Code Generation for Data Processing

Lecture 1: Introduction and Interpretation

Alexis Engelke

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

Winter 2022/23

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

Module "Code Generation for Data Processing"

Learning Goals

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

Prerequisites

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

Topic Overview

Topic Overview

Introduction

- Introduction and Interpretation
- Compiler Front-end

Topic Overview

Introduction

- Introduction and Interpretation
- Compiler Front-end

Intermediate Representations

- IR Concepts and Design
- LLVM-IR
- LLVM Transforms and Analyses

Topic Overview

Introduction

- Introduction and Interpretation
- Compiler Front-end

Intermediate Representations

- IR Concepts and Design
- LLVM-IR
- LLVM Transforms and Analyses

Compiler Back-end

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

Topic Overview

Introduction

- Introduction and Interpretation
- Compiler Front-end

Intermediate Representations

- IR Concepts and Design
- LLVM-IR
- LLVM Transforms and Analyses

Compiler Back-end

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

Applications

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

Lecture Organization

- Lecturer: Dr. Alexis Engelke `engelke@in.tum.de`
- Time slot: Thu 10-14, 02.11.018
- Material: `https://db.in.tum.de/teaching/ws2223/codegen/`

Exam

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

Exercises

- Weekly homework, often with programming exercise
- Submission via e-mail: `engelke+cghomework@in.tum.de`
  - Probably no explicit grading, feedback on best effort
- Exercise sessions to present and discuss solutions

Grade Bonus

- Requirement: $N - 2$ "sufficiently working" homework submissions **and** at least 2 presentations of homework in class
- Bonus: grades in $[1.3; 4.0]$ improved by 0.3

Why study compilers?

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

Compiler Lectures @ TUM

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

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

Why study code generation?

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, …

Why study code generation?

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

Proebsting's Law

Proebsting's Law

"Compiler advances double computing power every *18* years."

– Todd Proebsting, 1998[1]

[1] http://proebsting.cs.arizona.edu/law.html

Proebsting's Law

"Compiler advances double computing power every *18* years."

– Todd Proebsting, 1998[1]

- Still optimistic; depends on number of abstractions

[1] http://proebsting.cs.arizona.edu/law.html

Motivational Example: Brainfuck

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

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

- Execution with pen/paper? $\ddot\frown$

Program Execution

Program Execution

Program Execution

Programs

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

Program Execution

Programs

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

Hardware/ISA

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

Motivational Example: Brainfuck – Interpretation

- Write an interpreter!

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

Program Execution

Program Execution

Compiler

Program → Compiler → Program

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

Program Execution

Compiler

Program → Compiler → Program

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

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

Program Execution

Compiler

Program → Compiler → Program

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

Interpreter

Program → Interpreter → Result

- Directly execute program
- Computes program result

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

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)

Program Execution

Compiler

Program $\rightarrow$ [Compiler] $\rightarrow$ Program

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

- C, C++, Rust $\rightarrow$ machine code
- Python, Java $\rightarrow$ bytecode

Interpreter

Program $\rightarrow$ [Interpreter] $\longrightarrow$ Result

- Directly execute program
- Computes program result

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

- Multiple compilation steps can precede the "final interpretation"

Compilers

- Targets: machine code, bytecode, or other source language

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.

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.)

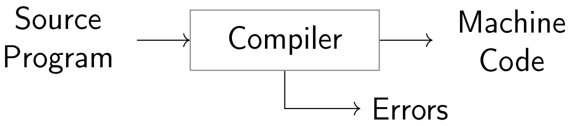
Compilers

- Targets: machine code, bytecode, or other source language
- Typical goals: better language usability and performance
  - Make lang. usable at all, faster, use less resources, etc.

- Constraints: specs, resources (comp.-time, etc.), requirements (perf., etc.)

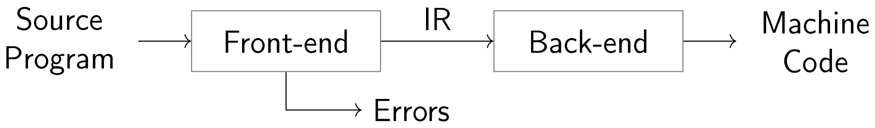
- Examples:
  - "Classic" compilers source $\rightarrow$ machine code
  - JIT compilation of JavaScript, WebAssembly, Java bytecode, ...
  - Database query compilation
  - ISA emulation/binary translation

Compiler Structure: Monolithic

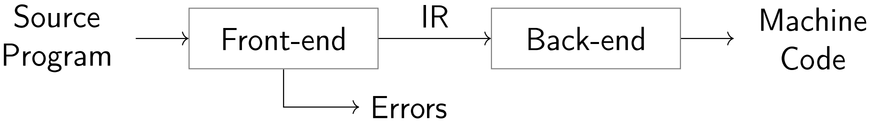


- Inflexible architecture, hard to retarget

Compiler Structure: Two-phase architecture



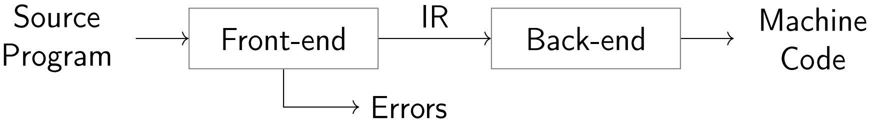
Compiler Structure: Two-phase architecture



Front-end

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

Compiler Structure: Two-phase architecture



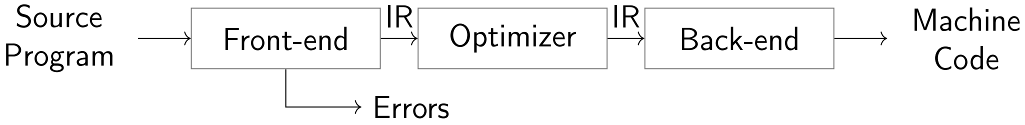
Front-end

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

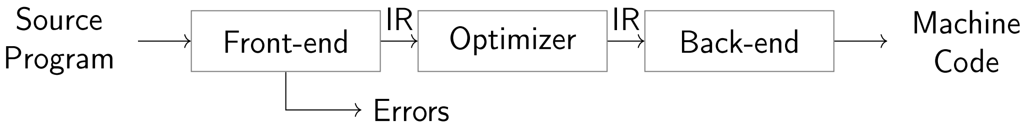
Back-end

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

Compiler Structure: Three-phase architecture

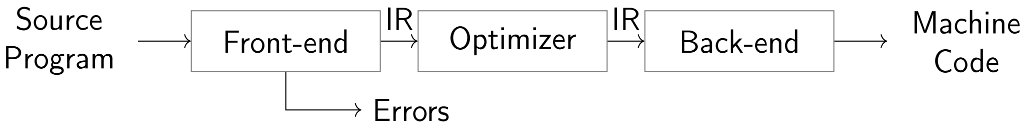


Compiler Structure: Three-phase architecture



- Optimizer: analyze/transform/rewrite program inside IR

Compiler Structure: Three-phase architecture



- Optimizer: analyze/transform/rewrite program inside IR

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

Compiler Front-end

Compiler Front-end

1. Tokenizer: recognize words, numbers, operators, etc. $\mathcal{R}e$
   - Example: `a+b*c` $\rightarrow$ `ID(a) PLUS ID(b) TIMES ID(c)`

Compiler Front-end

1. Tokenizer: recognize words, numbers, operators, etc. $\mathcal{R}e$
   - Example: `a+b*c` $\to$ `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"))`

Compiler Front-end

1. Tokenizer: recognize words, numbers, operators, etc. $\mathcal{Re}$
   - Example: `a+b*c` $\rightarrow$ `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.

Compiler Front-end

1. Tokenizer: recognize words, numbers, operators, etc. $\mathcal{R}e$
   - Example: `a+b*c` $\rightarrow$ `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

Compiler Back-end

Compiler Back-end

1. Instruction Selection: map IR operations to target instructions
   - Use target features: special insts., addressing modes, ...
   - Still using virtual/unlimited registers

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

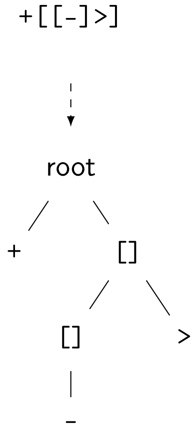
Compiler Back-end

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

Motivational Example: Brainfuck – Front-end

- Need to skip comments
- Bracket searching is expensive/redundant

- Idea: "parse" program!
- Tokenizer: yield next operation, skipping comments
- Parser: find matching brackets, construct AST



Motivational Example: Brainfuck – AST Interpretation

- AST can be interpreted recursively

```
struct node { char kind; int 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); break;
  case 0: children(n); break; // root
  }
}
void children(struct node* n, struct state* s) {
  for (int i = 0; i < n->cldCnt; i++) donode(n->cld + i, s);
}
```

Motivational Example: Brainfuck – Optimization

Motivational Example: Brainfuck – Optimization

- Inefficient sequences of +/-/</> can be combined
  - Trivially done when generating IR

Motivational Example: Brainfuck – Optimization

- Inefficient sequences of +/-/</> can be combined
  - Trivially done when generating IR
- Fold patterns into more high-level operations
  - `[-]` = set zero
  - `[>]` = find next zero (`memchr`)
  - `[->+>+«]` = add to next two siblings, set zero
  - `[->+++<]` = add 3 times to next sibling, set zero
  - ...

Motivational Example: Brainfuck – Optimization

- Fold offset into operation
  - `right(2) add(1)` = `addoff(2, 1) right(2)`
  - Also possible with loops

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

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.

Motivational Example: Brainfuck – Bytecode

Motivational Example: Brainfuck – Bytecode

- Tree is nice, but rather inefficient $\rightsquigarrow$ flat and compact bytecode
- Avoid pointer dereferences/indirections; keep code size small

Motivational Example: Brainfuck – Bytecode

- Tree is nice, but rather inefficient $\rightsquigarrow$ flat and compact bytecode
- Avoid pointer dereferences/indirections; keep code size small

- Superinstructions: combine common sequences to one instruction
- Maybe dispatch two instructions at once?
  - `switch (ops[pc] | ops[pc+1] << 8)`

Motivational Example: Brainfuck – Threaded Interpretation

Motivational Example: Brainfuck – Threaded Interpretation

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

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

    struct op* pc = ops;
    DISPATCH;

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

Fast Interpretation

- Key technique to “avoid” compilation to machine code

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

Fast Interpretation

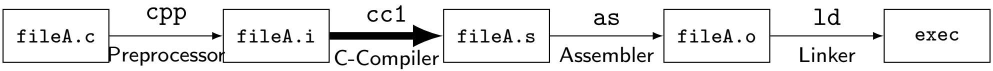
- Key technique to “avoid” compilation to machine code
- Preprocess program into efficiently executable bytecode
  - Easily identifiable opcode, homogeneous structure
  - Can be linear (fast to execute), but trees also work
- Perhaps optimize – if it’s worth the benefit
  - Fold constants, combine instructions, ...
  - Consider superinstructions for common sequences

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
- 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
- Use threaded-interpretation to avoid branch misses

Compiler: Surrounding – Compile-time

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



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

Compiler: Surrounding – Run-time

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

Motivational Example: Brainfuck – Runtime Environment

Motivational Example: Brainfuck – Runtime Environment

- Needs I/O for `.` and `,`

Motivational Example: Brainfuck – Runtime Environment

- Needs I/O for `.` and `,`
- Memory management: infinitely sized array
- Allocate on demand (easy?)

Motivational Example: Brainfuck – Runtime Environment

- Needs I/O for `.` and `,`
- Memory management: infinitely sized array
- Allocate on demand (easy?)
  - What if main memory or address space is insufficient?
- Deallocation of empty pages?

