> This reduces the number of branches per loop trip from 2 to 1. Additionally, if the loop is known to have a trip count of at least 1, the initial condition can be eliminated.

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

→

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

[Slide 175] Loop Strength Reduction

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

[Slide 176] Analyses and Transformations – Summary

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

[Slide 177] Analyses and Transformations – Questions

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

6. Optimizations in LLVM

6.1. Overview

[Slide 179] Optimizations in LLVM

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

[Slide 180] LLVM Optimization Pipeline (Simplified)[1]

Module Pipeline (`buildPerModuleDefaultPipeline`): two parts
Module Simplification Pipeline (`buildModuleSimplificationPipeline`)

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

> Inside a call graph strongly-connected component (CGSCC), functions are simplified in post-order (i.e., leaf functions first). The inliner therefore makes decisions based on the simplified callee.
> Other acronyms: SROA is Scalable Replacement Of Aggregates; GVN is Global Value Numbering (elimination of common expressions); DCE is Dead Code Elimination; LICM is Loop Invariant Code Motion (hoisting instructions that compute the same value inside the loop out of the loop).

Module Optimization Pipeline (`buildModuleOptimizationPipeline`)

[1] See `llvm/lib/Passes/PassBuilderPipelines.cpp` for full details.

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

6.2. Canonicalization

[Slide 181] SSA Construction: Mem2Reg and SROA

- Front-ends typically emit `alloca`s for variables
  - Easier; also simplifies debugging and debug information

    > Variables stored in `alloca`s have a single location, which greatly simplifies debug information, as the variable does not need to be tracked through registers. Additionally, as the variable is loaded on every use, modifications of the variable in the debugger are possible and in most cases immediately propagated.

- Mem2reg: promote `alloca` to SSA values/phis
  - Condition: only `load`/`store`, no address taken
  - Essentially just SSA construction

    > In code, this pass is called `PromotePass`.

- SROA: scalar replacement of aggregate
  - Separate structure fields into separate variables
  - Also promote them to SSA, subsumes mem2reg

> SROA is run multiple times in the pipeline, as further simplifications might enable the promotion of more variables (e.g., when escapes of the variable are eliminated).

[Slide 182] Simplification

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

⇒ Canonicalize IR: one single preferred form

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

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

> Canonicalization is one of LLVM's core approaches to program optimizations. In addition to simplifying the implementation of pattern matches (passes can assume the IR to be canonical form, although they must work correctly even on non-canonical IR), the canonicalizations also simplify and thereby optimize the program.

> In contrast to other optimizations, canonicalizations are not based on heuristics and should always work in the same direction (i.e., one pass should not undo the transformation of a previous pass). Inlining is a notable exception of a heuristics-based pass run during the simplification pipeline.
>
> Note that some canonicalization done on the IR are undone in the back-end. Examples include tail duplication (duplicating the function returns) and moving certain instructions back into loops (e.g., to reduce register pressure).
>
> Unfortunately, there is no documentation on the canonical form of IR; it is defined implicitly by what the passes (currently) produce.

[Slide 183] InstCombine

- Main instruction canonicalization and simplification pass
- Includes many arithmetic-logical patterns, folding of PHIs, ...
- Implementation:
  - Add instructions to work list
    - Trivially dead/constant instructions eliminated immediately
    - Blocks traversed in reverse post-order
  - Patterns applied to instructions; on match, replace all uses
  - Changed instruction re-enqueues all users to work list

  > The block ordering in and adding instructions in the order as they appear inside the basic block causes definitions to be visited before their uses (apart from ϕ-nodes). Therefore, adding users to the work list frequently has no effect, as it already contains all users.

- Historically: fixed-point iteration, now just one iter. per run

  > It turned out that most combines are performed in the first run. Very often, the second iteration would not change the IR but would still need to inspect it, causing easily avoidable compile time costs.
  > As the pass is executed multiple times throughout the default pass pipeline, combines missed in the first iteration are then simply performed in a later run of the pass.

- Covers many patterns, many patterns missing, >50kLOC C++

[Slide 184] IR Pattern Matching

- IR patterns are typically DAG matches on def–use graph
- Typically implemented via helper functions[2]

```
// Excerpt from InstCombinerImpl::visitAdd

// (add (xor A, B) (and A, B)) --> (or A, B)
```

[2] llvm/IR/PatternMatch.h

```
// (add (and A, B) (xor A, B)) --> (or A, B)
if (match(&I, m_c_BinOp(m_Xor(m_Value(A), m_Value(B)),
                        m_c_And(m_Deferred(A), m_Deferred(B)))))
  return BinaryOperator::CreateOr(A, B);
```

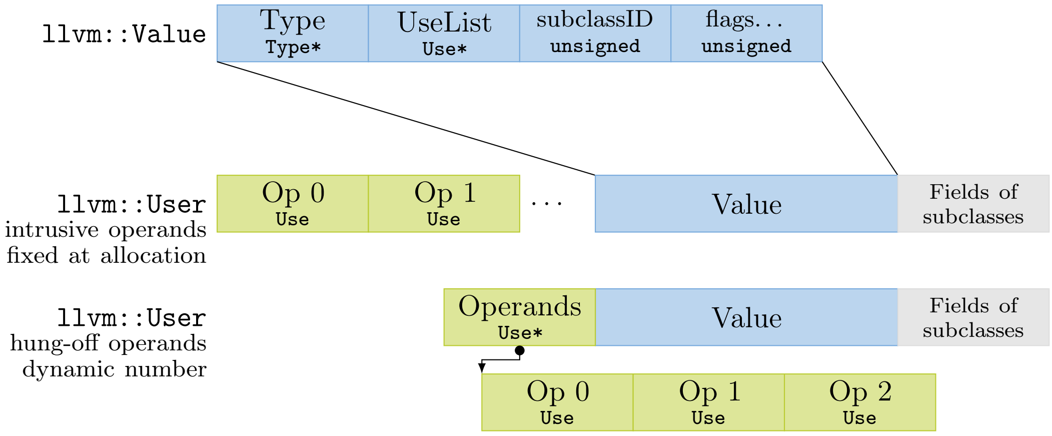
[Slide 185] LLVM: Use Tracking

- Values track their users

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

- Allows for easy replacement:
  - `inst->replaceAllUsesWith(replVal);`

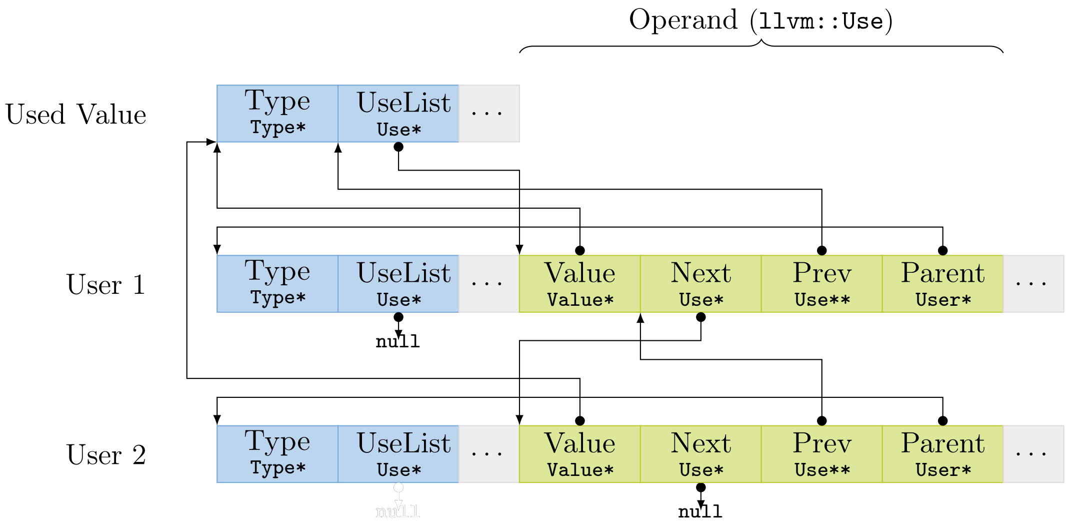
[Slide 186] LLVM IR Implementation: Value/User



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 *before* 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. Therefore, PHINode operands are always hung off.

[Slide 187] 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 188] 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 (⇝ getUniquePredecessor())
- Finding first non-ϕ requires iterating over ϕ-nodes

[Slide 189] Valid Transforms?

In-Class Exercise:

Which of these transformations `src`→`tgt` are valid?

```
define float @src1(float %x) {           define float @tgt1(float %x) {
  %a = fadd float %x, 0.0                  ret float %x
  ret float %a                           }
}
define i1 @src2(i8 %x) {                 define i1 @tgt2(i8 %x) {
  %xm1 = add i8 %x, -1                     %ctpop = call i8 @llvm.ctpop(i8 %x)
  %y = xor i8 %x, %xm1                     %r = icmp ult i8 %ctpop, 2
  %r = icmp ule i8 %x, %y                  ret i1 %r
  ret i1 %r                              }
}
define i8 @src3(i8 %x) {                 define i8 @tgt3(i8 %x) {
  %div = sdiv i8 %x, 4                     %div = ashr i8 %x, 2
  ret i8 %div                              ret i8 %div
}                                        }
```

Solution on page 195.

[Slide 190] Alive2

- Seemingly correct transformations might be wrong in edge cases
- Unexpected transformations can be correct

⇒ Prove correctness of transformations

- Alive2[3]: translation validation tool for LLVM-IR
- Convert LLVM-IR into SMT formulas
- Proves that target is a refinement of source function

> Alive2 transforms LLVM-IR functions, including memory accesses and control flow, into SMT formulas and uses the Z3 SMT solver to prove that the target function is indeed a refinement of the source function, i.e., that the target is more defined than the source. There is limited support for `undef` values (which are not tracked per-bit but just per-value) and loops (which are completely unrolled for a fixed number of iterations).

- InstCombine transformations require Alive2 proofs nowadays[4]

[Slide 191] Value Tracking

- Many transformations benefit from value ranges and known bits
  - E.g., fold `trunc`+shift into smaller shift if shift amount is known small
- `computeKnownBits`: recursively collect information from instrs
  - Arithmetic-logical instrs, casts, several intrinsics, etc.

[3] NP Lopes et al. "Alive2: bounded translation validation for LLVM". in: *PLDI*. 2021, pp. 65–79. URL: `https://users.cs.utah.edu/~regehr/alive2-pldi21.pdf`.

[4] Random Example: `https://alive2.llvm.org/ce/z/xe_vb2`

- Bitwise instructions easier to reason about ⇝ preferred canonicalization
  - E.g. `(1 << X) - 1` → `(-1 << X) ^ -1`
  - E.g. `sext(X)` → `zext(X)` if `X` is known non-negative
- Also use knowledge from front-end (but how encoded?)

6.3. Flags

[Slide 192] Attribute, Flags, Assume

- Paramter/return attributes: constrain value ranges
  - `range(from, to)`, `nonnull`, `noundef`, ...
  - Values outside of the range are `poison`
- Instruction flags: encode additional knowledge about operation
  - `add`/`sub`/`mul`/`trunc` `nsw`/`nuw`: no signed/unsigned wrap

    > This can be used, e.g., to restrict the value range or get more precise information about loop trip counts, memory access offsets.

  - `or disjoint`: combination of disjoint bits

    > This permits back-ends to replace the `or` with an `add`, which can be folded into other operations (e.g., `lea` on x86).

  - `udiv`/`sdiv`/`lshr`/`ashr` `exact`: divisor multiple of dividend

    > Permit replacing division with shift and implies that the bits that are shifted out are zero.

  - `getelementptr` `nuw`/`nusw`/`inbounds`: no wrap/stays inside allocation

    > Bounds information primarily improves alias analysis: if two base pointers are known to refer to different allocations, they will never be the same after two `inbounds` computations.

  - `zext`/`uitofp` `nneg`: operand is non-negative

    > When a value is known to be non-negative, the instruction is converted into the canonical unsigned form. However, some targets might want to undo this transformation in the back-end, e.g., because sign-extension is cheaper (RISC-V) or the floating-point conversion is cheaper for signed integers (x86), but it might be difficult to re-prove the non-negativeness after other transformations. This flag retains this information and permits relaxing the operations later on.

  - `icmp samesign`: operands have the same sign

> When both operands are known to have the same sign, `icmp` is canonicalized into the unsigned comparison predicate. However, when there are multiple comparisons of the same value with some being signed while others are unsigned, it is more difficult to the aggregate information. This flag retains this information and indicates that the signed and unsigned variants of the comparison predicate can be used interchangeably.