Motivational Example: Brainfuck – Runtime Environment

- Needs I/O for `.` and `,`
- Memory management: infinitely sized array
- Allocate on demand (easy?)
  - What if main memory or address space is insufficient?
- Deallocation of empty pages?
- Error handling: unmatched brackets

Compilation point: AoT vs. JIT

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()`

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

Compiler Design: Effect of Languages – Imperative

- Step-by-step execution of program modification of state
- Close to hardware execution model
- Direct influence of result

- Tracking of state is complex
- Dynamic typing: more complexity
- Limits optimization possibilities

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

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

Compiler Design: Effect of Languages – Declarative

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

- Allows for more optimizations
- Mapping to hardware non-trivial
  - Might need more stages
  - Preserve semantic info for opt!
- Programmer has less "control"

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

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

Introduction and Interpretation – Summary

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

Introduction and Interpretation – Questions

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

Code Generation for Data Processing

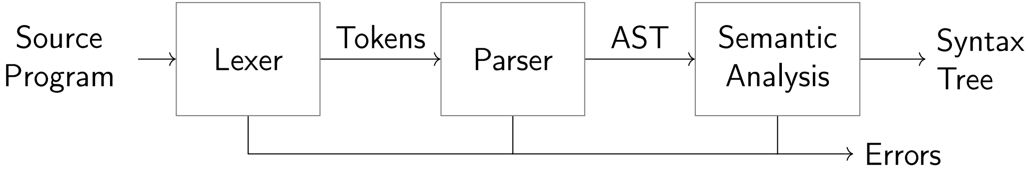
Lecture 2: Compiler Front-end

Alexis Engelke

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

Winter 2022/23

Compiler Front-end



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

Lexer

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

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

Typical Token Kinds

Typical Token Kinds

- Punctuators `( ) [ ] { } ; = + += | ||`
- Identifiers `abc123 main`
- Keywords `void int __asm__`
- Numeric constants `123 0xab1 5.7e3 0x1.8p1`
- 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`

Lexer Implementation

```python
def nextToken(inp: str) -> tuple[str, str, str]:
    # Get next token, return (kind, value, remainder)
    inp = inp.lstrip()
    if not inp:
        return "EOF", "", inp
    if inp[0].isdigit():
        m = re.match(r'[1-9][0-9]*|0([0-7]+|x[0-9a-fA-F]+|)', inp)
        return "NUM", m[0], inp[m.end():]
    if inp[0].isalpha():
        m = re.match(r'[a-zA-Z][a-zA-Z0-9_]*', inp)
        if m[0] in KEYWORDS: return m[0], m[0], inp[m.end():]
        return "IDENT", m[0], inp[m.end():]
    if inp[:2] == "+=": return "PLUSEQ", inp[:2], inp[2:]
    if inp[:1] == "+": return "PLUS", inp[:1], inp[1:]
    ...
    raise Exception()
```

Lexing C??=

```
main() <%
  // yay, this is C99??/
  puts("hi␣world!");
  puts("what's␣up??!");
%>
```

Output:

Lexing C??=

```
main() <%
  // yay, this is C99??/
  puts("hi␣world!");
  puts("what's␣up??!");
%>
```

Output: `what's up|`

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...

Lexer Implementation

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

Parsing

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

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

Context-Free Grammar (CFG)

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

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

Hand-written Parsing – First Try

- One function per non-terminal
- Check expected structure
- Return AST node

```
def parseBreakStmt(...):
    matchToken("break")
    matchToken("SEMICOLON")
    return ("breakStmt",)

def parseWhileStmt(...):
    matchToken("while")
    matchToken("LPAREN")
    expr = parseExpr(...)
    matchToken("RPAREN")
    stmt = parseStmt(...)
    return ("whileStmt", expr, stmt)

def parseStmt(...):
    # whoops!
```

Hand-written Parsing – First Try

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

```
def parseBreakStmt(...):
    matchToken("break")
    matchToken("SEMICOLON")
    return ("breakStmt",)

def parseWhileStmt(...):
    matchToken("while")
    matchToken("LPAREN")
    expr = parseExpr(...)
    matchToken("RPAREN")
    stmt = parseStmt(...)
    return ("whileStmt", expr, stmt)

def parseStmt(...):
    # whoops!
```

Hand-written Parsing – Second Try

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

$\Rightarrow$ Not all CFGs (easily) parseable (but most programming langs. are)

```
def parseBreakStmt(...):
    ... # as before
def parseWhileStmt(...):
    ... # as before

def parseStmt(...):
    tok = peekToken()
    if tok == "break":
        return parseBreakStmt(...)
    if tok == "while":
        return parseWhileStmt(...)
    expr = parseExpr(...)
    matchToken("SEMICOLON")
    return ("exprStmt", expr)
```

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

```
def parseBreakStmt(...):
    ... # as before
def parseWhileStmt(...):
    ... # as before

def parseStmt(...):
    tok = peekToken()
    if tok == "break":
        return parseBreakStmt(...)
    if tok == "while":
        return parseWhileStmt(...)
    expr = parseExpr(...)
    matchToken("SEMICOLON")
    return ("exprStmt", expr)
```

Ambiguity

$$expr \;\rightarrow\; expr \;\mathbf{+}\; expr \mid expr \;\mathbf{*}\; expr \mid expr \;\mathbf{=}\; expr \mid \mathbf{(}\; expr \;\mathbf{)} \mid \mathbf{number}$$

Input: $4 + 3 * 2$

Ambiguity

$$expr \rightarrow expr + expr \mid expr \; \mathbf{*} \; expr \mid expr = expr \mid \mathbf{(} \; expr \; \mathbf{)} \mid \mathbf{number}$$

Input: $4 + 3 * 2$

Ambiguity – Rewrite Grammar?

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

Ambiguity – Rewrite Grammar?

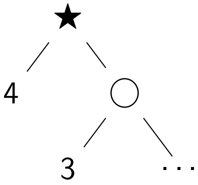
$$\begin{aligned} \textit{primary} &\rightarrow (\textit{ expr }) \mid \textbf{number} \\ \textit{expr} &\rightarrow \textit{primary} + \textit{expr} \mid \textit{primary} \textbf{ * } \textit{expr} \mid \textit{primary} = \textit{expr} \mid \textit{primary} \end{aligned}$$

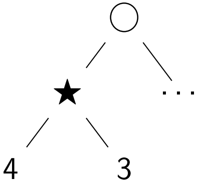
Input: $4 + 3 * 2$

Input: $4 * 3 + 2$

Ambiguity – Precedence

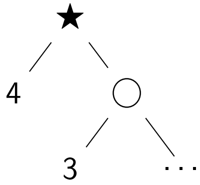
Input: $4 \star 3 \bigcirc \ldots$

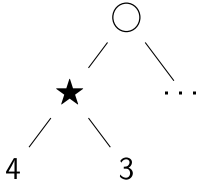




Ambiguity – Precedence

Input: 4 ★ 3 ◯ ...





- *prec*(◯) $>$ *prec*(★)
- Equal prec. and ★ is right-associative

- *prec*(◯) $<$ *prec*(★)
- Equal prec. and ★ is left-associative

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 `-` $<$ `[]`/`->`

```
def parsePrimaryExpr(...):
  # handle numbers, unary operators,
  # variables, parenthesized expr.
  ... # trivial ;)
def parseExpr(..., minPrec=0):
  lhs = parsePrimaryExpr(...)
  ... # (next slide)
```

Hand-written Parsing – Expression Parsing

- Only allow ops. with higher prec. on the right child
- Operator precedence
  - $* \to (3, \text{left-assoc})$
  - $+ \to (2, \text{left-assoc})$
  - $= \to (1, \text{right-assoc})$
- Right-assoc.: allow same prec.

```
def parsePrimaryExpr(...):
  # handle numbers, unary operators,
  # variables, parenthesized expr.
  ... # trivial ;)
def parseExpr(..., minPrec=0):
  lhs = parsePrimaryExpr(...)
  while True:
    tok = nextToken()
    prec, rassoc = OPERATORS[tok]
    if prec < minPrec:
      return lhs
    # XXX: handling for: (, [, ?:
    newPrec = prec if rassoc else prec+1
    rhs = parseExpr(..., newPrec)
    lhs = ("expr", tok, lhs, rhs)
```

Hand-written Parsing – Expression Parsing

- Only allow ops. with higher prec. on the right child
- Operator precedence
  - $* \to (3, \text{left-assoc})$
  - $+ \to (2, \text{left-assoc})$
  - $= \to (1, \text{right-assoc})$
- Right-assoc.: allow same prec.
  - Assignment, ternary

```
def parsePrimaryExpr(...):
  # handle numbers, unary operators,
  # variables, parenthesized expr.
  ... # trivial ;)
def parseExpr(..., minPrec=0):
  lhs = parsePrimaryExpr(...)
  while True:
    tok = nextToken()
    prec, rassoc = OPERATORS[tok]
    if prec < minPrec:
      return lhs
    # XXX: handling for: (, [, ?:
    newPrec = prec if rassoc else prec+1
    rhs = parseExpr(..., newPrec)
    lhs = ("expr", tok, lhs, rhs)
```

Hand-written Parsing – Expression Parsing

4+3*2*5+1

tok	minPrec	lhs
	0	4

```
OPERATORS = {
    "*": (3, False),
    "+": (2, False),
    "=": (1, True),
}
```

```
def parsePrimaryExpr(...):
  # handle numbers, unary operators,
  # variables, parenthesized expr.
  ... # trivial ;)
def parseExpr(..., minPrec=0):
  lhs = parsePrimaryExpr(...)
  while True:
    tok = nextToken()
    prec, rassoc = OPERATORS[tok]
    if prec < minPrec:
      return lhs
    # XXX: handling for: (, [, ?:
    newPrec = prec if rassoc else prec+1
    rhs = parseExpr(..., newPrec)
    lhs = ("expr", tok, lhs, rhs)
```

Hand-written Parsing – Expression Parsing

4+3*2*5+1

tok	minPrec	lhs
+	0	4

```
OPERATORS = {
    "*": (3, False),
    "+": (2, False),
    "=": (1, True),
}
```

```
def parsePrimaryExpr(...):
  # handle numbers, unary operators,
  # variables, parenthesized expr.
  ... # trivial ;)
def parseExpr(..., minPrec=0):
  lhs = parsePrimaryExpr(...)
  while True:
    tok = nextToken()
    prec, rassoc = OPERATORS[tok]
    if prec < minPrec:
      return lhs
    # XXX: handling for: (, [, ?:
    newPrec = prec if rassoc else prec+1
    rhs = parseExpr(..., newPrec)
    lhs = ("expr", tok, lhs, rhs)
```

Hand-written Parsing – Expression Parsing

4+3*2*5+1

tok	minPrec	lhs
+	0	4
	3	3

```
OPERATORS = {
    "*": (3, False),
    "+": (2, False),
    "=": (1, True),
}
```

```
def parsePrimaryExpr(...):
  # handle numbers, unary operators,
  # variables, parenthesized expr.
  ... # trivial ;)
def parseExpr(..., minPrec=0):
  lhs = parsePrimaryExpr(...)
  while True:
    tok = nextToken()
    prec, rassoc = OPERATORS[tok]
    if prec < minPrec:
      return lhs
    # XXX: handling for: (, [, ?:
    newPrec = prec if rassoc else prec+1
    rhs = parseExpr(..., newPrec)
    lhs = ("expr", tok, lhs, rhs)
```

Hand-written Parsing – Expression Parsing

4+3*2*5+1

tok	minPrec	lhs
+	0	4
*	3	3

```
OPERATORS = {
    "*": (3, False),
    "+": (2, False),
    "=": (1, True),
}
```

```
def parsePrimaryExpr(...):
  # handle numbers, unary operators,
  # variables, parenthesized expr.
  ... # trivial ;)
def parseExpr(..., minPrec=0):
  lhs = parsePrimaryExpr(...)
  while True:
    tok = nextToken()
    prec, rassoc = OPERATORS[tok]
    if prec < minPrec:
      return lhs
    # XXX: handling for: (, [, ?:
    newPrec = prec if rassoc else prec+1
    rhs = parseExpr(..., newPrec)
    lhs = ("expr", tok, lhs, rhs)
```

Hand-written Parsing – Expression Parsing

4+3*2*5+1

tok	minPrec	lhs
+	0	4
*	3	3
	4	2

```
OPERATORS = {
    "*": (3, False),
    "+": (2, False),
    "=": (1, True),
}
```

```
def parsePrimaryExpr(...):
  # handle numbers, unary operators,
  # variables, parenthesized expr.
  ... # trivial ;)
def parseExpr(..., minPrec=0):
  lhs = parsePrimaryExpr(...)
  while True:
    tok = nextToken()
    prec, rassoc = OPERATORS[tok]
    if prec < minPrec:
      return lhs
    # XXX: handling for: (, [, ?:
    newPrec = prec if rassoc else prec+1
    rhs = parseExpr(..., newPrec)
    lhs = ("expr", tok, lhs, rhs)
```