  - Violation is `poison`

[Slide 193] Assume

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

> Here's an example which demonstrate the use of `separate_storage` assumptions:
>
> ```
> struct Buffer {
>   char *data;
>   void write(char c) {
>     *data++ = c;
>   }
> };
> void writeLE(unsigned x, Buffer &buf) {
>   buf.write(x);
>   buf.write(x >> 8);
>   buf.write(x >> 16);
>   buf.write(x >> 24);
> }
> ```
>
> The resulting machine code will repeatedly load the `data` pointer from `buf`, because `data` might alias with `buf`, because in C/C++ `char*` can alias with everything. When exposing the information that inside `write()` the pointers `this` and `this->data` do not alias, the four stores can get combined.
>
> ```
> // Inside write:
> __builtin_assume_separate_storage(this, data);
> // LLVM IR: call void @llvm.assume(i1 true) [ "separate_storage"(ptr %this, ptr %data) ]
> ```
>
> Full example: `https://godbolt.org/z/TMGrTco9T`

[Slide 194] Valid Transforms?

In-Class Exercise:

Which of these transformations `src`→`tgt` are valid?

```
define i8 @src1(i8 %x) {                          define i8 @tgt1(i8 %x) {
  %div = udiv exact i8 1, %x                        ret i8 1
  ret i8 %div                                     }
}
define i1 @src2(i8 %x, i8 %y) {                   define i1 @tgt2(i8 %x, i8 %y) {
  %cmp = icmp sge i8 %x, %y                         %cmpeq = icmp samesign eq i8 %x, 127
  %cmpeq = icmp samesign eq i8 %x, 127              ret i1 %cmpeq
  %r = select i1 %cmp, i1 %cmpeq, i1 false        }
  ret i1 %r
}
define i8 @src3(i8 %x, i8 %y, i8 %z) {            define i8 @tgt3(i8 %x, i8 %y, i8 %z) {
  %xy = add nsw i8 %x, %y                           %xz = add nsw i8 %x, %z
  %xyz = add nsw i8 %xy, %z                         %xyz = add nsw i8 %xz, %y
  ret i8 %xyz                                       ret i8 %xyz
}                                                 }
```

Solution on page 195.

[Slide 195] Fast-Math Flags

- Floating-point arithmetic typically follows IEEE 754
  - Except for NaN, which is more relaxed

    > The behavior of floating-point instructions w.r.t. NaN varies strongly between different targets (e.g., taking one of the operands, returning a canonical NaN, converting signalling NaN to quiet NaN). Therefore, the semantics are specified rather generic for LLVM-IR.

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

[Slide 196] Branch Weights

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

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

!5 = !{!"branch_weights", !"expected", i32 1, i32 2000}
```

6.4. Inter-Procedural Optimizations

[Slide 197] Function Argument Optimizations

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

[Slide 198] Other Inter-Procedural Optimizations

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

[Slide 199] Optimization Remarks

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

[Slide 200] Other Aspects not Covered Here

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

6.5. Running and Writing LLVM Passes

[Slide 201] 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 202] 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 203] 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 204] Optimizations in LLVM – Summary

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

[Slide 205] Optimizations in LLVM – Questions

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

7. Vectorization

7.1. SIMD Overview

[Slide 207] 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 208] Single Instruction, Multiple Data (SIMD)

- Idea: perform same operations on multiple data in parallel
- First computer with SIMD operations: MIT Lincoln Labs TX-2, 1957[1]

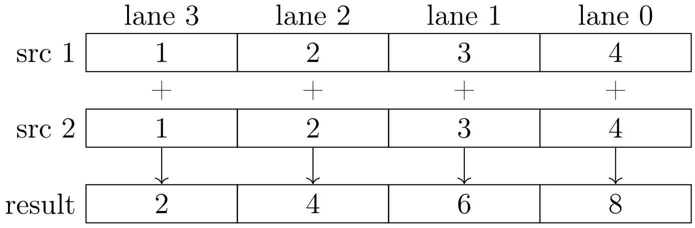
> The MIT TX-2 supported computations on one 36-bit integer, two 17-bit integers, four 9-bit integers, and the pair of one 27-bit and one 9-bit integer.

- 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

[1] 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`.

[Slide 209] 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 210] 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 211] Historical Development of SIMD Extensions

See Figure 7.1.

[Slide 212] 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

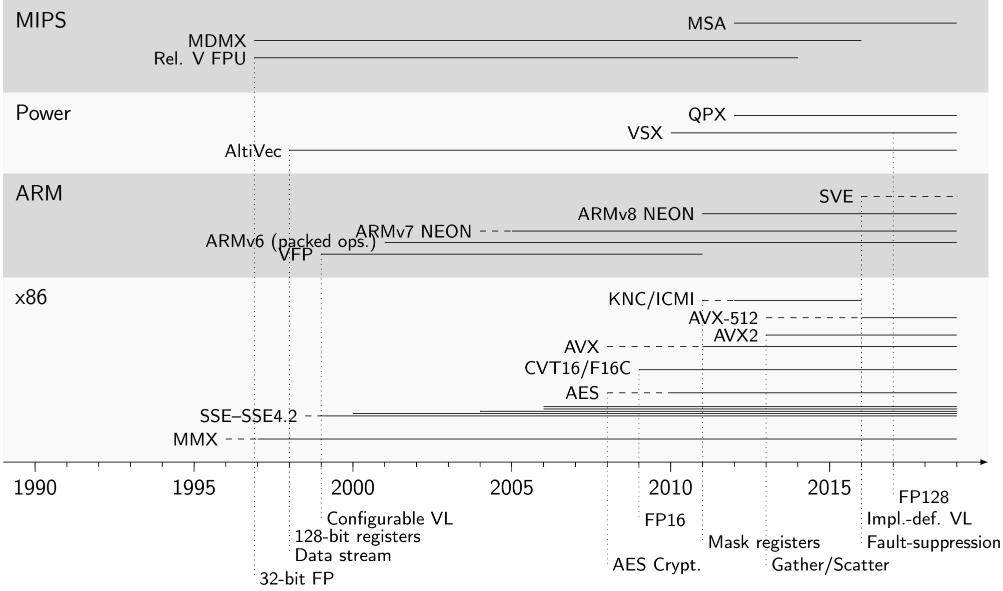


Figure 7.1.: Historical Development of SIMD extensions to contemporary ISAs.

- Cryptography

[Slide 213] 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 ~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

7.2. Low-Level SIMD Programming

[Slide 214] 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 215] 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 216] SIMD Programming: Intrinsics – Example

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

[Slide 217] 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

7.3. Target-Independent Vector Extensions

[Slide 218] 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 219] 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 195.

[a] https://godbolt.org/z/43v98rdno

7.4. Vectors in LLVM-IR

[Slide 220] 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

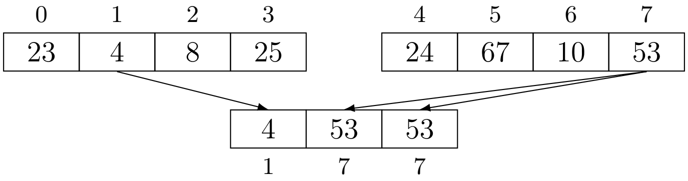
  - Intrinsic `@llvm.vscale.i32()` – get runtime value of `vscale`
- Most arithmetic operations can also operate on vectors
- `insertelement`/`extractelement`: modify single element
  - Example: `%4 = insertelement <4 x float> %0, float %1, i32 %2`
  - Index can be non-constant value

> 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 221] 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>`

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



[Slide 222] `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 196.

[Slide 223] 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 196.

[Slide 224] 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)
```

7.5. Single Program, Multiple Data

[Slide 225] 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 226] SPMD: Example (OpenMP)

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

- Compiler generates version that operates on vector

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

_ZGVxN4vv_foo:
  paddd xmm0, xmm1
  ret
```

[Slide 227] SPMD: Example (OpenMP)

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

- Uniform: always same value

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

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

[Slide 228] 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 229] 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 230] SPMD to SIMD: Handling Loops

- Uniform loops: nothing different
- Otherwise: need to retain loop structure
  - "active" mask added to all loop iterations
  - Loop only terminates once all lanes terminate (active is zero)
  - Lanes that terminated early need their values retained
- Approach also works for nested loops/conditions
- Irreducible loops need special handling[2]

[Slide 231] SPMD Implementations on CPUs

- OpenMP SIMD functions
  - Need to be combined with `#pragma omp simd` loops
- Intel ispc[3] (Implicit SPMD Program Compiler)
  - Extension of C with keywords `uniform`, `varying`
  - Still active and interesting history[4]
- OpenCL on CPU
  - Very similar programming model
  - But: higher complexity for communicating with rest of application

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

[3] M 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 232] 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

7.6. Auto-Vectorization

[Slide 233] 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 234] 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 235] 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 benefitial?
   - Consider: runtime checks, gather/scatter, masked operations, etc.

- Needs information about target architecture

3. Perform transformation

[Slide 236] 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 237] 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 238] Auto-Vectorization: Examples

In-Class Exercise:

Compile[a] the functions from `ex07.txt` with vectorization remarks.

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

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

[a] https://godbolt.org/z/aqbb937cb

[Slide 239] 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 240] 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?

8. Instruction Selection

8.1. Overview

[Slide 242] Code Generation – Overview

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

[Slide 243] 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 244] 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 245] Optimal ISel

- Find *most performant* instruction sequence with same semantics (?)
  - I.e., no program with better "performance" exists
  - Performance ≈ 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
  - $\mathcal{NP}$-complete[1]

    > 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 246] 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.

8.2. Macro Expansion

[Slide 247] Macro Expansion

- Expand each IR operation with corresponding machine instrs

[1] 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`.

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

[Slide 248] 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 249] Peephole Optimization

- Plain macro expansion leads to suboptimal results
- Idea: replace inefficient instruction sequences[2]
- Originally: physical window over assembly code
  - Replace with more efficient instructions having same effects
  - Possibly with allocated registers
- Extension: do expansion before register allocation[3]
  - 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 250] 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

[2] WM McKeeman. "Peephole optimization". In: *CACM* 8.7 (1965), pp. 443–444. URL: `https://dl.acm.org/doi/pdf/10.1145/364995.365000`.

[3] 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`.

8.3. Tree Covering

[Slide 251] 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

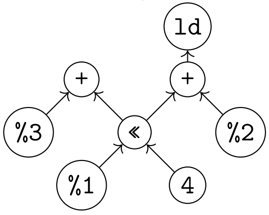
[Slide 252] Tree Covering: Converting SSA into Trees

- SSA form:

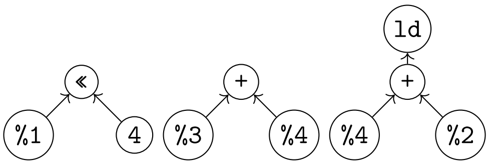
```
%4 = shl %1, 4
%5 = add %2, %4
%6 = add %3, %4
%7 = load %5
```

live-out: `%6`, `%7`

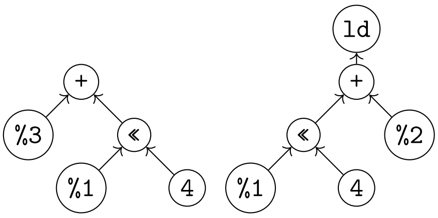
- Data flow graph:



- Method 1: Edge Splitting



- Method 2: Node Duplication



[Slide 253] Tree Covering: Patterns

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

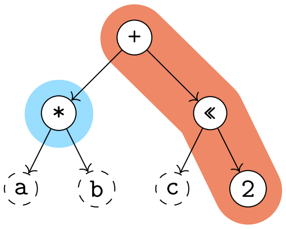
[Slide 254] Tree Covering: Greedy/Maximal Munch

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

+ Easy to implement, fast

– Result might be non-optimum

[Slide 255] Tree Covering: Greedy/Maximal Munch – Example



Matching Patterns:

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

Total cost: 5

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

[Slide 256] 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

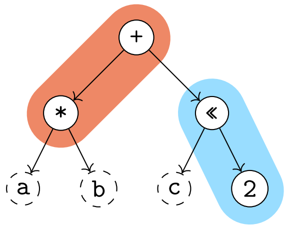
Disadvantages

- 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 257] Tree Covering: Dynamic Programming[5]

- Step 1: compute cost matrix, bottom-up for all nodes
  - Matrix: tree node × 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 258] Tree Covering: Dynamic Programming – Example



[4] 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`.

[5] 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`.

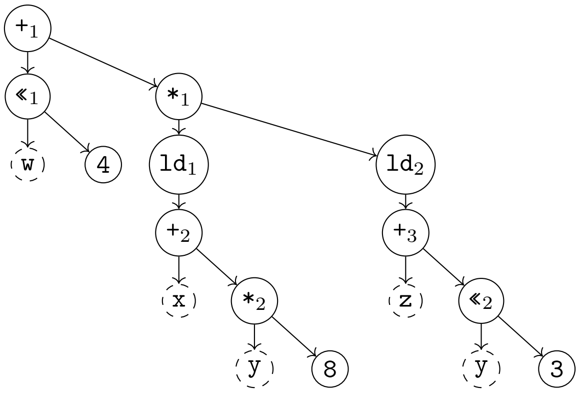
Node:	+
Pattern:	P_9: $GP \rightarrow$ +(*(GP, GP), GP)
Pat. Cost:	3
Cost Sum:	4

	Node	+	*	«	2
GP	Cost	4	3	1	1
	Pattern	P_9	P_8	P_0	P_{10}

[Slide 259] Tree Covering: Dynamic Programming – Example

In-Class Exercise:

Find a tree covering using the dynamic programming algorithm.



Solution on page 196.

[Slide 260] 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 tables[7]

[6] 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.

[7] 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.

[Slide 261] 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 ⇒ inefficient code)

– Cannot make use of multi-output instructions (e.g., `divmod`)

8.4. DAG Covering

[Slide 262] DAG Covering

- Idea: lift restriction of trees, operate on data flow DAG
  - Reminder: an SSA basic block already forms a DAG
- Trivial approach: split into trees ☹
- Least-cost covering is $\mathcal{NP}$-complete[8]

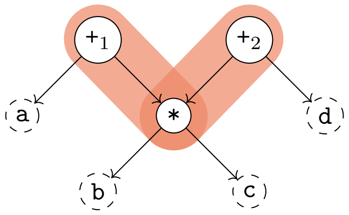
[Slide 263] DAG Covering: Adapting Dynamic Programming I[9]

- 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 264] DAG Covering: Adapting Dynamic Programming I – Example



[8] 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`.

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

Total cost: 6

```
madd %1, %b, %c, %a
madd %2, %b, %c, %d
```

Optimal cost: 5 ⇝ non-optimal result

	Node	$+_2$	$+_1$	$*$
GP	Cost	3	3	3
	Pattern	P_9	P_9	P_8

[Slide 265] DAG Covering: Adapting Dynamic Programming II[10]

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

+ Probably fast? "Near-optimal"?

− Generally not optimal, superlinear time

[Slide 266] DAG Covering: ILP[11]

- 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

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

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 267] DAG Covering: Greedy/Maximal Munch

- Top-down, start at roots, always take largest pattern
- Repeat for remaining roots until whole graph is covered

[10] 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.

[11] 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.

+ Easy to implement, reasonably fast
− Result often non-optimal

- Used by: LLVM SelectionDAG

8.5. Graph Covering

[Slide 268] 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)

8.6. ISel in Practice

[Slide 269] 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 270] Instruction Selection in Practice

- Most compilers use some form of greedy tree/DAG pattern matching
- Later stages use peephole optimizations
  - Basically also tree/DAG matching on machine operations
- Distinction between tree/DAG/graph matching somewhat artificial[12]

Problem in practice: implementing the huge amount of required patterns

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

[12] My personal opinion.

> To illustrate the last point: x86 has no instruction to reverse bits of an integer. However, with the GFNI extension, the instruction `vgf2p8affineqb` was introduced, which can be configured in a way that it performs a bit-reverse operation[a].[b] (Would you come up with this pattern?)
>
> [a] `https://github.com/llvm/llvm-project/pull/81764`
>
> [b] Example: `https://godbolt.org/z/34fW6hc64`

8.7. LLVM Instruction Selection

[Slide 271] 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

[Slide 272] 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 273] LLVM MIR Example

In-Class Exercise:

Analyze the Machine IR of the following code[a]. (Also consult the reference[b].)

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

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

[a] https://godbolt.org/z/zj9WM84Wh

[b] https://llvm.org/docs/MIRLangRef.html

[Slide 274] 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 275] LLVM SelectionDAG: IR to ISelDAG

- Construct DAG for basic block
  - EntryToken as ordering chain
- Legalize data types
  - Integers: promote or expand into multiple

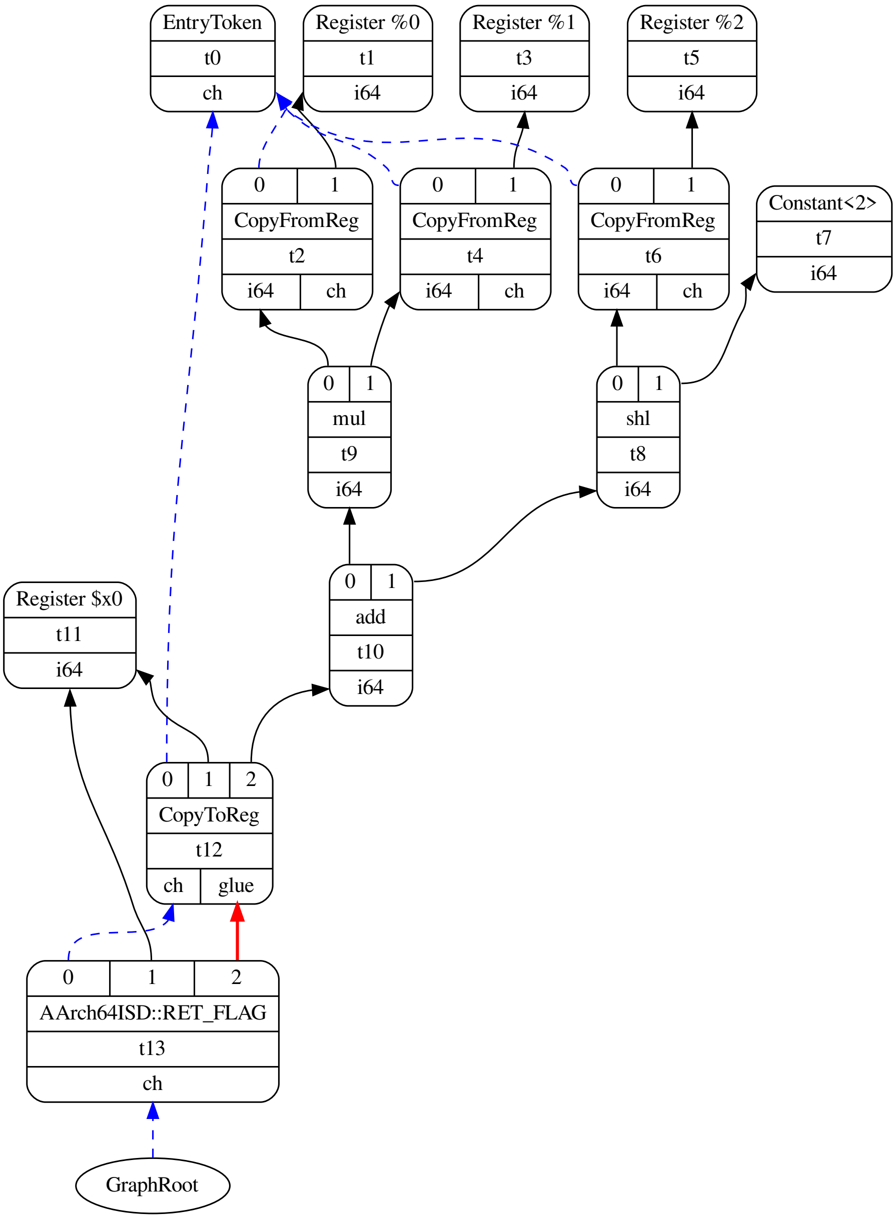


Figure 8.1.: ISel DAG right before instruction selection.

  - 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 8.1.

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

Note: needs LLVM debug build

[Slide 276] 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 8.2.

`llc -march=aarch64 -view-sched-dags`Note: needs LLVM debug build

[Slide 277] LLVM GlobalISel

- DAGs do have limitations w.r.t. instruction selection
- Graph loses original scheduling information (why problematic?)

> The original instruction ordering, e.g., as occurring in the input source, is lost during instruction selection. This gives programmers less opportunities to influence the order of operations in the resulting machine code and also can contributes to frequent jumps between lines when stepping through the code in a debugger.

- Building a separate IR is expensive
- Core idea of GlobalISel: do legalization and ISel in MachineIR
- MIR is target-specific ⇝ introduce generic operations
- Permit matching/combining instructions from different blocks

[Slide 278] LLVM GlobalISel: Passes

- IRTranslator: translate LLVM-IR into gMIR
  - One-to-one mapping, except call and switch
- PreLegalizerCombiner
- Localizer: try to improve placement of constant materialization
- Legalizer: replace illegal instructions with legal gMIR instrs

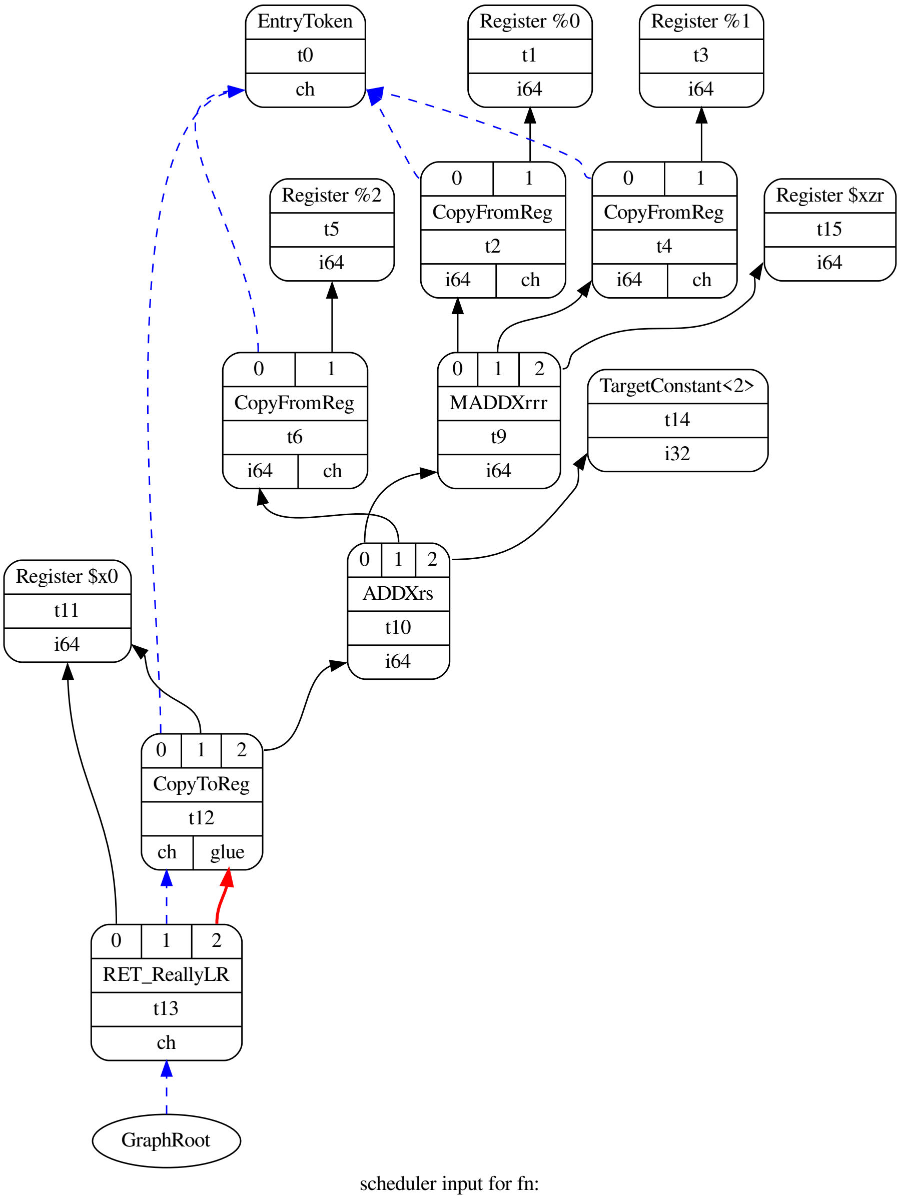


Figure 8.2.: X86 DAG after Instruction Selection

> Illegal instructions are instruction that the target does not support, e.g. due to unsupported types. For example, AArch64 only supports 32/64-bit integer arithmetic, so operations on smaller integers (e.g., `i1`) need to be promoted to a legal type; likewise for larger integers, which need to be split up.

- PostLegalizerCombiner
- RegBankSelect: assign register bank (reduce copies between banks)

> While moves between registers of the same register file are very cheap on modern CPUs, moves between different register files (e.g., general-purpose and floating-point registers) remain somewhat expensive.

- InstructionSelect

[Slide 279] Instruction Selection – Summary

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

[Slide 280] Instruction Selection – Questions

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

9. SSA Destruction and Liveness Analysis

[Slide 282] Register Allocation: Overview

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

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

[Slide 283] Register Allocation: Strategy Overview

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

9.1. SSA Destruction

[Slide 284] SSA Destruction

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

[Slide 285] SSA Destruction: Example 1

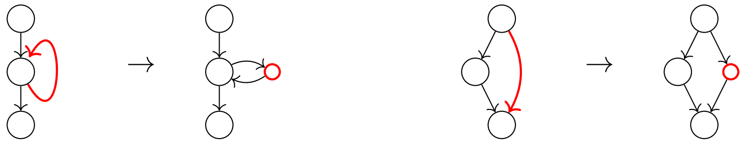
```
identity(%0)
e:
  br %loop
loop:
  %3 = phi [ 0, %e ], [ %4, %loop ]
  %4 = add %3, 1
  %5 = cmp ult %4, %0
  br %5, %loop, %ret
ret:
  ret %3
```

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

- Original value lost in return block!

[Slide 286] Lost Copy Problem

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



- *Break* critical edges: insert an empty block

[Slide 287] SSA Destruction: Example 1 – Critical Edge Split

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

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

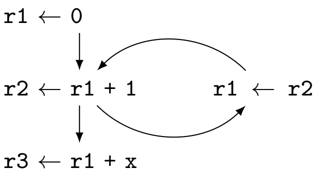
- Problem fixed, but one extra branch per loop iteration

> However, a good block layout algorithm will address this problem by rotating the block `critedge` to the top of the loop so that branch to `loop` is a fall-through.

[Slide 288] Handling Critical Edges

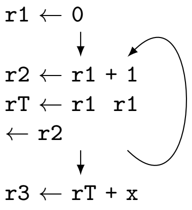
Breaking Edges

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



Copy Used Values

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



[Slide 289] SSA Destruction: Example 1 – Value Copy

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

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

- Problem fixed; register allocator can hopefully omit a copy

[Slide 290] SSA Destruction: Example 2

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

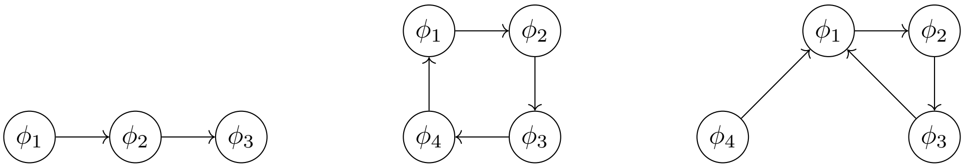
```
odd(%0)
e:
  %3 = copy %0
  %4 = copy 1
  %5 = copy 0
  br %loop
loop:
  %6 = cmp ne %3, 0
  br %6, %body, %ret
body:
  %8 = sub %3, 1
  %3 = copy %8
  %4 = copy %5
  %5 = copy %4
  br %loop
ret:
  ret %4
```

- Value of ϕ (%5) is wrong!

[Slide 291] Swap Problem / ϕ Dependencies

> ϕ-nodes execute concurrently, so a sequence of ϕ-nodes is a *parallel copy* when they refer to each other. These copies need to be serialized into multiple, sequentially executed copy operations.

- 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 292] 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 ⇝ start of chain
   - Handling node may unblock next element in chain
3. For all remaining ϕ-nodes: must be cycles, reader count always 1
   - For trivial cycles (single node), do nothing
   - Choose arbitrary node, load to temporary register, unblock value
   - Handle just-created chain
   - Write temporary register to target

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

[Slide 293] SSA Destruction: Example 2 – Fixed