Hand-written Parsing – Expression Parsing

4+3*2*5+1

tok	minPrec	lhs
+	0	4
*	3	3*2

```
OPERATORS = {
    "*": (3, False),
    "+": (2, False),
    "=": (1, True),
}
```

```
def parsePrimaryExpr(...):
  # handle numbers, unary operators,
  # variables, parenthesized expr.
  ... # trivial ;)
def parseExpr(..., minPrec=0):
  lhs = parsePrimaryExpr(...)
  while True:
    tok = nextToken()
    prec, rassoc = OPERATORS[tok]
    if prec < minPrec:
      return lhs
    # XXX: handling for: (, [, ?:
    newPrec = prec if rassoc else prec+1
    rhs = parseExpr(..., newPrec)
    lhs = ("expr", tok, lhs, rhs)
```

Hand-written Parsing – Expression Parsing

4+3*2*5+1

tok	minPrec	lhs
+	0	4
*	3	3*2
	4	5

```
OPERATORS = {
    "*": (3, False),
    "+": (2, False),
    "=": (1, True),
}
```

```
def parsePrimaryExpr(...):
  # handle numbers, unary operators,
  # variables, parenthesized expr.
  ... # trivial ;)
def parseExpr(..., minPrec=0):
  lhs = parsePrimaryExpr(...)
  while True:
    tok = nextToken()
    prec, rassoc = OPERATORS[tok]
    if prec < minPrec:
      return lhs
    # XXX: handling for: (, [, ?:
    newPrec = prec if rassoc else prec+1
    rhs = parseExpr(..., newPrec)
    lhs = ("expr", tok, lhs, rhs)
```

Hand-written Parsing – Expression Parsing

4+3*2*5+1

tok	minPrec	lhs
+	0	4
*	3	(3*2)*5

```
OPERATORS = {
    "*": (3, False),
    "+": (2, False),
    "=": (1, True),
}
```

```
def parsePrimaryExpr(...):
  # handle numbers, unary operators,
  # variables, parenthesized expr.
  ... # trivial ;)
def parseExpr(..., minPrec=0):
  lhs = parsePrimaryExpr(...)
  while True:
    tok = nextToken()
    prec, rassoc = OPERATORS[tok]
    if prec < minPrec:
      return lhs
    # XXX: handling for: (, [, ?:
    newPrec = prec if rassoc else prec+1
    rhs = parseExpr(..., newPrec)
    lhs = ("expr", tok, lhs, rhs)
```

Hand-written Parsing – Expression Parsing

4+3*2*5+1

tok	minPrec	lhs
+	0	4+((3*2)*5)

```
OPERATORS = {
    "*": (3, False),
    "+": (2, False),
    "=": (1, True),
}
```

```
def parsePrimaryExpr(...):
  # handle numbers, unary operators,
  # variables, parenthesized expr.
  ... # trivial ;)
def parseExpr(..., minPrec=0):
  lhs = parsePrimaryExpr(...)
  while True:
    tok = nextToken()
    prec, rassoc = OPERATORS[tok]
    if prec < minPrec:
      return lhs
    # XXX: handling for: (, [, ?:
    newPrec = prec if rassoc else prec+1
    rhs = parseExpr(..., newPrec)
    lhs = ("expr", tok, lhs, rhs)
```

Hand-written Parsing – Expression Parsing

4+3*2*5+1

tok	minPrec	lhs
+	0	4+((3*2)*5)
	3	1

```
OPERATORS = {
    "*": (3, False),
    "+": (2, False),
    "=": (1, True),
}
```

```
def parsePrimaryExpr(...):
  # handle numbers, unary operators,
  # variables, parenthesized expr.
  ... # trivial ;)
def parseExpr(..., minPrec=0):
  lhs = parsePrimaryExpr(...)
  while True:
    tok = nextToken()
    prec, rassoc = OPERATORS[tok]
    if prec < minPrec:
      return lhs
    # XXX: handling for: (, [, ?:
    newPrec = prec if rassoc else prec+1
    rhs = parseExpr(..., newPrec)
    lhs = ("expr", tok, lhs, rhs)
```

Hand-written Parsing – Expression Parsing

4+3*2*5+1

tok	minPrec	lhs
+	0	(4+((3*2)*5))+1

```
OPERATORS = {
    "*": (3, False),
    "+": (2, False),
    "=": (1, True),
}
```

```
def parsePrimaryExpr(...):
  # handle numbers, unary operators,
  # variables, parenthesized expr.
  ... # trivial ;)
def parseExpr(..., minPrec=0):
  lhs = parsePrimaryExpr(...)
  while True:
    tok = nextToken()
    prec, rassoc = OPERATORS[tok]
    if prec < minPrec:
      return lhs
    # XXX: handling for: (, [, ?:
    newPrec = prec if rassoc else prec+1
    rhs = parseExpr(..., newPrec)
    lhs = ("expr", tok, lhs, rhs)
```

Top-down vs. Bottom-up Parsing

Top-down Parsing

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

Bottom-up Parsing

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

Parser Generators

- Writing parsers by hand can be large effort
- Parser generators can simplify parser writing a lot
  - Yacc/Bison, PLY, ANTLR, ...
- Automatic generation of parser/parsing tables from CFG
  - But: lexer often written by hand either way
- Used heavily in practice

Bison Example – part 1

```
%define api.pure full
%define api.value.type {ASTNode*}
%param { Lexer* lexer }
%code{
static int yylex(ASTNode** lvalp, Lexer* lexer);
}
%token NUMBER
%token WHILE "while"
%token BREAK "break"

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

Bison Example – part 2

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

Parsing in Practice

- Some use parser generators, e.g. Python
  some use hand-written parsers, e.g. GCC, Clang

Parsing in Practice

- Some use parser generators, e.g. Python
  some use hand-written parsers, e.g. GCC, Clang
- Optimization of grammar for performance
  - Rewrite rules to reduce states, etc.

Parsing in Practice

- Some use parser generators, e.g. Python
  some use hand-written parsers, e.g. GCC, Clang
- Optimization of grammar for performance
  - Rewrite rules to reduce states, etc.
- Useful error-handling: complex!
  - Try skipping to next separator, e.g. ; or ,

Parsing in Practice

- Some use parser generators, e.g. Python
  some use hand-written parsers, e.g. GCC, Clang
- 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

Parsing C++

Parsing C++

- C++ is not context-free (inherited from C): `T * a;`

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

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[2]

[2] TL Veldhuizen. *C++ templates are Turing complete*. 2003. 🌐.

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[2]
- C++ *parsing* is hence *undecidable*[3]
  - Template instantiation combined with C `T * a` ambiguity

[2] TL Veldhuizen. *C++ templates are Turing complete*. 2003. .

[3] J Haberman. *Parsing C++ is literally undecidable*. 2013. .

Semantic Analysis

- Syntactical correctness $\not\Rightarrow$ correct program

```
void foo = doesntExist / ++"abc";
```

Semantic Analysis

- Syntactical correctness $\not\Rightarrow$ correct program

  ```
  void foo = doesntExist / ++"abc";
  ```

- Needs context-sensitive analysis:
  - Variable existence, storage, accessibility, ...
  - Function existence, arguments, ...
  - Operator type compatibility
  - Attribute allowance
- Additional type complexity: inference, polymorphism, ...

Semantic Analysis: Scope Checking with AST Walking

- Idea: walk through AST (in DFS-order) and validate on the way
- Keep track of scope with declared variables
  - $Scope = (Map[Name \to Type]\ names, Scope\ parent)$
  - Might need to keep track of defined types separately
- For identifiers: check existence and get type
- For expressions: check types and derive result type
- For assignment: check lvalue-ness of left side

- *Might* be possible during AST creation
- Needs care with built-ins and other special constructs

Semantic Analysis and Post-Parsing Transformations

- Check for error-prone code patterns
  - Completeness of `switch`, out-of-range constants, unused variables, ...
- Check method calls, parameter types

- Duplicate code for templates
- Make implicit value conversions explicit
- Handle attributes: visibility, warnings, etc.

- Mangle names, split functions (OpenMP), ABI-specific setup, ...
- Last step: generate IR code

Parsing Performance

Is parsing/front-end performance important?

Parsing Performance

Is parsing/front-end performance important?

- Not necessarily: normal compilers
  - Some languages (e.g., Rust) need unbounded time *for parsing*

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

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

Data Types

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

Data Types: Implementing Classes

- Simple `class`/`struct`: trivial, just bunch of fields
  - Methods take (pointer to) `this` as implicit parameter
- Single inheritance: also trivial – extend struct at end

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

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: embed all parents, multiple vtables

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: embed all parents, multiple vtables
- Dynamic casts: needs run-time type information (RTTI)

Recommended Lectures

AD IN2227 "Compiler Constructions" covers parsing/analysis in depth

AD CIT3230000 "Programming Languages" covers dispatching/mixins/...

Interpretation on the AST

Interpretation on the AST

1. Find entry point
2. Walk through AST
   - Compute values for expressions
   - Track values of variables in scope
3. Profit!

Compiler Front-end – Summary

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

Compiler Front-end – Questions

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

Code Generation for Data Processing

Lecture 3: Intermediate Representations

Alexis Engelke

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

Winter 2022/23

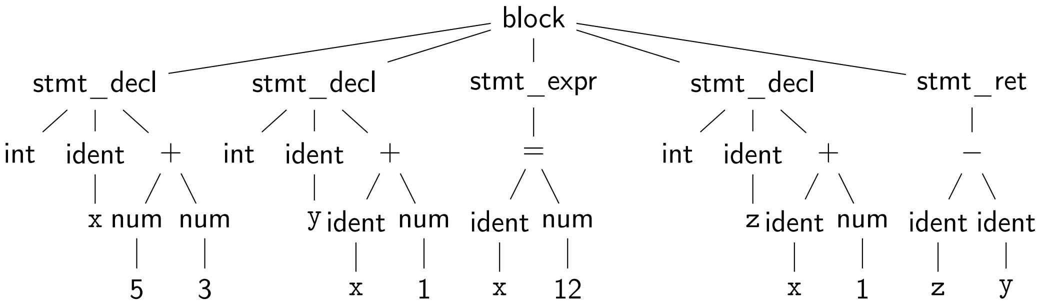
Intermediate Representations: Motivation

- So far: program parsed into AST

+ Great for language-related checks
+ Easy to correlate with original source code (e.g., errors)

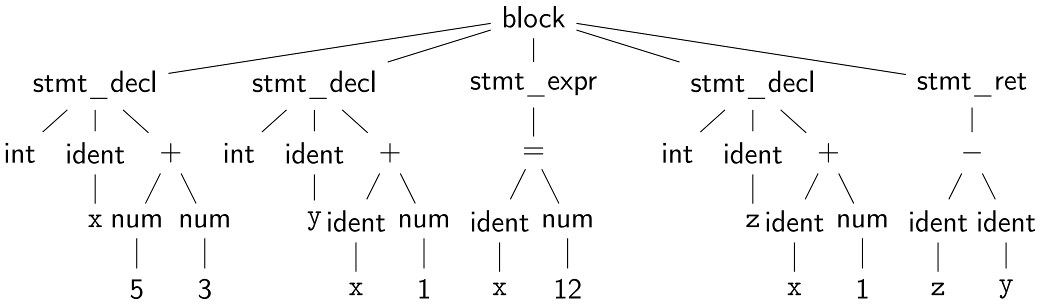
– Hard for analyses/optimizations due to high complexity
  - variable names, control flow constructs, etc.
  - Data and control flow implicit
– Highly language-specific

Intermediate Representations: Motivation



Question: how to optimize? Is `x+1` redundant?

Intermediate Representations: Motivation



Question: how to optimize? Is x+1 redundant? ⇝ hard to tell ☹

Intermediate Representations: Motivation

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

Question: how to optimize? Is `x+1` redundant?

Intermediate Representations: Motivation

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

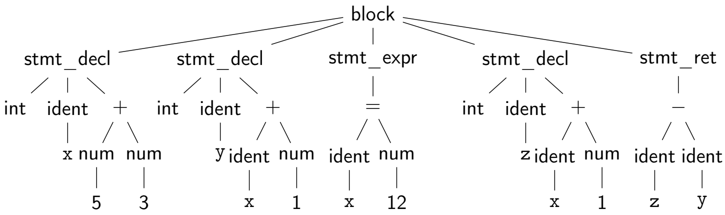
Question: how to optimize? Is `x+1` redundant? $\rightsquigarrow$ No! ☺

Intermediate Representations

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

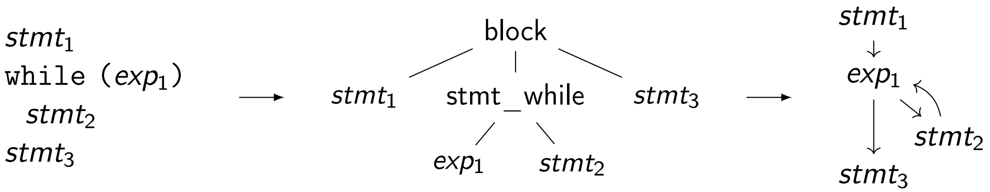
Graph IRs: Abstract Syntax Tree (AST)

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



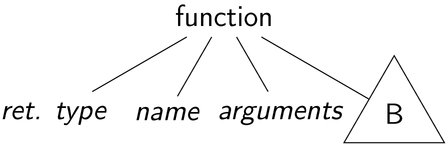
Graph IRs: Control Flow Graph (CFG)

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



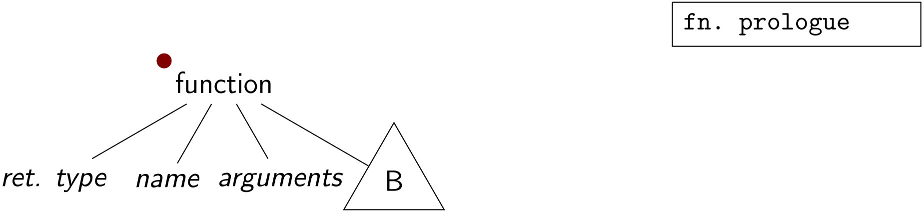
Build CFG from AST – Function

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



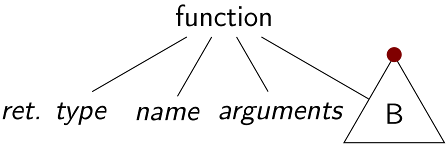
Build CFG from AST – Function

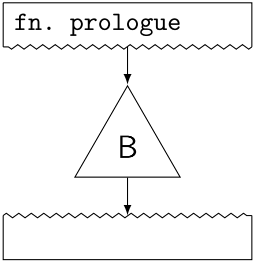
- Idea: Keep track of current insert block while walking through AST



Build CFG from AST – Function

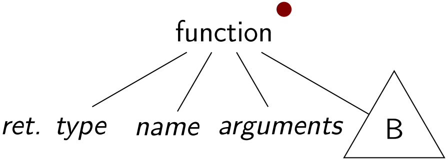
- Idea: Keep track of current insert block while walking through AST

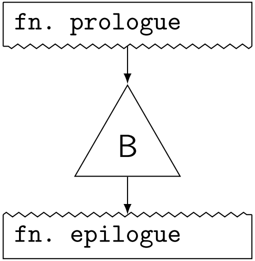




Build CFG from AST – Function

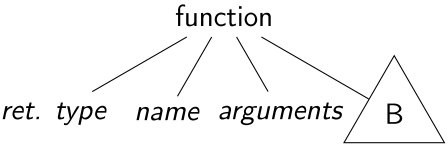
- Idea: Keep track of current insert block while walking through AST

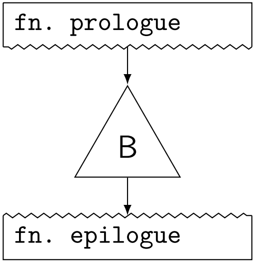




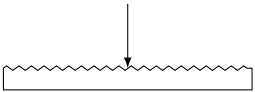
Build CFG from AST – Function

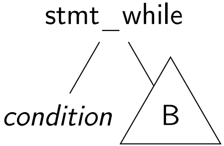
- Idea: Keep track of current insert block while walking through AST



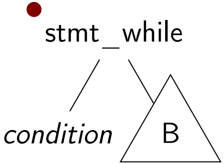


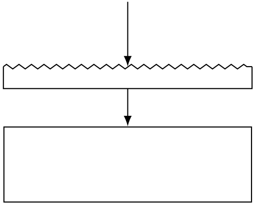
Build CFG from AST – While Loop



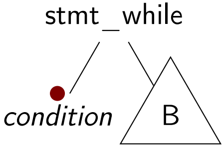


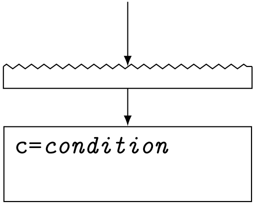
Build CFG from AST – While Loop



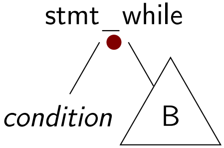


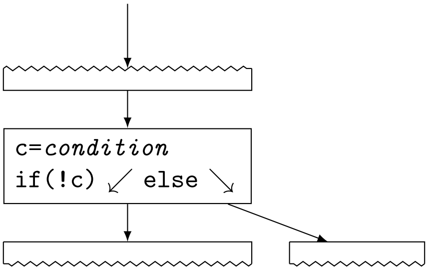
Build CFG from AST – While Loop



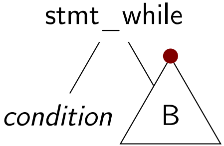


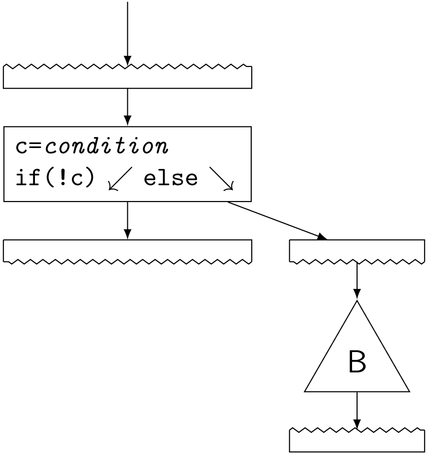
Build CFG from AST – While Loop



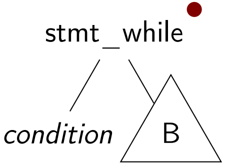


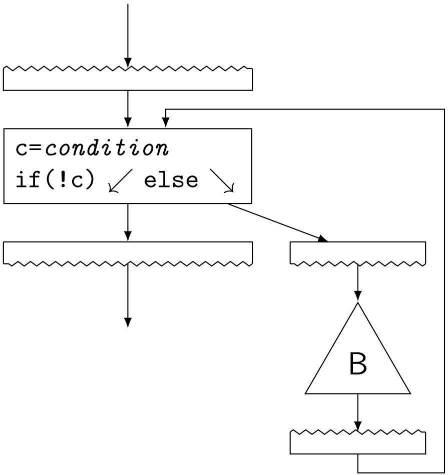
Build CFG from AST – While Loop



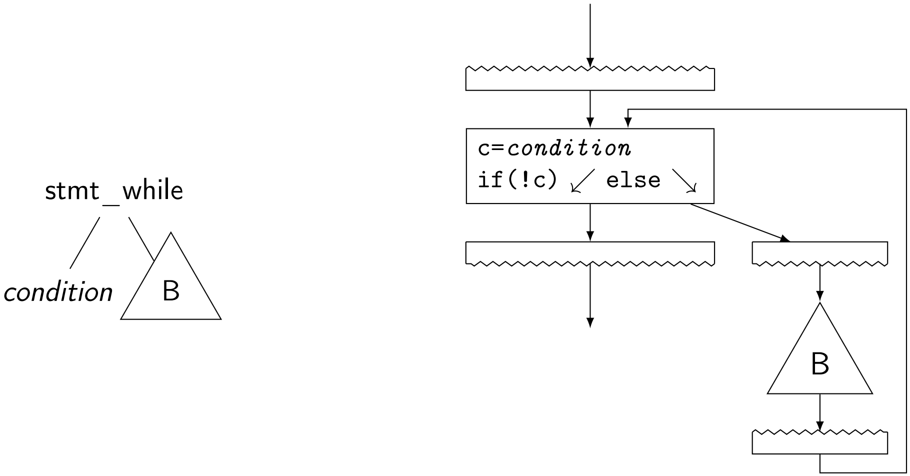


Build CFG from AST – While Loop

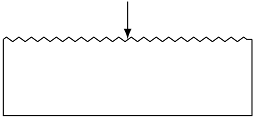


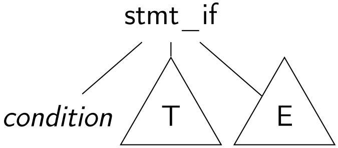


Build CFG from AST – While Loop

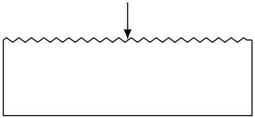


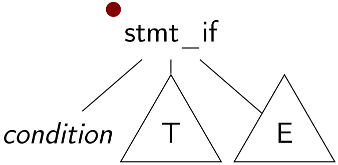
Build CFG from AST – If Condition



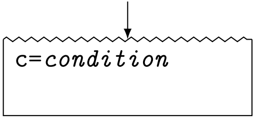


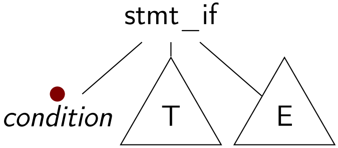
Build CFG from AST – If Condition



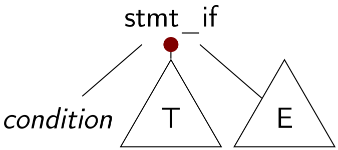


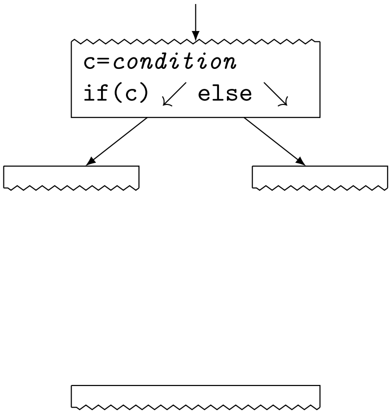
Build CFG from AST – If Condition



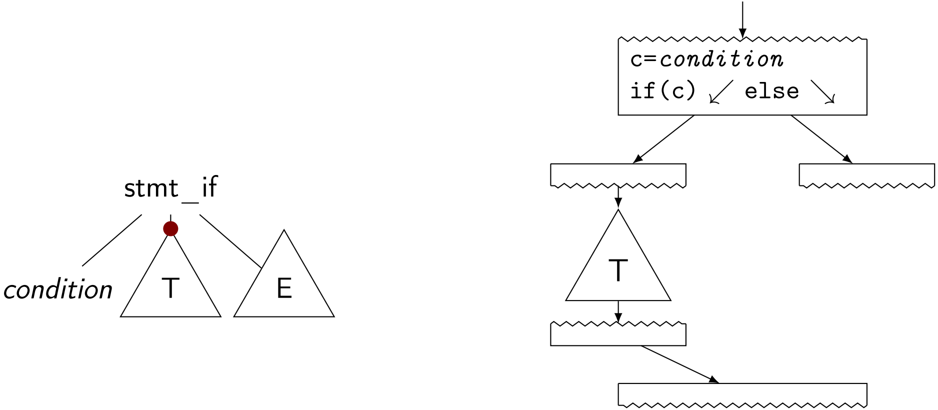


Build CFG from AST – If Condition

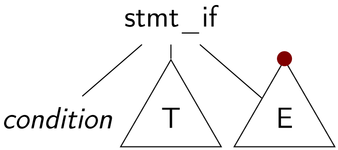


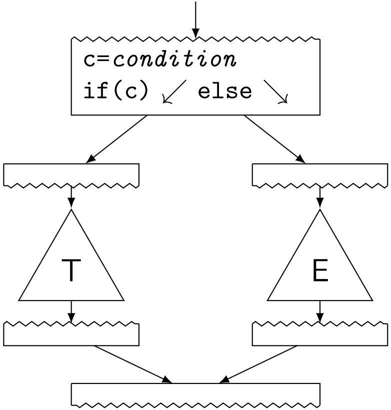


Build CFG from AST – If Condition

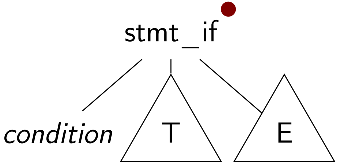


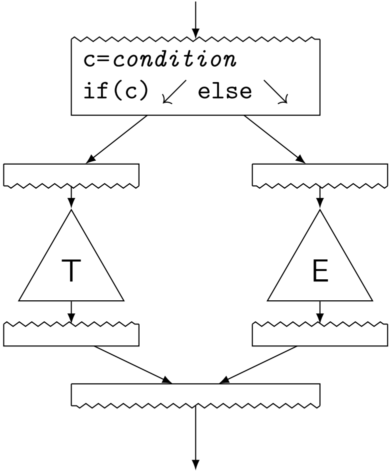
Build CFG from AST – If Condition



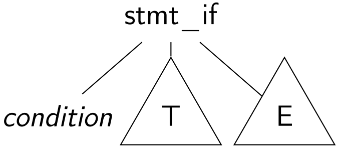


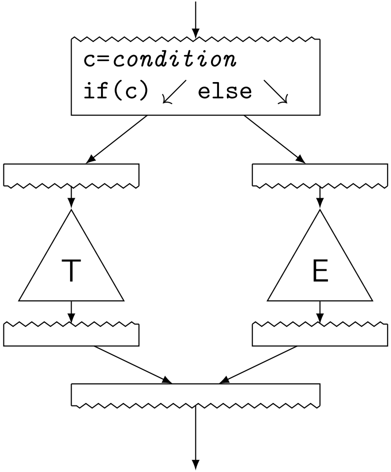
Build CFG from AST – If Condition





Build CFG from AST – If Condition





Build CFG from AST: Switch

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

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

Jump table

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

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

Build CFG from AST: Switch

Linear search

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

+ Trivial
– Slow, lot of code

Binary search

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

+ Good: sparse values
– Even more code

Jump table

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

table = {
  B_D, B_D, B_D, B₃,
  B₄, B_D, ... }
```