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


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

- ϕ-cycle on edge `body`→`loop` broken with temporary register

[Slide 294] SSA Destruction: Exercise

In-Class Exercise:

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

1. Dependencies between 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

Solution on page 197.

[Slide 295] SSA Destruction: Alternative Approach

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

> This is what LLVM does.

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

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

9.2. Simple Register Allocation

> There are, of course, extremely simple ways to deal with the complex problem of register allocation. Unfortunately, they are inherently unsuitable for any kind of practical programs, as they ignore fundamental information about the program. Do not implement the “ideas” outlined in this section, they are merely for illustration.

[Slide 296] Register Allocation: Simple Approaches

- Simplest thing that could possibly work: allocate a one stack slot for every (SSA) variable/argument

  - Reload all operands immediately before the instruction
  - Store to stack slot after computation

+ Simple, always works, debugging easy

– Extremely slow: values are stored and immediately reloaded

– Extremely inefficient memory usage: huge stack frames

[Slide 297] Register Allocation: Simple Approaches

- Next simplest thing that could possibly work: avoid reload if value is still in a register
  - When assigning target register: choose any register
  - Across basic blocks, spill everything

+ Still simple, always works, debugging easy

– Very slow: unnecessary stores, loop variants in memory

– Extremely inefficient memory usage: huge stack frames

[Slide 298] Register Allocation: Problems of Simple Approaches

- Many avoidable spills
  - Spill of last use of a value can/should be omitted
  - Stack slots for unused variables should be reused (why?)

  > Large stack frames result in a higher memory usage and worse cache utilization. Additionally, especially on ISAs with fixed-length encoding, accessing stack slots beyond a certain distance from the stack/frame/base pointer requires multiple instructions (e.g., to materialize the offset into a temporary register).

- 2-address instructions (destructive source) require value copy
- Bad eviction decisions: value might be used immediately afterwards
- Loop-variants should kept be in registers if possible

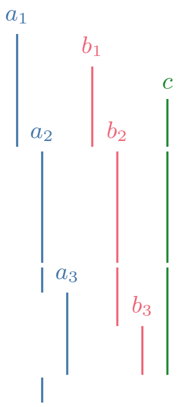
$\rightsquigarrow$ Need information about what variables need to be preserved

9.3. Liveness Analysis

[Slide 299] Liveness: Definitions

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

[Slide 300] Liveness: Example



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

> Some observations:
>
> - Although c only has a single user, the conditional branch in `loop`, it must also be live throughout the entire `body` block, as later iterations of the loop might (will) need the value.
> - The computation of a_3 kills a_2. The live ranges of a_2 and a_3 don't overlap, which implies that a_2 and a_3 will be able to share the same location.
> - The live range of a_2 is not contiguous, it has a *hole*.
>
> If the basic block order is not fixed, live ranges have no real meaning across basic block boundaries.

[Slide 301] Liveness: Definitions[1]

- $Defs(B)$: values defined in B (no defs from ϕ)
- $Uses(B)$: values used in B (no uses in ϕ)
- $UpwardExposed(B)$: values used in B before a definition in B
  - SSA: this is $Uses(B) \setminus (Defs(B) \cup PhiDefs(B))$

> In non-SSA form, this includes values in sequences like $\dots \leftarrow v; v \leftarrow \dots$. In SSA form, these are strictly that are used in B but not defined in B.

- $PhiDefs(B)$: values defined by ϕ at entry of B
- $PhiUses(B)$: values used in ϕs in successors of B

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

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

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

[Slide 302] Liveness: Iterative Data Flow Algorithm[2]

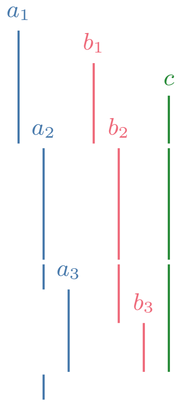
- Precompute per-basic block sets
  - Requires backwards pass over instructions of basic block
- Initialize *LiveIn*/*LiveOut* to empty sets
- Iteratively recompute *LiveIn*/*LiveOut* until convergence
- Block order has strong impact on runtime; post-order preferable

> Actually, the best results can be obtained using reverse post order of reversed CFG (i.e., the CFG where all edges reversed), in which case as many successors as possible are visited before a block itself, with the only exception of backedges.

- Exact live ranges can be tracked
  - LiveOut ⇒ live range extends to end of block; otherwise ends at last use inside block

[Slide 303] Liveness: Iterative Algorithm Example

> Animated version on slides only. The algorithm takes two iterations to converge; in the second iteration, c will be marked as live throughout the entire loop.



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

[Slide 304] Liveness: Exercise

In-Class Exercise:

[2] GA Kildall. "A unified approach to global program optimization". In: *POPL*. 1973, pp. 194–206. URL: `https://dl.acm.org/doi/pdf/10.1145/512927.512945`.

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

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

Solution on page 198.

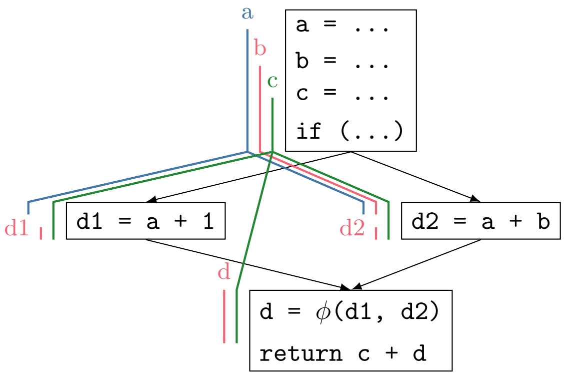
[Slide 305] Liveness: Iterative Algorithm Analysis

- Convergence: must converge, liveness sets grow monotonically
- Max. iterations: $d(G) + 3$[3]
  - $d(G)$ is depth = max. number of backedges on cycle-free path in G
- Variation: worklist instead of round-robin[4]

[Slide 306] Liveness on SSA

- SSA values have one definition; uses must be dominated by that definition

⇒ Live ranges are sub-trees of dominator tree



[3] JB Kam and JD Ullman. "Global data flow analysis and iterative algorithms". In: *JACM* 23.1 (1976), pp. 158–171. URL: https://dl.acm.org/doi/pdf/10.1145/321921.321938

[4] KD Cooper, TJ Harvey, and K Kennedy. "An empirical study of iterative data-flow analysis". In: *CIC*. 2006, pp. 266–276

> This implies:
> - A live range begins either at the definition in the block or at the start of the block (*live in*).
> - A live range ends either at the last use inside a basic block (the *killing* use) or at the end of the block (*live out*).

[Slide 307] Liveness: Two-Pass Algorithm on SSA[5]

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

[Slide 308] Liveness: Two-Pass Algorithm on SSA for Irreducible CFG[6]

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

[Slide 309] Liveness: Over-Approximation

- Precise liveness analysis is somewhat expensive for large functions
- Also requires super-linear amount of memory to store
- For JIT compilation, not always feasible
- ⇝ Over-approximation of liveness information
- Typical approach: single live interval per value[7]
  - Define fixed block order (typically RPO or similar)

    > As an implication, the block order has a substantial impact on the length of the live intervals.

  - Store begin and end location of live interval

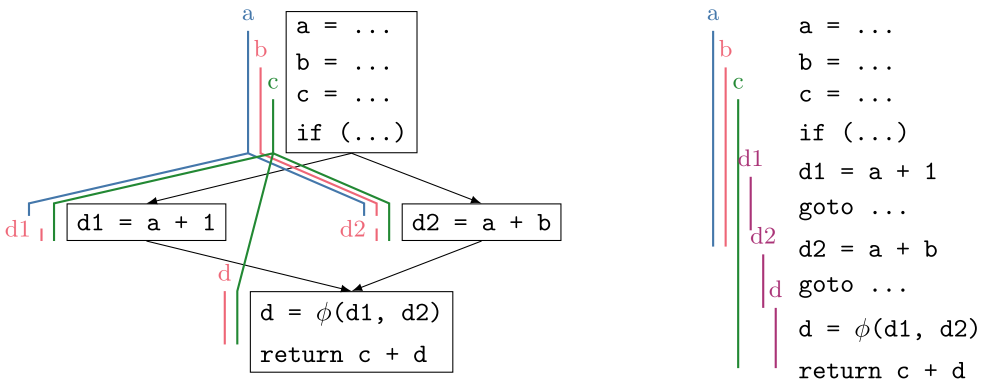
[5] B Boissinot et al. "A non-iterative data-flow algorithm for computing liveness sets in strict SSA programs". In: *ASPLAS*. 2011, pp. 137–154.

[6] B Boissinot and F Rastello. "Liveness". In: *SSA-based Compiler Design*. Ed. by F Rastello and F Bouchez Tichadou. 2022, pp. 107–122. DOI: `10.1007/978-3-030-80515-9_9`.

[7] M Poletto and V Sarkar. "Linear scan register allocation". In: *TOPLAS* 21.5 (1999), pp. 895–913. URL: `https://dl.acm.org/doi/pdf/10.1145/330249.330250`

  - Value might be marked as live in between even if it actually isn't

[Slide 310] Liveness: Live Ranges vs. Single Live Interval