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

Build CFG from AST: Break, Continue, Goto

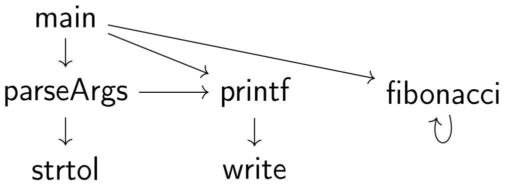
- `break`/`continue`: trivial
  - Keep track of target block, insert branch
- `goto`: also trivial
  - Split block at target label, if needed
  - But: may lead to irreducible control flow graph

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

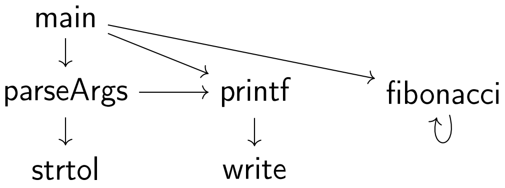
Graph IRs: Call Graph

- Graph showing (possible) call relations between functions



Graph IRs: Call Graph

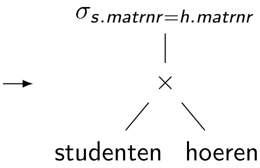
- Graph showing (possible) call relations between functions
- Useful for interprocedural optimizations
  - Function ordering
  - Stack depth estimation
  - ...



Graph IRs: Relational Algebra

- Higher-level representation of query plans
  - Explicit data flow

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

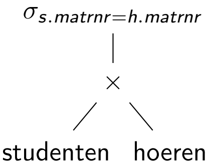


Graph IRs: Relational Algebra

- 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.

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

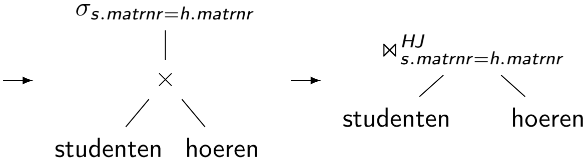
→



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.

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



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

\+

−

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

Linear IRs: Stack Machines

- Operands stored on a stack
- Operations pop arguments from top and push result
- Typically accompanied with variable storage

- Generating IR from AST: trivial
- Often used for bytecode, e.g. Java, Python

+ Compact code, easy to generate and implement
− Performance, hard to analyze

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

Linear IRs: Register Machines

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

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

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
```

Example: Low GIMPLE

```c
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
```

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
```

Linear IRs: Register Machines

- Problem: no clear def–use information
  - Is $x + 1$ the same?
  - Hard to track actual values!

- **How to optimize?**

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

Linear IRs: Register Machines

- Problem: no clear def–use information
  - Is $x + 1$ the same?
  - Hard to track actual values!
- **How to optimize?**

$\Rightarrow$ Disallow mutations of variables

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

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

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

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

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

Single Static Assignment: Control Flow

- How to handle diverging values in control flow?

$$
\begin{aligned}
\mathit{entry}: \quad & x \leftarrow \ldots \\
& \text{if } (x > 2) \text{ goto } \mathit{cont} \qquad \longrightarrow \\
\mathit{then}: \quad & x \leftarrow x * 2 \\
\mathit{cont}: \quad & \text{return } x
\end{aligned}
$$

Single Static Assignment: Control Flow

- How to handle diverging values in control flow?

$$
\begin{array}{rl}
\textit{entry}: & x \leftarrow \ldots \\
 & \text{if } (x > 2) \text{ goto } \textit{cont} \\
\textit{then}: & x \leftarrow x * 2 \\
\textit{cont}: & \text{return } x
\end{array}
\quad \rightarrow \quad
\begin{array}{rl}
\textit{entry}: & v_1 \leftarrow \ldots \\
 & \text{if } (v_1 > 2) \text{ goto } \textit{cont} \\
\textit{then}: & v_2 \leftarrow v_1 * 2 \\
\textit{cont}: & \text{return } ???
\end{array}
$$

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)

$$
\begin{aligned}
\textit{entry}: \quad & x \leftarrow \ldots \\
& \text{if } (x > 2) \text{ goto } \textit{cont} \\
\textit{then}: \quad & x \leftarrow x * 2 \\
\textit{cont}: \quad & \text{return } x
\end{aligned}
\quad \rightarrow \quad
\begin{aligned}
\textit{entry}: \quad & v_1 \leftarrow \ldots \\
& \text{if } (v_1 > 2) \text{ goto } \textit{cont} \\
\textit{then}: \quad & v_2 \leftarrow v_1 * 2 \\
\textit{cont}: \quad & v_3 \leftarrow \Phi(\textit{entry}: v_1, \textit{then}: v_2) \\
& \text{return } v_3
\end{aligned}
$$

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
```

SSA Construction – Local Value Numbering

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

Code | **SSA IR** | **Variable Mapping**

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

SSA Construction – Local Value Numbering

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

Code

$\rightarrow$	x	$\leftarrow$	5	$+$	3
	y	$\leftarrow$	x	$+$	1
	x	$\leftarrow$	12		
	z	$\leftarrow$	x	$+$	1
	tmp_1	$\leftarrow$	z	$-$	y
	return		tmp_1		

SSA IR

$v_1 \leftarrow$ add 5, 3

Variable Mapping

$x \rightarrow v_1$

SSA Construction – Local Value Numbering

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

Code

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

SSA IR

$$\begin{array}{lll} v_1 & \leftarrow & \texttt{add } 5,\ 3 \\ v_2 & \leftarrow & \texttt{add } v_1,\ 1 \end{array}$$

Variable Mapping

$$\begin{array}{lll} x & \rightarrow & v_1 \\ y & \rightarrow & v_2 \end{array}$$

SSA Construction – Local Value Numbering

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

Code

	x	$\leftarrow$	5	$+$	3
	y	$\leftarrow$	x	$+$	1
$\rightarrow$	x	$\leftarrow$	12		
	z	$\leftarrow$	x	$+$	1
	tmp_1	$\leftarrow$	z	$-$	y
	return		tmp_1		

SSA IR

v_1	$\leftarrow$	`add 5, 3`
v_2	$\leftarrow$	`add` v_1`, 1`
v_3	$\leftarrow$	`const 12`

Variable Mapping

x	$\rightarrow$	v_3	!
y	$\rightarrow$	v_2	

SSA Construction – Local Value Numbering

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

Code						SSA IR			Variable Mapping		
	x	$\leftarrow$	5	$+$	3	v_1	$\leftarrow$	`add 5, 3`	x	$\rightarrow$	v_3
	y	$\leftarrow$	x	$+$	1	v_2	$\leftarrow$	`add` v_1, 1	y	$\rightarrow$	v_2
	x	$\leftarrow$	12			v_3	$\leftarrow$	`const 12`	z	$\rightarrow$	v_4
$\rightarrow$	z	$\leftarrow$	x	$+$	1	v_4	$\leftarrow$	`add` v_3, 1			
	tmp_1	$\leftarrow$	z	$-$	y						
	return		tmp_1								

SSA Construction – Local Value Numbering

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

Code

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

SSA IR

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

Variable Mapping

$$
\begin{array}{rcl}
x & \rightarrow & v_3 \\
y & \rightarrow & v_2 \\
z & \rightarrow & v_4 \\
tmp_1 & \rightarrow & v_5
\end{array}
$$

SSA Construction – Local Value Numbering

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

Code

	x	$\leftarrow$	5	$+$	3
	y	$\leftarrow$	x	$+$	1
	x	$\leftarrow$	12		
	z	$\leftarrow$	x	$+$	1
	tmp_1	$\leftarrow$	z	$-$	y
$\rightarrow$	return		tmp_1		

SSA IR

v_1	$\leftarrow$	`add` 5, 3
v_2	$\leftarrow$	`add` v_1, 1
v_3	$\leftarrow$	`const` 12
v_4	$\leftarrow$	`add` v_3, 1
v_5	$\leftarrow$	`sub` v_4, v_2
		`ret` v_5

Variable Mapping

x	$\rightarrow$	v_3
y	$\rightarrow$	v_2
z	$\rightarrow$	v_4
tmp_1	$\rightarrow$	v_5

SSA Construction – Local Value Numbering

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

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

SSA Construction – Across Blocks

- SSA construction with control flow is non-trivial
- Key problem: find value for variable in predecessor

- Naive approach: 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

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*

SSA Construction – Across Blocks (“simple”[4])

- 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

[4] M Braun et al. “Simple and efficient construction of static single assignment form”. In: *CC*. 2013, pp. 102–122. 🌐.

SSA Construction – Example

func *foo* (v_1)

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

SSA Construction – Example

```
        func foo (v1)
entry:  sealed; varmap: n→ v1
```

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

SSA Construction – Example

func foo (v_1)

entry: sealed; varmap: n$\rightarrow v_1$, res$\rightarrow v_2$

$v_2 \leftarrow 1$

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

SSA Construction – Example

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

func $foo\,(v_1)$

entry: sealed; varmap: n→ v_1, res→ v_2

$v_2 \leftarrow 1$

header: NOT sealed; varmap: $\emptyset$

body: NOT sealed; varmap: $\emptyset$

cont: NOT sealed; varmap: $\emptyset$

SSA Construction – Example

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

func $foo\,(v_1)$

entry: sealed; varmap: n $\to v_1$, res $\to v_2$

$v_2 \leftarrow 1$

header: NOT sealed; varmap: $\emptyset$

$v_3 \leftarrow$ equal ???, 0

body: NOT sealed; varmap: $\emptyset$

cont: NOT sealed; varmap: $\emptyset$

SSA Construction – Example

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

func $foo\,(v_1)$

entry: sealed; varmap: n→ v_1, res→ v_2

$v_2 \leftarrow 1$

header: NOT sealed; varmap: n→ ϕ_1

$\phi_1 \leftarrow \phi$ *incomplete, for* n

$v_3 \leftarrow$ equal ϕ_1, 0

body: NOT sealed; varmap: $\emptyset$

cont: NOT sealed; varmap: $\emptyset$

SSA Construction – Example

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

func foo (v_1)

entry: sealed; varmap: n→ v_1, res→ v_2

$v_2 \leftarrow 1$

header: NOT sealed; varmap: n→ ϕ_1

$\phi_1 \leftarrow \phi$ *incomplete, for* n

$v_3 \leftarrow$ equal ϕ_1, 0

br v_3, cont, body

body: NOT sealed; varmap: $\emptyset$

cont: NOT sealed; varmap: $\emptyset$

SSA Construction – Example

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

func $foo(v_1)$

entry:
sealed; varmap: n→ v_1, res→ v_2
$v_2 \leftarrow 1$

header:
NOT sealed; varmap: n→ ϕ_1
$\phi_1 \leftarrow \phi$ *incomplete, for* n
$v_3 \leftarrow$ equal ϕ_1, 0
br v_3, cont, body

body:
sealed; varmap: $\emptyset$

cont:
NOT sealed; varmap: $\emptyset$

SSA Construction – Example

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

func *foo* (v_1)

entry: sealed; varmap: n→ v_1, res→ v_2

$v_2 \leftarrow 1$

header: NOT sealed; varmap: n→ ϕ_1

$\phi_1 \leftarrow \phi$ *incomplete, for* n

$v_3 \leftarrow$ equal ϕ_1, 0

br v_3, cont, body

body: sealed; varmap: $\emptyset$

$v_4 \leftarrow$ mul ???, ???

cont: NOT sealed; varmap: $\emptyset$

SSA Construction – Example

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

func *foo* (v_1)

entry: sealed; varmap: n→ v_1, res→ v_2

$v_2 \leftarrow 1$

header: NOT sealed; varmap: n→ ϕ_1

$\phi_1 \leftarrow \phi$ *incomplete, for* n

$v_3 \leftarrow$ equal ϕ_1, 0

br v_3, cont, body

body: sealed; varmap: n→ϕ_1

$v_4 \leftarrow$ mul ϕ_1, ϕ_1

cont: NOT sealed; varmap: $\emptyset$

SSA Construction – Example

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

func $foo\,(v_1)$

entry: sealed; varmap: n$\rightarrow v_1$, res$\rightarrow v_2$

$v_2 \leftarrow 1$

header: NOT sealed; varmap: n$\rightarrow \phi_1$

$\phi_1 \leftarrow \phi$ *incomplete, for* n

$v_3 \leftarrow$ equal ϕ_1, 0

br v_3, cont, body

body: sealed; varmap: n$\rightarrow \phi_1$

$v_4 \leftarrow$ mul ϕ_1, ϕ_1

$v_5 \leftarrow$ mul ???, v_4

cont: NOT sealed; varmap: $\emptyset$

SSA Construction – Example

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

func $foo\,(v_1)$

entry: sealed; varmap: n$\rightarrow v_1$, res$\rightarrow v_2$

$v_2 \leftarrow 1$

header: NOT sealed; varmap: n$\rightarrow \phi_1$, res$\rightarrow \phi_2$

$\phi_1 \leftarrow \phi$ *incomplete, for* n

$\phi_2 \leftarrow \phi$ *incomplete, for* res

$v_3 \leftarrow$ equal ϕ_1, 0

br v_3, cont, body

body: sealed; varmap: n$\rightarrow\phi_1$, res$\rightarrow v_5$

$v_4 \leftarrow$ mul ϕ_1, ϕ_1

$v_5 \leftarrow$ mul ϕ_2, v_4

cont: NOT sealed; varmap: $\emptyset$

SSA Construction – Example

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

func foo (v_1)

entry: sealed; varmap: n$\to v_1$, res$\to v_2$

$v_2 \leftarrow 1$

header: NOT sealed; varmap: n$\to \phi_1$, res$\to \phi_2$

$\phi_1 \leftarrow \phi$ *incomplete, for* n

$\phi_2 \leftarrow \phi$ *incomplete, for* res

$v_3 \leftarrow$ equal ϕ_1, 0

br v_3, cont, body

body: sealed; varmap: n$\to v_6$, res$\to v_5$

$v_4 \leftarrow$ mul ϕ_1, ϕ_1

$v_5 \leftarrow$ mul ϕ_2, v_4

$v_6 \leftarrow$ sub ϕ_1, 1

cont: NOT sealed; varmap: $\emptyset$

SSA Construction – Example

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

func $foo\,(v_1)$

entry: sealed; varmap: n→ v_1, res→ v_2

$v_2 \leftarrow 1$

header: NOT sealed; varmap: n→ ϕ_1, res→ ϕ_2

$\phi_1 \leftarrow \phi$ *incomplete, for* n

$\phi_2 \leftarrow \phi$ *incomplete, for* res

$v_3 \leftarrow$ equal ϕ_1, 0

br v_3, cont, body

body: sealed; varmap: n→v_6, res→ v_5

$v_4 \leftarrow$ mul ϕ_1, ϕ_1

$v_5 \leftarrow$ mul ϕ_2, v_4

$v_6 \leftarrow$ sub ϕ_1, 1

br header

cont: NOT sealed; varmap: $\emptyset$

SSA Construction – Example

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

func *foo* (v_1)

entry:
- sealed; varmap: n$\rightarrow v_1$, res$\rightarrow v_2$
- $v_2 \leftarrow 1$

header:
- sealed; varmap: n$\rightarrow \phi_1$, res$\rightarrow \phi_2$
- $\phi_1 \leftarrow \phi$ *incomplete, for* n
- $\phi_2 \leftarrow \phi$ *incomplete, for* res
- $v_3 \leftarrow$ equal ϕ_1, 0
- br v_3, cont, body

body:
- sealed; varmap: n$\rightarrow v_6$, res$\rightarrow v_5$
- $v_4 \leftarrow$ mul ϕ_1, ϕ_1
- $v_5 \leftarrow$ mul ϕ_2, v_4
- $v_6 \leftarrow$ sub ϕ_1, 1
- br header

cont:
- NOT sealed; varmap: $\emptyset$

SSA Construction – Example

```
int foo(int n) {
  int res = 1;
  while (n) {
    res *= n * n;
    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→ ϕ_1, res→ ϕ_2
$\phi_1 \leftarrow \phi(\text{entry: } v_1, \text{body: } v_6)$
$\phi_2 \leftarrow \phi(\text{entry: } v_2, \text{body: } v_5)$
$v_3 \leftarrow$ equal ϕ_1, 0
br v_3, cont, body

body:
sealed; varmap: n→v_6, res→ v_5
$v_4 \leftarrow$ mul ϕ_1, ϕ_1
$v_5 \leftarrow$ mul ϕ_2, v_4
$v_6 \leftarrow$ sub ϕ_1, 1
br header

cont:
NOT sealed; varmap: $\emptyset$

SSA Construction – Example

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

func foo (v_1)

entry: sealed; varmap: n$\rightarrow v_1$, res$\rightarrow v_2$

$v_2 \leftarrow 1$

header: sealed; varmap: n$\rightarrow \phi_1$, res$\rightarrow \phi_2$

$\phi_1 \leftarrow \phi$(entry: v_1, body: v_6)

$\phi_2 \leftarrow \phi$(entry: v_2, body: v_5)

$v_3 \leftarrow$ equal ϕ_1, 0

br v_3, cont, body

body: sealed; varmap: n$\rightarrow v_6$, res$\rightarrow v_5$

$v_4 \leftarrow$ mul ϕ_1, ϕ_1

$v_5 \leftarrow$ mul ϕ_2, v_4

$v_6 \leftarrow$ sub ϕ_1, 1

br header

cont: sealed; varmap: $\emptyset$

SSA Construction – Example

```
int foo(int n) {
  int res = 1;
  while (n) {
    res *= n * n;
    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→ ϕ_1, res→ ϕ_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 ϕ_1, 0

br v_3, cont, body

body: sealed; varmap: n→v_6, res→ v_5

$v_4 \leftarrow$ mul ϕ_1, ϕ_1

$v_5 \leftarrow$ mul ϕ_2, v_4

$v_6 \leftarrow$ sub ϕ_1, 1

br header

cont: sealed; varmap: res→ ϕ_2

ret ϕ_2

SSA Construction – Pruned/Minimal Form

- Resulting SSA is *pruned* – all ϕ are used
- But not *minimal* – ϕ nodes might have single, unique value

[5] M Braun et al. "Simple and efficient construction of static single assignment form". In: *CC*. 2013, pp. 102–122. .

[6] R Cytron et al. "Efficiently computing static single assignment form and the control dependence graph". In: *TOPLAS* 13.4 (1991), pp. 451–490. .

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?

[5] M Braun et al. "Simple and efficient construction of static single assignment form". In: *CC*. 2013, pp. 102–122.

[6] R Cytron et al. "Efficiently computing static single assignment form and the control dependence graph". In: *TOPLAS* 13.4 (1991), pp. 451–490.

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[5] or different construction method[6]

AD	IN2053 "Program Optimization" covers this more formally

[5] M Braun et al. "Simple and efficient construction of static single assignment form". In: *CC*. 2013, pp. 102–122.

[6] R Cytron et al. "Efficiently computing static single assignment form and the control dependence graph". In: *TOPLAS* 13.4 (1991), pp. 451–490.

SSA: Implementation

- Value is often just a pointer to instruction
- ϕ nodes placed at beginning of block
  - They execute "concurrently" and on the edges, after all

SSA: Implementation

- Value is often just a pointer to instruction
- ϕ nodes placed at beginning of block
  - They execute "concurrently" and on the edges, after all

- Variable number of operands required for ϕ nodes
- Storage format for instructions and basic blocks
  - Consecutive in memory: hard to modify/traverse
  - Array of pointers: $\mathcal{O}(n)$ for a single insertion...
  - Linked List: easy to insert, but pointer overhead

Is SSA a graph IR?

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

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

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?

Code Generation for Data Processing

Lecture 4: LLVM and IR Design

Alexis Engelke

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

Winter 2022/23

LLVM[7]

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, ...

[7] C Lattner and V Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *CGO*. 2004, pp. 75–86. 🌐.

LLVM[7]

LLVM "Core" Library

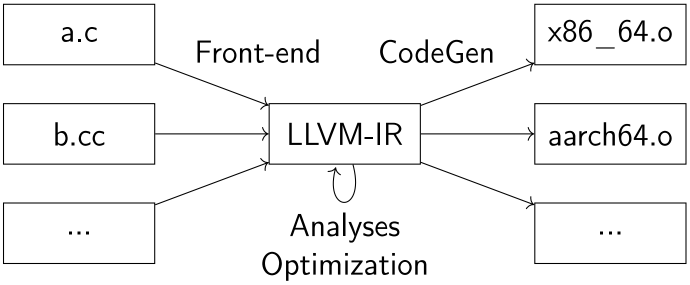
- Optimizer and compiler back-end
- "Set of compiler components"
  - IRs: LLVM-IR, SelDag, MIR
  - Analyses and Optimizations
  - Code generation back-ends
- Started from Chris Lattner's master's thesis
- Used for C, C++, Swift, D, Julia, Rust, Haskell, ...

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

[7] C Lattner and V Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *CGO*. 2004, pp. 75–86.

LLVM: Overview



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

LLVM-IR: Overview

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

```
define dso_local i32 @foo(i32 %0) {
  %2 = icmp eq i32 %0, 0
  br i1 %2, label %10, label %3

3: ; preds = %1, %3
  %4 = phi i32 [ %7, %3 ], [ 1, %1 ]
  %5 = phi i32 [ %8, %3 ], [ %0, %1 ]
  %6 = mul nsw i32 %5, %5
  %7 = mul nsw i32 %6, %4
  %8 = add nsw i32 %5, -1
  %9 = icmp eq i32 %8, 0
  br i1 %9, label %10, label %3