```
a = ...
b = ...
c = ...
if (...)
```

```
d1 = a + 1
```

```
d2 = a + b
```

```
d = ϕ(d1, d2)
return c + d
```

```
a = ...
b = ...
c = ...
if (...)
d1 = a + 1
goto ...
d2 = a + b
goto ...
d = ϕ(d1, d2)
return c + d
```

- Single live interval can be substantially worse

[Slide 311] Live Interval Computation

- Can be done in single pass over program[8]
- Also can omit computation of live-in/live-out sets[9]
- Conceptually no large difference to two-pass algorithm except: growing live ranges simply adjusts end of interval

[Slide 312] Liveness Through Path Exploration[10]

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

[8] C Wimmer and M Franz. "Linear scan register allocation on SSA form". In: *CGO*. 2010, pp. 170–179. URL: `http://www.christianwimmer.at/Publications/Wimmer10a/Wimmer10a.pdf`.

[9] A Kohn, V Leis, and T Neumann. "Adaptive execution of compiled queries". In: *ICDE*. 2018, pp. 197–208. URL: `https://db.in.tum.de/~leis/papers/adaptiveexecution.pdf`.

[10] F Rastello. "On Sparse Intermediate Representations: Some Structural Properties and Applications to Just-In-Time Compilation". Habilitation thesis. Inria Grenoble Rhône-Alpes, 2012. URL: `https://inria.hal.science/hal-00761555/file/habilitation.pdf`.

[Slide 313] SSA Destruction and Liveness Analysis – Summary

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

[Slide 314] SSA Destruction and Liveness Analysis – Questions

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

10. Register Allocation

[Slide 316] Register Allocation

- Question: are there enough registers for all values?
  - *Register pressure* = number of values live at some point
  - Register pressure > #registers ⇒ 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 317] 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 ⇒ 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

- Traditionally done on non-SSA form

10.1. Scan-Based Approaches, no SSA

[Slide 318] Linear Scan Register Allocation[1]

In-Class Exercise:

Read sections 3 and 4 of the paper.

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

Solution on page 199.

[Slide 319] Linear Scan Register Allocation

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

- What's missing?
  - Registers to load spilled values
  - Shuffling of values between blocks
  - Register constraints (e.g., for instructions or function calls)
- Other disadvantage: once a value is spilled, it is spilled everywhere
  - Some other approaches based on lifetime splitting[2]
- Function calls: clobber lots of registers

[Slide 320] Scan-based Register Allocation[3]

Iterate over basic blocks[4]

- Start with register assignment from predecessor
  - Multiple predecessors: choose assignment from any one
- Iterate over instructions top-down
  - Ensure all instruction operands are in registers
    - When out of registers: move any value to stack
  - For operands in their last use: mark register as free
  - Assign instruction result to new free register

[1] M Poletto and V Sarkar. "Linear scan register allocation". In: *TOPLAS* 21.5 (1999), pp. 895–913. URL: `https://dl.acm.org/doi/pdf/10.1145/330249.330250`.

[2] 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`.

[3] Mostly following Go: `https://github.com/golang/go/blob/5f7abe/src/cmd/compile/internal/ssa/regalloc.go`

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

- Shuffle values back into registers where successor expects them[5]

[Slide 321] Scan-based Register Allocation – Spilling

What to spill?

- Spill value with furthest use in future[6]
  - Frees register for longest time
  - Requires information on next use to be stored during analysis
  - But: avoid spilling values computed inside loops (esp. loop-carried dependencies), reloads are fine[7]
  - Downside: superlinear run-time

Where to store?

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

[Slide 322] Scan-based Register Allocation – Spilling

Where to insert store?

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

[Slide 323] 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

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

[6] C Wimmer and H Mössenböck. "Optimized interval splitting in a linear scan register allocator". In: *VEE*. 2005, pp. 132–141.

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

10.2. Graph-Based Approaches, no SSA

[Slide 324] Graph Coloring: Overview

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

[Slide 325] Chaitin's Algorithm[8]

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

[Slide 326] Chaitin's Algorithm: Coalescing

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

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

[Slide 327] Chaitin's Algorithm: Register Constraints

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

[8] GJ Chaitin. "Register allocation & spilling via graph coloring". In: *SIGPLAN* 17.6 (1982), pp. 98–101. URL: `https://dl.acm.org/doi/pdf/10.1145/872726.806984`.

[Slide 328] Graph Coloring: More Approaches

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

10.3. Register Allocation on SSA

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

10.4. Generating Assembly

[Slide 330] 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 331] 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

[Slide 332] Register Allocation – Summary

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

[Slide 333] Register Allocation – Questions

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

11. Object Files, Linker, and Loader

[Slide 335] 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 336] 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)

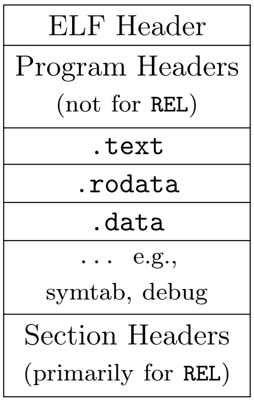
> Although required for shared libraries and increasingly used for binaries as a security measure, position-independent code (PIC) can have a significant performance impact, as an extra indirection step is needed whenever an absolute address is required.

Compiler needs to know code model

11.1. Object Files

[Slide 338] Executable and Linkable Format (ELF)[1]

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


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

[Slide 339] 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 */
```

[1] gABI specification: `https://gabi.xinuos.com/`

```
  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;
```

> - Version: There is only one version, `EV_CURRENT` (1).
> - Machine: processor architecture (e.g. x86-64, RISC-V)
> - Processor-specific flags: rarely used, RISC-V e.g. uses this to encode information about the used floating-point ABI and whether compressed instructions are used.

[Slide 340] 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 341] 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 342] 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 343] 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 344] 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 345] 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 346] 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_PLT32       foo-0x4
  10:   e8 00 00 00 00 call   15 <func+0xe>
      11: R_X86_64_PLT32      external-0x4
  15:   48 83 c4 08     add    rsp,0x8
  19:   c3              ret
```

[Slide 347] Relocations

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

Static Relocation

`ET_REL`

- 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 348] Relocation Types

- Types and meaning defined by psABI[2]

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
`R_X86_64_64`	64	$S + A$
`R_X86_64_PC32`	32	$S + A - P$
`R_X86_64_GOT32`	32	$G + A$
`R_X86_64_PLT32`	32	$L + A - P$
`R_X86_64_GLOB_DAT`	addr	S
`R_X86_64_JUMP_SLOT`	addr	S
`R_X86_64_RELATIVE`	addr	$B + A$
`R_X86_64_GOTPCREL`	32	$G + GOT + A - P$
`R_X86_64_GOTPCRELX`		
`R_X86_64_REX_GOTPCRELX`		

Name	Field	Calculation
`R_X86_64_32`	32	$S + A$ (zext)
`R_X86_64_32S`	32	$S + A$ (sext)
`R_X86_64_GOTOFF64`	64	$S + A - GOT$
`R_X86_64_GOTPC32`	32	$GOT + A - P$
`R_X86_64_GOT64`	64	$G + A$
`R_X86_64_GOTPCREL64`	64	$G + GOT + A - P$
`R_X86_64_GOTPC64`	64	$GOT + A - P$
`R_X86_64_PLTOFF64`	64	$L - GOT + A$
`R_X86_64_SIZE32`	32	$Z + A$
`R_X86_64_SIZE64`	64	$Z + A$

[2] x86-64: 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 349] 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
```

[Slide 350] Relocations on RISC Architectures

- RISC architectures typically have *more* relocation types
  - Example: AArch64[3] 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 351] 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 ⇝ 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 ⇝ needs ind. jump *often*

[Slide 352] Branch Relocations on RISC-V

[3] Arm Ltd. *ELF for the Arm 64-bit Architecture (AArch64)*. URL: https://github.com/ARM-software/abi-aa/blob/main/aaelf64/aaelf64.rst.

In-Class Exercise:

1. Compile the code[a] with:
   `clang --target=riscv64 -c -o rv.o rv.c -falign-functions=16`
   ```
   int f() { return 0; }
   int g() { return f(); }
   int h() { return g(); }
   ```
2. Look at the relocations and disassembly: `llvm-objdump -dr rv.o`
   How are the function calls lowered? What types of relocations are there?
   Look up the relocations types in the psABI[b] – what do they mean?
3. Link the file[c]: `ld.lld -shared -o rv.so rv.o` and disassemble `rv.so`.
   What is different now?

Solution on page 199.

[a] https://godbolt.org/z/7d3PWzY4f

[b] https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/b17fc7b9b/riscv-elf.adoc#relocations

[c] Compiler Explorer: Output → uncheck *Compile to object*, check *Link to binary*.

11.2. Executable Files

[Slide 354] Linker[4]

- 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 355] ELF Executable File

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

[4] Interesting blog on linkers, etc.: F Song. *Personal Blog*. URL: https://maskray.me/ (visited on 01/07/2026).

- Compiler driver (e.g. `gcc`, `clang`) can output low-level tool invocations: `clang -### ...`

[Slide 356] ELF Executable File: Program Headers

- 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&(pgsize-1)==offset&(pgsize-1)` – kernel will just `mmap` the file

    > `mmap` simply maps the parts of the file from the kernel's file cache into the address space of the executable without copying (copy-on-write). Therefore, if the same executable or shared library is needed in multiple processes, the code and read-only data exist in memory just once; only the read-write data is duplicated on demand.

  - memory size > file size ⇒ 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 357] 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

[Slide 358] Loading a Binary to Memory

- Load ELF header and program header
- If `ET_DYN` (⇝ PIE), set random base added to all addresses
- Look if `PT_INTERP` is present
  - If present, load interpreter using same algorithm (but no nested interpreters)
- Iterate over `PT_LOAD` and `mmap` segments
  - May needs zeroing of last page and mapping extra zero pages

- Setup initial stack frame and auxiliary vector (e.g., with phdr address)
- Start execution at (the interpreter's) entry

This is the kernel's job

11.3. Linker Optimizations

[Slide 360] Eliminating Duplicate Strings/Constants

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

[Slide 361] 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 inline{1,2}.o
```

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

Solution on page 199.

[Slide 362] Linker Garbage Collection

- Problem: objects may contain unused functions
  - Compiler can't know whether function is used

> This is relevant for, e.g., static libraries. To avoid linking unneeded functions into the executable, some projects like musl put every function into a separate source file (⇒ separate object, separate sections). The more practical approach, however, is to instruct the compiler to emit every function/object into a separate section.

- 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 ⇝ possibly less efficient code
- GCC/Clang `-ffunction-sections`, `ld --gc-sections`

[Slide 363] 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 364] Link-Time Optimization

- Problem: Compilers still suckno optimizations across object files
  - Inlining, constant propagation+cloning, specialized call conv., ...
  - Optimization across language boundaries
- Idea 1: glue all source code together, compile with `-fwhole-program`
  - Downside: single core, problematic with same-name `static` functions
- Idea 2: Use static binary optimization during linking (severely limited)
- Idea 3: dump IR into object, glue IR together, optimize (`-flto`)
  - Done as very first step at link-time
- LTO is widely used and highly effective

[Slide 365] Link-Time Optimization in LLVM

- During object file compilation: just run simplification pipeline

- Fat LTO: all LLVM-IR modules from objects merged into single module
  - All existing optimizations can be applied, (conceptually) best performance
  - But: single-threaded compilation makes this not viable
- Thin LTO[5]: do cross-object analysis only on summary
  - Every object has a summary index: function sizes, linkage, call graph
  - Serial cross-module analysis: identify inlining candidate, other global opt.
  - Parallel optimization/compilation: apply optimizations found in analysis

11.4. Static Libraries

[Slide 367] 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

11.5. Shared Libraries

[Slide 369] 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 370] Shared Libraries: Changes in Compiler

None $\ddot\smile$ (almost)

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

[5] T Johnson, M Amini, and XD Li. “ThinLTO: scalable and incremental LTO”. in: *CGO*. 2017, pp. 111–121

[Slide 371] 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 372] 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 373] `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 374] Symbol Lookup

- Symbol lookup using linear search + `strcmp` is slow

- Idea: linker creates hash table
  - Hash symbol names and store them in hash table
  - Dynamic symbols grouped by hash bucket
  - Additional bloom filter to avoid useless walks for absent symbols
- Lookup:
  - Compute hash of target symbol string
  - Check bloom filter, if absent: abort
  - Iterate through symbols in bucket, compare names (and version)
- Documentation unfortunately sparse[6]

[Slide 375] Miscellaneous Things

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

> Symbols can be provided multiple times in different versions, permitting programs linked against an older version of the shared library to continue working, even if the ABI of the shared library changed in newer versions.

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

[Slide 376] Object Files, Linker, and Loader – Summary

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

[Slide 377] Object Files, Linker, and Loader – Questions

- Which ELF file types exist? What is different?

[6] A Roenky. *ELF: better symbol lookup via DT_ GNU_ HASH.* URL: `https://flapenguin.me/elf-dt-gnu-hash` (visited on 12/14/2022)

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

12. Unwinding and Debuginfo

[Slide 379] 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 380] 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

12.1. Stack Unwinding

[Slide 381] 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 — this improves makes debugging easier, as the debugger will stop at the frame of the `throw` for an unhandled exception. 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 (e.g., forced by another thread) is slightly different and skips the search phase.

[Slide 382] 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

[Slide 383] Stack Unwinding: `setjmp`/`longjmp`

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

+ Needs no extra information, small binary footprint

− High overhead in non-exceptional case

> The only widespread platform that uses `setjmp`/`longjmp` unwinding is Darwin on 32-bit ARM (i.e., iOS) for historical reasons — when Darwin was originally ported to 32-bit ARM, GCC didn't support zero-cost exceptions on ARM.[a]
>
> [a] Source: `https://lists.llvm.org/pipermail/cfe-dev/2013-October/032637.html`

[Slide 384] Stack Unwinding: Frame Pointer

- Frame pointers allow for fast unwinding
- `fp` points to stored caller's `fp`
- Return address stored adjacent to frame pointer

+ Fast and simple, also without exception

- Not all programs have frame pointers
  - Overhead of creating full stack frame
  - Causes loss of one register (esp. x86)
- Not generally possible to restore callee-saved registers
- Not sufficient!