10: ; preds = %3, %1
  %11 = phi i32 [ 1, %1 ], [ %7, %3 ]
  ret i32 %11
}
```

LLVM-IR: Data types

- First class types:
  - `i<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`

LLVM-IR: Modules

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

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

- Function declarations/definitions

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

- Global named metadata (discarded during compilation)

LLVM-IR: Functions

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

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

LLVM-IR: Instructions – Control Flow and Terminators

- Terminators end a block/modify control flow

- `ret <ty> <val>`/`ret void`
- `br label <dest>`/`br i1 <cond>, label <then>, label <else>`
- `switch`/`indirectbr`
- `unreachable`
- Few others for exception handling

- Not a terminator: `call`

LLVM-IR: Instructions – Arithmetic-Logical

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

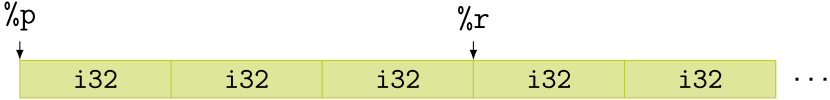
LLVM-IR: Instructions – Memory and Pointer

- alloca <ty> – allocate addressable stack slot
- load <ty>, ptr <ptr>/store <ty> <val>, ptr <ptr>
  - May be volatile (e.g., MMIO) and/or atomic
- cmpxchg/atomicrmw – similar to hardware operations
- ptrtoint/inttoptr
  - Changes provenance! inttoptr(ptrtoint(%x)) is not equal to %x
- getelementptr – address computation on ptr/structs/arrays

LLVM-IR: getelementptr Examples

LLVM-IR: getelementptr Examples

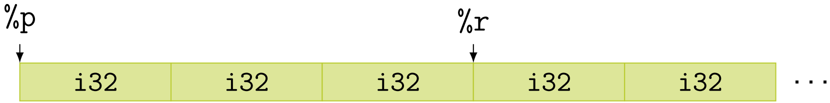
- `%r = getelementptr i32, ptr %p, i64 3`



Equivalent in C:

LLVM-IR: getelementptr Examples

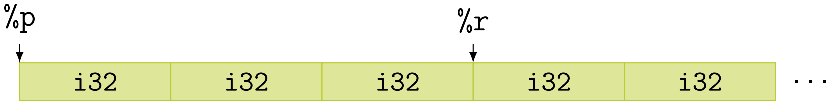
- `%r = getelementptr i32, ptr %p, i64 3`



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

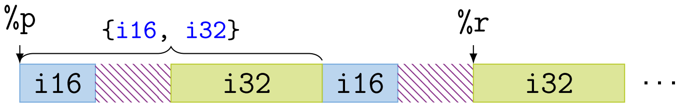
LLVM-IR: getelementptr Examples

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



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

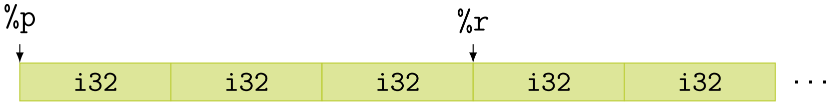
- `%r = getelementptr {i16, i32}, ptr %p, i64 1, i32 1`



Equivalent in C:

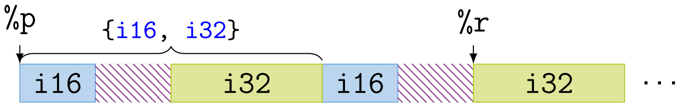
LLVM-IR: getelementptr Examples

- `%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

LLVM-IR: `undef` and `poison`

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` →

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` →

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` →

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*

LLVM-IR: `undef` and `poison`

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

LLVM-IR: Intrinsics

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

LLVM-IR: Tools

LLVM-IR: Tools

- `clang` can emit LLVM-IR bitcode
  `clang -O -emit-llvm -c test.c -o test.bc`

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`

LLVM-IR: Tools

- `clang` can emit LLVM-IR bitcode
  `clang -O -emit-llvm -c test.c -o test.bc`
- `llvm-dis` disassembles bitcode to textual LLVM-IR
  `clang -O -emit-llvm -c test.c -o - | llvm-dis`
- `llc` compiles LLVM-IR (textual or bitcode) to assembly
  `clang -O -emit-llvm -c test.c -o - | llc`
  `clang -O -emit-llvm -c test.c -o - | llvm-dis | llc`

Example Listings omitted – they would span several slides

LLVM-IR: Example

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

LLVM-IR: Example

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

LLVM-IR: Example

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

LLVM-IR: Example

```
@switch.table.sw = 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 @sw(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 @switch.table.sw, i64 0, i64 %10
  %12 = load i32, ptr %11, align 4
  br label %13
13: ; preds = %1, %4, %9
  %14 = phi i32 [ %12, %9 ], [ %0, %4 ], [ %0, %1 ]
  ret i32 %14
}
```

LLVM-IR API

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

LLVM-IR C++ API: Basic Example

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

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

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

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

- 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
        - ...
      - ...

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[8]
- Preferably uses `ArrayRef`, `StringRef`, `Twine` for references
- LLVM implements custom streams instead of std streams
  - `outs()`, `errs()`, `dbgs()`

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

LLVM-IR API: Use Tracking

- Values track their users

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

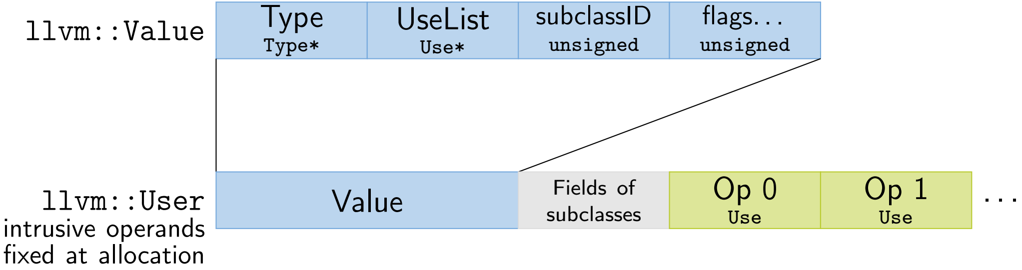
- Simplifies implementation of analyses
- Allows for easy replacement:
  - `inst->replaceAllUsesWith(replVal);`

LLVM IR Implementation: Value/User

`llvm::Value`	Type	UseList	subclassID	flags...
	`Type*`	`Use*`	`unsigned`	`unsigned`

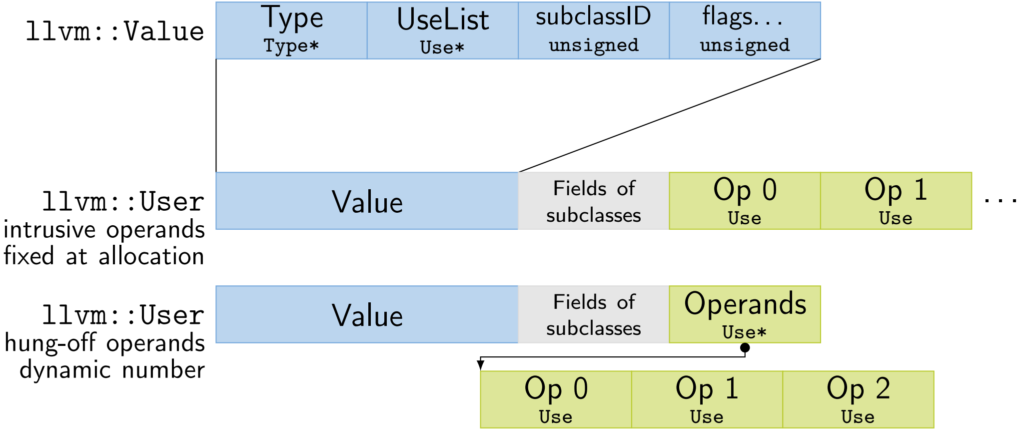
`PHINode` additionally stores n `BasicBlock*` after the operands, but aren't users of blocks.

LLVM IR Implementation: Value/User



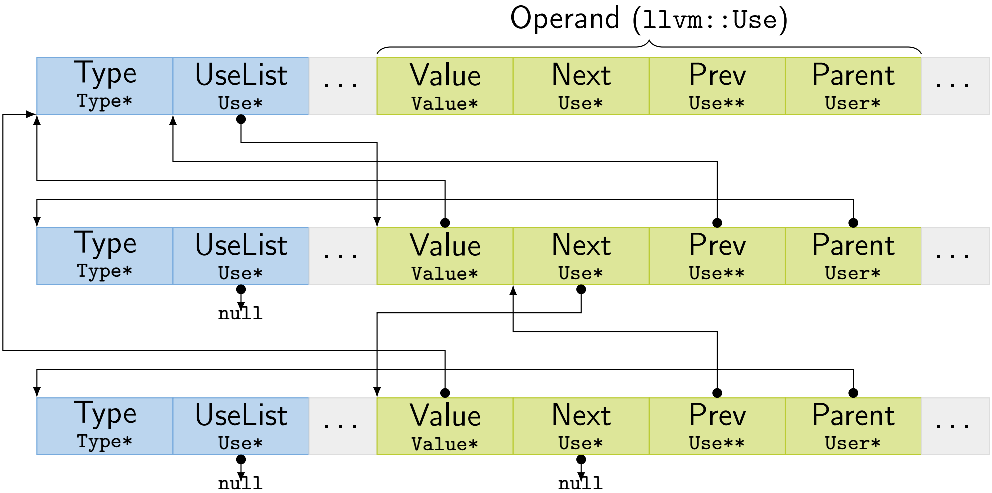
PHINode additionally stores *n* BasicBlock* after the operands, but aren't users of blocks.

LLVM IR Implementation: Value/User



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

LLVM IR Implementation: Use



LLVM IR Implementation: Instructions/Blocks

- `Instruction` and `BasicBlock` have pointers to parent and next/prev
  - Linked list updated on changes and used for iteration
  - Instructions have cached *order* (integer) for fast "comes before"

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

LLVM IR Implementation: Instructions/Blocks

- `Instruction` and `BasicBlock` have pointers to parent and next/prev
  - Linked list updated on changes and used for iteration
  - Instructions have cached *order* (integer) for fast "comes before"

- `BasicBlock` successors: blocks used by terminator
- `BasicBlock` predecessors:
  - Iterate over users of block – these are terminators (and `blockaddress`)
  - Ignore non-terminators, parent of using terminator is predecessor
  - Same predecessor might be duplicated ($\rightsquigarrow$ `getUniquePredecessor()`)

LLVM IR Implementation: Instructions/Blocks

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

LLVM and IR Design

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

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
- Thus: heavy trend towards custom IRs

IR Design: High-level Considerations

IR Design: High-level Considerations

- Define purpose!

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

IR Design: High-level Considerations

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

IR Design: Operations

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?

IR Design: Operations

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

IR Design: Operations

IR Design: Operations

- Allow instruction side effects?
  - E.g.: memory, floating-point arithmetic, implicit control flow

IR Design: Operations

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

IR Design: 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?

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)
```

LLVM and IR Design – Summary

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

- IR design depends on purpose and integration constraints
- Structurally similar IRs can strongly differ in capabilities

LLVM and IR Design – Questions

- What is the structure of an LLVM-IR module/function?
- Which LLVM-IR data types exist?
  How do they relate to the target architecture?
- How do semantically invalid operations in LLVM-IR behave?
- What is special about intrinsic functions?
- How to derive LLVM-IR from C code using Clang?
- How does LLVM's `replaceAllUsesWith` work?
  How could this work without building/maintaining user lists?
- How can an SSA-based IR make side effects explicit?
- How would you design an IR for optimizing Brainfuck?

Code Generation for Data Processing

Lecture 5: Analyses and Transformations

Alexis Engelke

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

Winter 2022/23

Program Transformation: Motivation

- "User code" is often not very efficient

[9] FE Allen and J Cocke. *A catalogue of optimizing transformations*. 1971. 🌐.

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

[9] FE Allen and J Cocke. *A catalogue of optimizing transformations*. 1971. 🌐.

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*

[9] FE Allen and J Cocke. *A catalogue of optimizing transformations.* 1971. 🌐.

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

[9] FE Allen and J Cocke. *A catalogue of optimizing transformations.* 1971. 🌐.

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[9]

[9] FE Allen and J Cocke. *A catalogue of optimizing transformations*. 1971. .