```
x86_64:
  push rbp
  mov rbp, rsp
  // ...
  mov rsp, rbp
  pop rbp
  ret

aarch64:
  stp x29, x30, [sp, -32]!
  mov x29, sp
  // ...
  ldp x29, x30, [sp], 32
  ret
```

[Slide 385] 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: $retaddr - 1$ ~`pc` and CFA ~`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 layout
  - Stack frame varies throughout function, e.g. prologue, stack arguments
- Case 1: some register used as frame pointer – CFA constant offset to `fp`
  - E.g., for variable stack frame size, stack realignment on function entry
- Case 2: no frame pointer: CFA is constant offset to `sp`

⇝ Unwinding *must* restore register values
  - Frame pointer of caller, caller's landing pad can use callee-saved regs
  - Need to know where return address is stored (to identify caller)

12.2. Call Frame Information

[Slide 386] 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 387] Call Frame Information – Example 1

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

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

> 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 388] 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 389] 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 390] Call Frame Information – Exercise

In-Class Exercise:

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

Solution on page 199.

[Slide 391] 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 now 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 ofset 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

[a] DWARF Debugging Information Committee. *DWARF Debugging Information Format Version 5.* Feb. 2017. URL: `http://dwarfstd.org/doc/DWARF5.pdf`.

[Slide 392] Call Frame Information: Bytecode – Example 1

		CFA	rip	rbx
	foo:			
0:	push rbx	rsp+8	[CFA-8]	
1:	mov ebx, edi	rsp+16	[CFA-8]	[CFA-16]
3:	call bar	rsp+16	[CFA-8]	[CFA-16]
8:	mov eax, ebx	rsp+16	[CFA-8]	[CFA-16]
a:	pop rbx	rsp+16	[CFA-8]	[CFA-16]
b:	ret	rsp+8	[CFA-8]	[CFA-16]

```
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: RBX -16
DW_CFA_advance_loc: 10
DW_CFA_def_cfa_offset: +8
```

[Slide 393] Call Frame Information: Bytecode – Example 2

		CFA	rip	rbp
	foo:			
0:	push rbp	rsp+8	[CFA-8]	
1:	mov rbp, rsp	rsp+16	[CFA-8]	[CFA-16]
4:	shl rdi, 4	rbp+16	[CFA-8]	[CFA-16]
8:	sub rsp, rdi	rbp+16	[CFA-8]	[CFA-16]
b:	mov rdi, rsp	rbp+16	[CFA-8]	[CFA-16]
e:	call bar	rbp+16	[CFA-8]	[CFA-16]
13:	leave	rbp+16	[CFA-8]	[CFA-16]
14:	ret	rsp+8	[CFA-8]	[CFA-16]

```
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: 16
DW_CFA_def_cfa: RSP +8
```

[Slide 394] Call Frame Information: Bytecode – Example 3

		CFA	rip
	foo:		
0:	sub rsp, 8	rsp+8	[CFA-8]
4:	test edi, edi	rsp+16	[CFA-8]
6:	js 0x12	rsp+16	[CFA-8]
8:	call positive	rsp+16	[CFA-8]
d:	add rsp, 8	rsp+16	[CFA-8]
11:	ret	rsp+8	[CFA-8]
12:	call negative	rsp+16	[CFA-8]
17:	add rsp, 8	rsp+16	[CFA-8]
1a:	ret	rsp+8	[CFA-8]

```
DW_CFA_def_cfa: RSP +8
DW_CFA_offset: RIP -8
DW_CFA_advance_loc: 4
DW_CFA_def_cfa_offset: +16
DW_CFA_advance_loc: 13
DW_CFA_remember_state:
DW_CFA_def_cfa_offset: +8
DW_CFA_advance_loc: 1
DW_CFA_restore_state:
DW_CFA_advance_loc: 8
DW_CFA_def_cfa_offset: +8
```

Remember stack: {}

[Slide 395] Call Frame Information: Bytecode – Exercise

In-Class Exercise:

- For the functions in `ex12.txt`:
  encode your CFI tables in DWARF CFI bytecode
- Can you reduce the size of the bytecode by changing or omitting instructions while maintaining correctness?

Solution on page 200.

- Some ABIs guarantee a *red zone* below stack pointer
  - Red zone: area guaranteed to not be clobbered by signal handlers
  - E.g. x86-64 System V: 128 byte

> The red zone permits two optimizations:
>
> - For leaf functions with a stack frame smaller than 128 bytes, the instructions to adjust the stack pointer can be omitted.
> - In the CFI table, the rows corresponding to the function epilogue don't need to be updated to *same* if their storage location is part of the red zone. This reduces the size of the bytecode.

[Slide 396] Call Frame Information: Bytecode

- DWARF[1] 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 397] 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

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

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

- 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 398] 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 399] Call Frame Information: Assembler Directives – Example

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

gcc -O -S foo.c

```
        .globl foo
        .type foo, @function
foo:
        .cfi_startproc
        push rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        mov rbp, rsp
        .cfi_def_cfa_register 6
        shl rdi, 4
        sub rsp, rdi
        mov rdi, rsp
        call bar
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
        .size foo, .-foo
```

[Slide 400] 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 401] Asynchronous vs. Synchronous Unwind Info

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

> Windows requires asynchronous unwind info: with structured exception handling (SEH), it is possible to not just catch C++ exceptions, but also several other exceptional conditions like null-pointer dereferences (on UNIX-like systems, this is typically a segmentation fault signal).

[Slide 402] Unwinding: Other Platforms

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

Needs to work reliably for exception handling

[Slide 403] Compact Unwind Information I

- Problem: DWARF bytecode is rather repetitive and large
  - Many prologues have largely same structure, resulting in similar bytecode
- Problem: DWARF bytecode is not cheap to interpret
- Idea 1: represent each function with a fixed-size descriptor often sufficient for synchronous unwind info
- Code generator ensures that stack frame layout is same at all calls
- Restrict location of callee-saved registers due to limited encoding

+ Much more compact, typically 4-5 bytes per function

− Synchronous only, imposes code generator restrictions

[Slide 404] Compact Unwind Information II

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

[Slide 405] C++ Exceptions

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

12.3. Debug Information

[Slide 406] 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 407] 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 408] 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 410] DWARF: Hierarchical Program Description

- Extensible, flexible, Turing-complete[3] 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 411] DWARF: Data Types

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

DW_TAG_pointer_type [0xb1]
  DW_AT_byte_size (8)
  DW_AT_type      (0xb6 "char *")

DW_TAG_pointer_type [0xb6]
  DW_AT_byte_size (8)
  DW_AT_type      (0xbb "char")

DW_TAG_base_type [0xbb]
  DW_AT_byte_size (0x01)
  DW_AT_encoding  (DW_ATE_signed_char)
  DW_AT_name      ("char")
```

[Slide 412] DWARF: Variables

```
DW_TAG_variable [0xa3]
  DW_AT_name            ("x")
  DW_AT_decl_file       ("/path/to/main.c")
```

[3] 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.

```
  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 413] 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 414] 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 416] Other Debuginfo Formats

- DWARF is big despite compression
- Cannot run in time-constrained environments
  - Unsuited 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 417] 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 is 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 418] 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?

13. JIT Compilation and Sandboxing

13.1. JIT Compilation

[Slide 420] JIT Compilation

- Ahead-of-Time compilation not always possible/sufficient
- "Dynamic source" code: pre-compilation not possible
  - JavaScript, `eval()`, database queries
  - Binary translation of highly-dynamic/JIT-compiled code
- Additional verification/analysis or increased portability desired
  - (e)BPF, WebAssembly
- Dynamic optimization on common types/values
  - Run-time sampling of frequent code paths, allows dynamic speculation
  - Relevant for highly dynamic languages – otherwise prefer PGO[1]

[Slide 421] 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 422] 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

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

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

[Slide 423] JIT: Making Code Executable

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

[Slide 424] JIT: Making Code Executable

- Flush instruction cache
  - Flush DCache to unification point (last-level cache)
  - Invalidate ICache in *all* cores for virtual address range
    - After local flush, kernel might move thread to other core with old ICache
- x86: coherent ICache/DCache hierarchy – hardware detects changes
  - Also includes: transparent (but expensive) detection of self-modifying code
- AArch64, MIPS, SPARC, ... (Linux): user-space instructions
- ARMv7, RISC-V[2] (Linux), all non-x86 (Darwin): system call
- Skipping ICache flush: spurious, hard-to-debug problems

[Slide 425] Code Generation: Differences AoT vs. JIT

	Ahead-of-Time	JIT Compilation
Code Model	Arbitrary	Large or Small-PIC with custom PLT
Relocations	Linker/Loader	JIT compiler/linker
Symbols	Linker/Loader	JIT compiler/linker 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 426] JIT: Code Model

- Code can be located anywhere in address space
  - Cannot rely on linker to put in, e.g., lowest 2 GiB

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

- Large code model: allows for arbitrarily-sized addresses
- Small-PIC: possible for relocations inside object
  - Needs new PLT/GOT for other symbols
- Overhead trade-off: wide immediates vs. extra indirection (PLT)

  > On x86-64, the large code model implies a substantial cost, as every function call is indirect. The small-PIC code model is typically slightly more efficient for call-heavy code.

- 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 427] 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 428] 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 429] JIT: `.eh_frame` Registration (required for C++)

- Unwinder finds `.eh_frame` using program headers

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

[Slide 430] 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 431] 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 432] JIT: Linux `perf` JITDUMP format (optional)

- JIT-compiler dumps function name/address/size/code[4]
  - JITDUMP file: record list for each function, may contain debuginfo
  - File name must be `jit-<PID>.dump`
- JIT-compiler `mmaps` part of the file as executable somewhere
  - Only use: perf keeps track of executable mappings ⇝ mapping is JIT marker, s.t. perf can find the file later
- Need to run `perf report` with `-k 1` to use monotonic clock

[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`

- 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 433] Compilation Time

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

[Slide 434] 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 435] 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 436] 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 437] 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”

13.2. Sandboxing

[Slide 438] 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 439] 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 440] 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 441] 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 442] Approach: Sandbox Native Code with Modification

- Software-based Fault Isolation (SFI)
- Idea: enforce limitations on machine code
  - Define restrictions on machine code, e.g. no unbounded memory access
  - Modify compiler to comply with restrictions
  - Verify program at load time
- Verifier ensures: no out-of-bounds accesses, no jump to unverified code
- Google Native Client[5], originally x86-32, ported to x86-64 and ARM
- Designed as browser extension
- Native code shipped to browser, executed after validation

[Slide 443] NaCl Constraints on i386

- Problem: dynamic code not verifiable
  ⇒ No self-modifying/dynamically generated code

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

- 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

[Slide 444] NaCl on non-i386 Systems

- Other architectures[6] use base register instead of segment offsets
  - – Additional verification required
- Deprecated in 2017 in favor of WebAssembly
- Similar approach recently revived under new name LFI[7]

+ 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 445] 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 446] 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

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

[7] Z Yedidia. "Lightweight fault isolation: Practical, efficient, and secure software sandboxing". In: *ASPLOS*. 2024, pp. 649–665

- asm.js[8]: 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 447] asm.js Example

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

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

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

  return +sum;
}
```

Example taken from the specification

[Slide 448] 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 449] Approach: Using Bytecode – WebAssembly

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

[8] D Herman, L Wagner, and A Zakai. *asm.js*. 2014. URL: `http://asmjs.org/spec/latest/`.

  - No irreducible control flow representable

[Slide 450] 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 451] 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 452] 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?

14. Binary Translation

14.1. Overview

[Slide 454] Motivation

- Run program on other architecture
- Use-case: application compatibility
    - Other architecture with incompatible instruction encoding
    - Applications using unavailable ISA extensions[1]
- Use-case: architecture research
    - Development of new ISA extensions without existing hardware

[Slide 455] ISA Emulation

- Simplest approach: interpreting machine code
    - Simulate individual instructions, don't generate new code
- Frequently used approach before JIT-compilation became popular

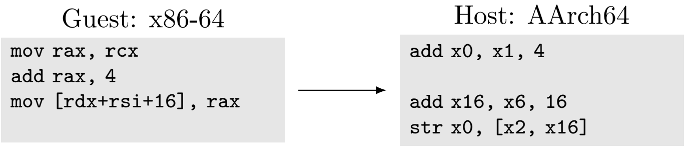
+ Simple, works almost anywhere, high correctness

– Very inefficient

14.2. Binary Translation

[Slide 456] 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



Guest: x86-64

```
mov rax, rcx
add rax, 4
mov [rdx+rsi+16], rax
```

Host: AArch64

```
add x0, x1, 4

add x16, x6, 16
str x0, [x2, x16]
```

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

[1] Exception-based implementation possible, but slow.

[Slide 457] 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 458] 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 459] Dynamic Binary Translation

See Figure 14.1.

- 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 460] Dynamic Binary Translation: Code Fragment

RISC-V Code

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

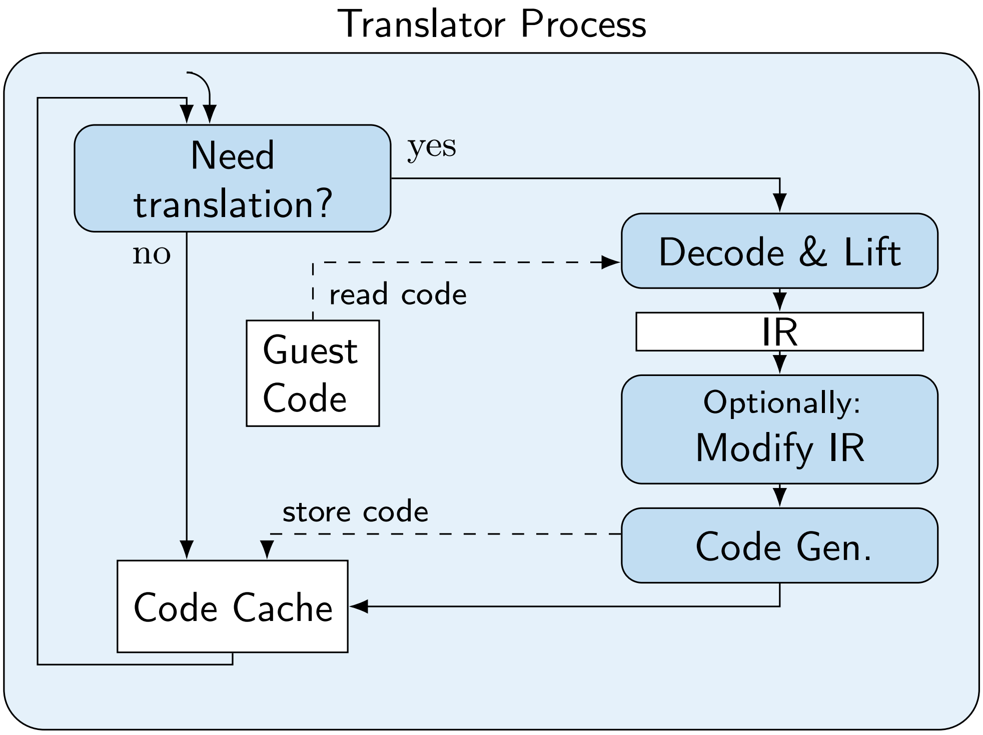


Figure 14.1.: Overview of a dynamic binary translator.

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.

14.3. Guest State

[Slide 461] 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 462] 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

14.4. Fast DBT

[Slide 463] 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 464] 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 465] Chaining

- Observation: many basic blocks have constant successors
  - Often conditional branches with fallthrough and constant offset
- (Hash)map lookup and indirect jump after everyblock 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 466] 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 467] 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 468] 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 469] 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

14.5. Correct DBT

[Slide 470] 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 471] POSIX Signals

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

[Slide 472] 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 473] 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 474] 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 475] 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 476] Correct DBT: Floating-point

- Floating-point arithmetic is standardized in IEE-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 477] Correct DBT: OS and CPU Specifics

- Emulating all syscalls correctly is hard
  - Version-specifics, structure layouts, feature support
  - Huge interface
- `/proc/self/*` – how to emulate?
  - Catch all file system accesses? Follow all possible symlinks?
  - What if procfs is mounted somewhere else?
- `cpuid` – how to emulate?
  - Cache sizes, processor model, ...
  - Application can do timing experiment to detect DBT

[Slide 478] 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 479] 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?

15. Query Compilation

15.1. Overview

[Slide 481] 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 482] 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 483] 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.

15.2. Subscripts

[Slide 484] 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: $\bowtie^{HJ}_{s.matrnr=h.matrnr}$, $\sigma_{a.x<5\cdot(b.y-a.z)}$

[Slide 485] Subscripts: Execution

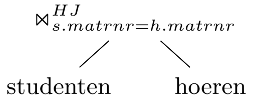
- Option: keep as tree, interpret
  - + Simple, flexible
  - − Slow
- Option: compile to bytecode
  - + More efficient
  - − More effort to implement, some compile-time
- Option: compile to machine code
  - − Code can be complex to accurately represent semantics
  - + Most efficient
  - − Most effort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

[Slide 486] 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

15.3. Query Execution

[Slide 487] Query Execution: Simplest Approach



- Execute operators individually
- Materialize all results after each operator
- “Full Materialization”

+ Easy to implement
+ Can dynamicnamically adjust plan
− Inefficient, intermediate results can be big

[Slide 488] Iterator Model[1]

- 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 489] 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;
    }
```

[1] G 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 490] 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 491] Push-based Model[2]

- 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 492] 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;
```

[2] T 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 493] 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 494] Pull-based Model vs. Push-based Model[3]

- 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 495] 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

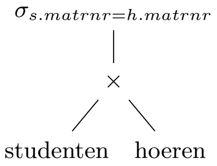
[Slide 496] Code Generation for Push-Based Model

- Inlining code in push-based model yields nice code

[3] A 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 497] Code Generation: Example



```
struct Query {
  Output out;
  Table tabLeft, tabRight;
  Tuple* curLeft = nullptr;
  void produce() {
    for (Tuple* tl : tabLeft) {
      curLeft = tl;
      for (Tuple* tr : tabRight) {
        Tuple* t = concat(curLeft, tr);
        if (t.s_matrnr == t.h_matrnr)
          out.write(t);
      }
    }
  }
};
```

[Slide 498] How to Generate Code

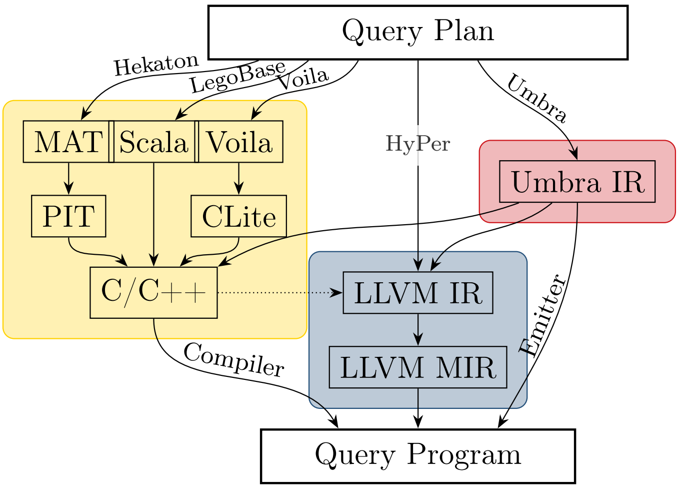
- Code generator executes produce/consume methods
  - Method bodies don't do actual operations, but construct code
  - E.g., call `IRBuilder`
  - Call to helper functions for complex operations e.g. hash table insert/lookup, string operations, memory allocation, etc.
- Resulting code doesn't contain 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 499] 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)

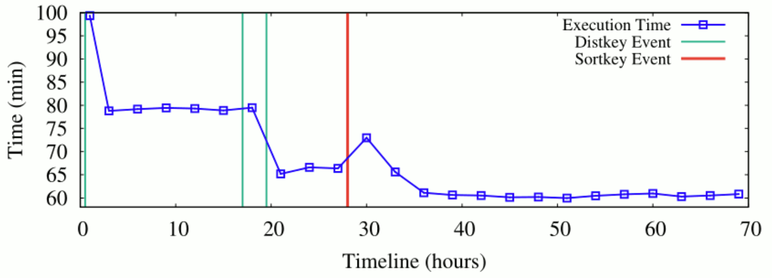
[Slide 500] What to Generate



[Slide 501] Case Study: Amazon Redshift[4]

"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."

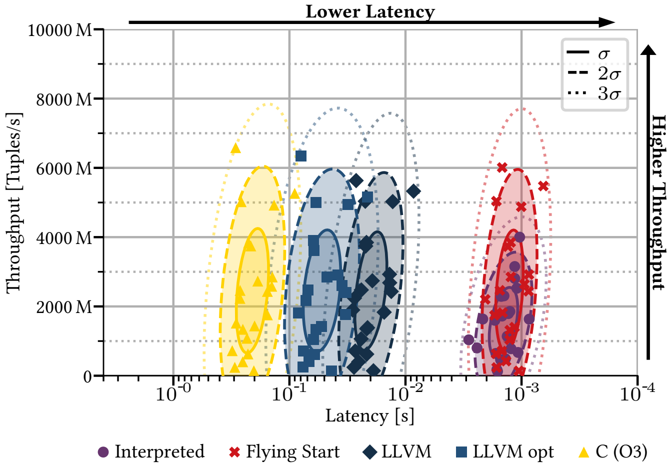
[Slide 502] Case Study: Amazon Redshift[5]



[4] N Armenatzoglou et al. "Amazon Redshift Re-invented". In: *SIGMOD*. 2022.
[5] N 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 work- load 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 503] Compile Times: Umbra



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

[Slide 504] 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 505] 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 506] 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-interpred vec. execution
  - But: a good vectorized engine is not necessarily *slow*
- Vectorized execution probably more popular than code generation

[Slide 507] 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 508] 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 22]

Some suggestions:

- Sequences of `+`/`-`/`<`/`>`
- `[-]` → set zero
- `[>]` → find next zero (`memchr`)
- `[->+>+«]` → add to next two siblings, set zero
- `[->+++<]` → add 3 times to next sibling, set zero
- ...

[Slide 28]

Indirect/direct threading and computed `goto`/tail calls are orthogonal.

- Computed `goto` is a non-standard feature. A lot of state can be kept in registers across different bytecode operations, this is done automatically by the compiler. The possibility of indirect jumps inside the function may prevent some compiler optimizations.
- Tail calls (largely) rely on standard features and the code is typically more maintainable (e.g., operations can be implemented in separate functions). Some state can be kept in registers across operations, but this has to be implemented manually; the usable number of preservable registers is defined by the calling convention and is architecture-dependent. Non-standard calling convention (e.g., ghccc) can be used to increase this limit. Compilers cannot perform optimization across the different functions.
- Indirect threading permits a small opcode (e.g., one byte as in this example), which leads to higher code density and therefore increased data locality.
- Direct threading avoids the indirection to the opcode handler at the cost of larger bytecode (in the example, an operation grows from 2 to 16 bytes). On modern out-of-order CPUs, avoiding the indirection might not lead to significant performance differences.

[Slide 33]

As a simple approach, check the array boundaries at every pointer move, when reaching bounds, re-allocate to larger buffer and copy the elements. As pointer moves occur often, the bounds checks are rather expensive.

To improve efficiency, it is possible to use virtual memory by padding the buffer with known-unmapped pages (guard pages) and intercepting accesses to these pages (e.g., through a signal handler). On an access, the buffer can grow and/or be moved (e.g., through `mremap`). Care must be taken to ensure that the guard pages are large enough (the length of the program should be sufficient).

[Slide 52]

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/–)		

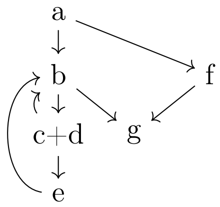
[Slide 60]

There are two ways of implementing a scoped hash table:

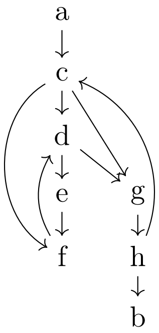
- Chain of hash maps: $Scope = (Map[Name \to Type]\ names, Scope\ parent)$. This is, however, very slow for deeply nested scopes, as all hash maps of the parent scopes must be queried. Hash map lookups are fairly expensive.
- Hash map of lists: $Map[Name \to List[Tuple[Depth, Type]]]$. For every identifier, 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 83]

Control flow graph for `fn1`:



Control flow graph for `fn2`:



[Slide 99]

```
phis(v1, v2) {
entry:
    v3 = mul v1, v2 // a * b
    v4 = mul v2, v2 // b * b
    v5 = cmpgt v1, v4 // a > b * b
    br v5, ifthen, ifelse
ifthen:
    br header
header:
    phi1 = phi(ifthen: v1, body: v7) // a
    phi2 = phi(ifthen: 1, body: phi2) // c
    v6 = cmpgt phi1, 0 // a > 0
    br v6, body, cont
body:
    v7 = sub phi1, phi2 // a - c
    br header
cont:
    br ret
ifelse:
    v8 = mul v2, v2 // b * b
    br ret
ret:
    phi3 = phi(cont: phi1, ifelse: v8) // a
    ret phi3
}

swap(v1, v2, v3) {
entry:
    br header
header:
    phi1 = phi(entry: v3, body: v5) // c
    // Note: all phi nodes execute concurrently. Therefore, these two phi nodes swap a and b.
    phi2 = phi(entry: v1, body: phi3) // a
    phi3 = phi(entry: v2, body: phi2) // b
    v4 = cmpgt phi1, 0 // a > 0
    br v4, body, cont
body:
    v5 = sub phi1, 1 // c - 1
    br header
cont:
    ret phi2
}
```

[Slide 128]

```
int sw(int x) {
  switch (x) {
  case 4:
  case 100: return 12;
  case 5: return 32;
  case 8: return 9;
  default: return x;
  }
}
```

[Slide 129]

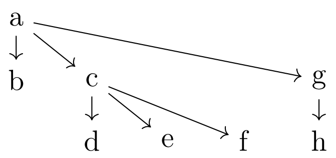
```
int sw(int x) {
  switch (x) {
  case 4:
  case 10: return 12;
  case 5: return 32;
  case 8: return 9;
  default: return x;
  }
}
```

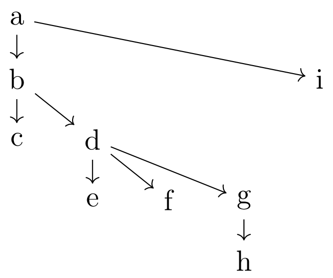
[Slide 149]

The algorithm never visits the `%else` block and therefore can prove that `%j4` and thus `%j2` are always the constant 1.

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

[Slide 157]





[Slide 171]

The first CFG has no reducible loop, so the algorithm as written will not find any loops.

The second CFG contains two reducible loops:

- Loop 1: `d` (header), `e`, `f`, `g`
- Loop 2: `b` (header), `c`, `d`, `e`, `f`, `g`, `h`

[Slide 189]

You can use Alive2 as validation tool: `https://alive2.llvm.org/ce/z/_PuqaK`

1. The neutral element of floating-point addition is -0.0; adding $+0.0$ to -0.0 results in $+0.0$, which is not a generally correct transformation.
2. This transformation, although non-obvious, is correct.
3. `sdiv` rounds towards zero, but `ashr`, so for e.g. -3, `src3` returns 0 while `tgt3` return -1.

[Slide 194]

You can use Alive2 as validation tool: `https://alive2.llvm.org/ce/z/HgpRmK`

1. Correct: `udiv exact` results in `poison` when 1 is not a multiple of the parameter, i.e., when the parameter is not 1. Thus, always returning 1 makes the target function more defined than the source function.
2. Not correct: `%cmpeq` is `poison` if `%x` is negative. The transformation would be correct if the `samesign` flag was dropped.
3. Not correct: integer addition is associative, but the `nsw` flag needs to be dropped.

[Slide 219]

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:
  paddd 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 222]

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 223]

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 259]

Dynamic Programming Table:

Node	$+_1$	$\ll_1$	4	$*_1$	ld_1	$+_2$	$*_2$	8	ld_2	$+_3$	$\ll_2$	3
Cost	13	1	1	12	6	4	4	1	3	2	1	1
Pattern	P_9	P_0	P_{10}	P_8	P_4	P_9	P_8	P_{10}	P_6	P_1	P_0	P_{10}

Resulting instructions:

```
lsl r0, rW, 4
mov r1, 8
madd r2, rY, r1, rX
ldr r3, [r2]
ldr r4, [rZ, rY, lsl #3]
madd r5, r3, r4, r0
```

[Slide 294]

1. Dependencies between ϕ-nodes on edge `b3`→`b2`: `%4`→`%3`, `%3`→`%4` (cycle), `%5`→`%3`. This is also the only edge where dependencies can occur.
2. Critical edges: `b2`→`b4`, `b3`→`b4`, `b3`→`b2`

ϕ resolution after splitting critical edges:

```
fn_crit_edges_broken(%0, %1) {
b1:
  %2 = add %0, %1
  %3 = %1
  %4 = %0
  %5 = %2
  %6 = 0
  br %b2

b2:
  %7 = icmp lt %3, %6
  br %7, %b3, %b2b4

b3:
  %8 = add %6, 1
  %9 = icmp gt %8, %1
  br %9, %b3b4, %b3b2

b4:
  %12 = add %10, %11
  ret %12

b2b4:
  %10 = %4
  %11 = %5
  br %b4
b3b4:
  %10 = %3
  %11 = %8
  br %b4
b3b2:
  %5 = %3
  %t = %3
  %3 = %4
  %4 = %t
  %6 = %8
  br %b2
}
```

ϕ resolution by copying values used afterwards:

```
fn_copy_values(%0, %1) {
```

```
b1:
  %2 = add %0, %1
  %3 = %1
  %4 = %0
  %5 = %2
  %6 = 0
  br %b2

b2:
  %7 = icmp lt %3, %6
  %10 = %4
  %11 = %5
  br %7, %b3, %b4

b3:
  %8 = add %6, 1
  %9 = icmp gt %8, %1
  %t1 = %3 ; copy %3, needed afterwards
  %5 = %3
  %t = %3
  %3 = %4
  %4 = %t
  %6 = %8
  %10 = %t1
  %11 = %8
  br %9, %b4, %b2

b4:
  %12 = add %10, %11
  ret %12
}
```

[Slide 304]

Post-order traversal: $b6, b5, b4, b3, b2, b1$.

		Iteration 1	Iteration 2	Iteration 3	Iteration 4
$b1$	live-in	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
	live-out	a	a	a	a
$b2$	live-in	a	a	a	a
	live-out	b	a, b	a, b	a, b
$b3$	live-in	b	a, b	a, b	a, b
	live-out	b	a, b	a, b	a, b
$b4$	live-in	c	b, c	a, b, c	a, b, c
	live-out	d	b, d	a, b, d	a, b, d
$b5$	live-in	d	d	b, d	a, b, d
	live-out	e	e	b, e	a, b, e
$b6$	live-in	e	e	e	e
	live-out	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Observation: the algorithm needs one iteration to propagate a value only used in the loop header for every loop nesting depth.

[Slide 318]

Key idea of the algorithm: treat the whole function as a single basic block.

1. Liveness intervals without holes are used to determine when a register can be free'ed which value is spilled.
2. Unfortunate block ordering impacts the length of the live intervals, but not the correctness.
3. Furthest end of live interval first.
4. This is not mentioned in the paper.
5. See next slide.

[Slide 352]

- Function calls always consist of two instructions, `auipc` and `jalr`, and are always encoded as indirect jumps. There are two relocations per function call: the call relocation, which informs the linker about the call target, and at the same position a *relax* relocation, which gives the linker the opportunity to merge these two instructions into one; shortening the code by 4 bytes.
- At the end of a function, there is an *align* relocation, which enforces a realignment of the subsequent code after possible previous linker relaxations.
- The linker relaxed the function call to a single `jal` instruction.
- As a consequence of linker relaxation, almost all branches even inside functions need relocations.

Note: the relax and align relocations are specific to RISC-V and LoongArch. Other architectures don't do this.

[Slide 361]

- `inline` functions get compiled in every object file. To facilitate deduplication, the definitions are stored in separate sections. These sections form a *COMDAT group* (described by the accompanied `.group` section), instructing the linker to keep exactly one group with the same symbol and discard the others.
- The final executable only contains one definition of `x`.
- Symbols of inline functions have weak linkage to avoid errors about multiple conflicting definitions.

[Slide 390]

`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 395]

```
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.

[Arm+22] N Armenatzoglou, S Basu, N Bhanoori, M Cai, N Chainani, K Chinta, V Govindaraju, TJ Green, M Gupta, S Hillig, et al. "Amazon Redshift Reinvented". 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.

[Bel73] JR Bell. "Threaded Code". In: *CACM* 16.6 (1973), pp. 370–372. URL: https://dl.acm.org/doi/pdf/10.1145/362248.362270.

[Boi+11] B Boissinot, F Brandner, A Darte, BD de Dinechin, and F Rastello. "A non-iterative data-flow algorithm for computing liveness sets in strict SSA programs". In: *ASPLAS.* 2011, pp. 137–154.

[BR22] B Boissinot and F Rastello. "Liveness". In: *SSA-based Compiler Design.* Ed. by F Rastello and F Bouchez Tichadou. 2022, pp. 107–122. DOI: 10.1007/978-3-030-80515-9_9.

[Bra+13] M Braun, S Buchwald, S Hack, R Leißa, C Mallon, and A Zwinkau. "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.

[Cha82] GJ Chaitin. "Register allocation & spilling via graph coloring". In: *SIGPLAN* 17.6 (1982), pp. 98–101. URL: https://dl.acm.org/doi/pdf/10.1145/872726.806984.

[CHK06] KD Cooper, TJ Harvey, and K Kennedy. "An empirical study of iterative data-flow analysis". In: *CIC.* 2006, pp. 266–276.

[Cla+57] W Clark, J Frankovich, H Peterson, J Forgie, R Best, and K Olsen. *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, J Ferrante, BK Rosen, MN Wegman, and FK Zadeck. "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.

[EG03] MA Ertl and D Gregg. "The structure and performance of efficient interpreters". In: *JILP* 5 (2003), pp. 1–25. URL: http://www.jilp.org/vol5/v5paper12.pdf.

[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. URL: https://www.cs.princeton.edu/techreports/2005/737.pdf.

[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/.

[JAL17] T Johnson, M Amini, and XD Li. "ThinLTO: scalable and incremental LTO". In: *CGO*. 2017, pp. 111–121.

[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.

[Kil73] GA Kildall. "A unified approach to global program optimization". In: *POPL*. 1973, pp. 194–206. URL: https://dl.acm.org/doi/pdf/10.1145/512927.512945.

[KLN18] A Kohn, V Leis, and T Neumann. "Adaptive execution of compiled queries". In: *ICDE*. 2018, pp. 197–208. URL: https://db.in.tum.de/~leis/papers/adaptiveexecution.pdf.

[KU76] JB Kam and JD Ullman. "Global data flow analysis and iterative algorithms". In: *JACM* 23.1 (1976), pp. 158–171. URL: https://dl.acm.org/doi/pdf/10.1145/321921.321938.

[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.

[Lop+21] NP Lopes, J Lee, CK Hur, Z Liu, and J Regehr. "Alive2: bounded translation validation for LLVM". In: *PLDI*. 2021, pp. 65–79. URL: https://users.cs.utah.edu/~regehr/alive2-pldi21.pdf.

[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, M Matz, M Girkar, J Hubička, A Jaeger, and M Mitchell. *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. URL: https://dl.acm.org/doi/pdf/10.1145/330249.330250.

[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.

[Ras12] F Rastello. “On Sparse Intermediate Representations: Some Structural Properties and Applications to Just-In-Time Compilation”. Habilitation thesis. Inria Grenoble Rhône-Alpes, 2012. URL: https://inria.hal.science/hal-00761555/file/habilitation.pdf.

[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, R Muth, C Biffle, V Khimenko, E Pasko, K Schimpf, B Yee, and B Chen. “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 01/07/2026).

[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.

[Tro+21] M Trofin, Y Qian, E Brevdo, Z Lin, K Choromanski, and D Li. *MLGO: a Machine Learning Guided Compiler Optimizations Framework*. 2021. arXiv: 2101.04808 [cs.PL]. URL: https://arxiv.org/abs/2101.04808.

[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. URL: http://www.christianwimmer.at/Publications/Wimmer10a/Wimmer10a.pdf.

[WM05] C Wimmer and H Mössenböck. “Optimized interval splitting in a linear scan register allocator”. In: *VEE*. 2005, pp. 132–141.

[WZ91] MN Wegman and FK Zadeck. “Constant propagation with conditional branches”. In: *TOPLAS* 13.2 (1991), pp. 181–210. URL: https://dl.acm.org/doi/pdf/10.1145/103135.103136.

[Yed24] Z Yedidia. “Lightweight fault isolation: Practical, efficient, and secure software sandboxing”. In: *ASPLOS*. 2024, pp. 649–665.

[Yee+09] B Yee, D Sehr, G Dardyk, JB Chen, R Muth, T Ormandy, S Okasaka, N Narula, and N Fullagar. “Native client: A sandbox for portable, untrusted x86 native code”. In: *SP*. 2009, pp. 79–93.