Dead Block Elimination

- CFG not necessarily connected
- E.g., consequence of optimization
  - Conditional branch $\rightarrow$ unconditional branch

Dead Block Elimination

- CFG not necessarily connected
- E.g., consequence of optimization
  - Conditional branch $\rightarrow$ unconditional branch

- Removing dead blocks is trivial
  1. DFS traversal of CFG from entry, mark visited blocks
  2. Remove unmarked blocks

Optimization Example 1

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

Simple Dead Code Elimination (DCE)

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

1. Add all instructions to work queue
2. While work queue not empty:
   2.1 Check for deadness
   2.2 If dead, remove and add all operands to work queue

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

Applying Simple DCE

```
               define i32 @fac(i32 %0) {
eff.: cf        br label %for.header
               for.header: ; preds = %for.body, %1
users: 1        %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]
users: 1        %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
               }
```

Applying Simple DCE

```
        define i32 @fac(i32 %0) {
eff.: cf    br label %for.header
        for.header: ; preds = %for.body, %1
users: 1    %a = phi i32 [ 1, %1 ], [ %a.new, %for.body ]
users: 1    %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

eff.: cf    ret i32 %a
        }
```

Dead Code Elimination

- Problem: unused value cycles

Dead Code Elimination

- Problem: unused value cycles
- Idea: find "value sinks" and mark all needed values as live
  unmarked values can be removed
  - Sink: instruction with side effects (e.g., store, control flow)

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

Applying Liveness-based DCE

Work list (stack)

```
define i32 @fac(i32 %0) {
 br1 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
 br2 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
 br3 label %for.header
exit: ; preds = %for.header
 %absum = add i32 %a, %b
 ret i32 %a
}
```

Applying Liveness-based DCE

```
      define i32 @fac(i32 %0) {
live   br1 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
live   br2 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
live   br3 label %for.header
      exit: ; preds = %for.header
       %absum = add i32 %a, %b
live   ret i32 %a
      }
```

Work list (stack)

br_1

br_2

br_3

ret

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 ]
      %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
      %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
      %cond = icmp sle i32 %i, %0
live  br₂ 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
live  br₃ label %for.header
     exit: ; preds = %for.header
      %absum = add i32 %a, %b
live  ret i32 %a
     }
```

Work list (stack)

br_1

br_2

br_3

%a

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 ]
       %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
       %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
       %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
       %b.new = add i32 %b, %i
       %i.new = add i32 %i, 1
live   br₃ label %for.header
      exit: ; preds = %for.header
       %absum = add i32 %a, %b
live   ret i32 %a
      }
```

Work list (stack)

br_1

br_2

br_3

%a.new

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 ]
       %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
live   %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
       %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
       %b.new = add i32 %b, %i
       %i.new = add i32 %i, 1
live   br₃ label %for.header
      exit: ; preds = %for.header
       %absum = add i32 %a, %b
live   ret i32 %a
      }
```

Work list (stack)

br_1

br_2

br_3

%i

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 ]
        %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
live    %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
        %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
        %b.new = add i32 %b, %i
live    %i.new = add i32 %i, 1
live    br₃ label %for.header
      exit: ; preds = %for.header
        %absum = add i32 %a, %b
live    ret i32 %a
      }
```

Work list (stack)

br_1

br_2

br_3

%i.new

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 ]
      %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
live  %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
      %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
      %b.new = add i32 %b, %i
live  %i.new = add i32 %i, 1
live  br3 label %for.header
     exit: ; preds = %for.header
      %absum = add i32 %a, %b
live  ret i32 %a
     }
```

Work list (stack)

br_1

br_2

br_3

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 ]
        %b = phi i32 [ 0, %1 ], [ %b.new, %for.body ]
live    %i = phi i32 [ 0, %1 ], [ %i.new, %for.body ]
        %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
        %b.new = add i32 %b, %i
live    %i.new = add i32 %i, 1
live    br₃ label %for.header
       exit: ; preds = %for.header
        %absum = add i32 %a, %b
live    ret i32 %a
       }
```

Work list (stack)

br_1

br_2

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 ]
         %b = phi i32 [ 0, %1 ], [ %b.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
         %b.new = add i32 %b, %i
live     %i.new = add i32 %i, 1
live     br₃ label %for.header
        exit: ; preds = %for.header
         %absum = add i32 %a, %b
live     ret i32 %a
        }
```

Work list (stack)

br_1

%cond

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 ]
       %b = phi i32 [ 0, %1 ], [ %b.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
       %b.new = add i32 %b, %i
live   %i.new = add i32 %i, 1
live   br₃ label %for.header
      exit: ; preds = %for.header
       %absum = add i32 %a, %b
live   ret i32 %a
      }
```

Work list (stack)

br_1

Applying Liveness-based DCE

Work list (stack)

```
      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 ]
       %b = phi i32 [ 0, %1 ], [ %b.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
       %b.new = add i32 %b, %i
live   %i.new = add i32 %i, 1
live   br₃ label %for.header
      exit: ; preds = %for.header
       %absum = add i32 %a, %b
live   ret i32 %a
      }
```

Applying Liveness-based DCE

Work list (stack)

```
      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
      }
```

Optimization Example 2

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

Common Subexpression Elimination (CSE) – Attempt 1

- Idea: find/eliminate redundant computation of same value

Common Subexpression Elimination (CSE) – Attempt 1

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

CSE Attempt 1 – Example 1

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

CSE Attempt 1 – Example 1

→ ht

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

CSE Attempt 1 – Example 1

→ ht (%4)

→ ht (%5)

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

CSE Attempt 1 – Example 1

→ ht
→ ht
→ ht

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

CSE Attempt 1 – Example 1

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

CSE Attempt 1 – Example 1

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

CSE Attempt 1 – Example 1

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

CSE Attempt 1 – Example 1

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

CSE Attempt 1 – Example 1

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

CSE Attempt 1 – Example 1

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

CSE Attempt 1 – Example 1

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

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

CSE Attempt 1 – Example 2

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

CSE Attempt 1 – Example 2

→ ht

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

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
             %add1 = add i32 %a, %b
             br label %if.end
           if.end: ; preds = %if.then, %entry
             %condvar = phi i32 [ %add1, %if.then ], [ %a, %entry ]
             %add2 = add i32 %a, %b
             %res = add i32 %condvar, %add2
             ret i32 %res
           }
```

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
            br label %if.end
           if.end: ; preds = %if.then, %entry
            %condvar = phi i32 [ %add1, %if.then ], [ %a, %entry ]
            %add2 = add i32 %a, %b
            %res = add i32 %condvar, %add2
            ret i32 %res
           }
```

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
         %condvar = phi i32 [ %add1, %if.then ], [ %a, %entry ]
         %add2 = add i32 %a, %b
         %res = add i32 %condvar, %add2
         ret i32 %res
        }
```

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 ]
          %add2 = add i32 %a, %b
          %res = add i32 %condvar, %add2
          ret i32 %res
        }
```

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
                 %res = add i32 %condvar, %add2
                 ret i32 %res
                }
```

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
               %res = add i32 %condvar, %add1
               ret i32 %res
              }
```

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, %add1
               ret i32 %res
             }
```

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, %add1
→ ht            ret i32 %res
              }
```

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, %add1
→ ht         ret i32 %res
            }
```

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

World Domination

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: $d \text{ sdom } n \Leftrightarrow d \text{ dom } n \wedge d \neq n$
- Immediate dominator:
  $\text{idom } (n) = d : d \text{ sdom } n \wedge \nexists d'.d \text{ sdom } d' \wedge d' \text{ sdom } n$

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 $n \wedge d \neq n$
- Immediate dominator:
  idom $(n) = d : d$ sdom $n \wedge \nexists d'.d$ sdom $d' \wedge d'$ sdom n

$\Rightarrow$ All strict dominators are always executed before the block

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: $d \text{ sdom } n \Leftrightarrow d \text{ dom } n \wedge d \neq n$
- Immediate dominator:
  $\text{idom}\ (n) = d : d \text{ sdom } n \wedge \nexists d'.d \text{ sdom } d' \wedge d' \text{ sdom } n$

⇒ All strict dominators are always executed before the block

⇒ All values from dominators available/usable

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: $d \text{ sdom } n \Leftrightarrow d \text{ dom } n \wedge d \neq n$
- Immediate dominator:
  $\text{idom}(n) = d : d \text{ sdom } n \wedge \nexists d'.d \text{ sdom } d' \wedge d' \text{ sdom } n$

⇒ All strict dominators are always executed before the block

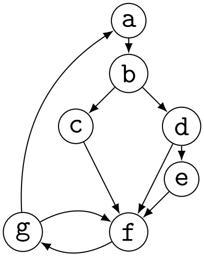
⇒ All values from dominators available/usable

⇒ All values not from dominators **not** usable

Dominator Tree

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

Control Flow Graph

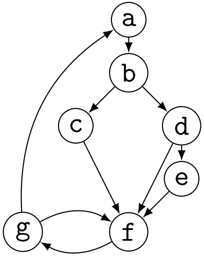


Dominator Tree

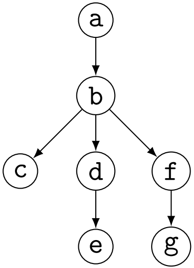
Dominator Tree

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

Control Flow Graph



Dominator Tree



Dominator Tree: Construction

- Naive: inefficient (but reasonably simple)[10]
  - 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

[10] ES Lowry and CW Medlock. "Object code optimization". In: *CACM* 12.1 (1969), pp. 13–22.

Dominator Tree: Construction

- Naive: inefficient (but reasonably simple)[10]
  - 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[11]
  - 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 frequently in compilers[12]

[10] ES Lowry and CW Medlock. "Object code optimization". In: *CACM* 12.1 (1969), pp. 13–22.

[11] T Lengauer and RE Tarjan. "A fast algorithm for finding dominators in a flowgraph". In: *TOPLAS* 1.1 (1979), pp. 121–141.

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

Dominator Tree: Implementation

- Per node store: *idom*, idom-children, DFS pre-order/post-order number

Dominator Tree: Implementation

- Per node store: *idom*, idom-children, DFS pre-order/post-order number
- Get immediate dominator:

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:

Dominator Tree: Implementation

- Per node store: *idom*, idom-children, DFS pre-order/post-order number
- Get immediate dominator: ...lookup *idom*
- Iterate over all dominators/dominated by: ...trivial
- Check whether *a* sdom *b*

Dominator Tree: Implementation

- Per node store: *idom*, idom-children, DFS pre-order/post-order number
- Get immediate dominator: ...lookup *idom*
- Iterate over all dominators/dominated by: ...trivial
- Check whether a sdom b[13]
  - $a.preNum < b.preNum \;\wedge\; a.postNum > b.postNum$
  - After updates, numbers might be invalid: recompute or walk tree

[13] PF Dietz. "Maintaining order in a linked list". In: *STOC*. 1982, pp. 122–127.

Dominator Tree: Implementation

- Per node store: *idom*, idom-children, DFS pre-order/post-order number

- Get immediate dominator: ...lookup *idom*
- Iterate over all dominators/dominated by: ...trivial
- Check whether a sdom b[13]
  - $a.preNum < b.preNum \ \wedge \ a.postNum > b.postNum$
  - After updates, numbers might be invalid: recompute or walk tree

- Problem: dominance of unreachable blocks ill-defined $\rightsquigarrow$ special handling

[13] PF Dietz. "Maintaining order in a linked list". In: *STOC*. 1982, pp. 122–127.

CSE Attempt 2

- Option 1:
  - For identical instructions, store all
  - Add dominance check before replacing
  - Visit nodes in reverse post-order (i.e., topological order)
- Option 2:[14]
  - Do a DFS over dominator tree
  - Use scoped hashmap to track available values

Does this work?

[14] P Briggs, KD Cooper, and LT Simpson. *Value numbering*. Tech. rep. CRPC-TR94517-S. Rice University, 1997.

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:[14]
  - Do a DFS over dominator tree
  - Use scoped hashmap to track available values

Does this work? Yes.

[14] P Briggs, KD Cooper, and LT Simpson. *Value numbering*. Tech. rep. CRPC-TR94517-S. Rice University, 1997.

CSE: Hashing an Instruction (and Beyond)

- Needs hash function *and* “relaxed” equality
- Idea: combine opcode and operands/constants into hash value
  - Use pointer or index for instruction result operands

CSE: Hashing an Instruction (and Beyond)

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

Global Value Numbering – or: advanced CSE

- Hash-based approach only catches trivially removable duplicates

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

Global Value Numbering – or: advanced CSE

- Hash-based approach only catches trivially removable duplicates

- Alternative: partition values into *congruence classes*
  - Congruent values are guaranteed to always have the same value
- Optimistic approach: values are congruent unless proven otherwise
- Pessimistic approach: values are not congruent unless proven

- Combinable with: reassociation, DCE, constant folding

- Rather complex, but can be highly beneficial[15]

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

Simple Transformations: Inlining

- Estimate whether inlining is beneficial
  - Savings of avoided call/computations/branches; cost of increased size

Simple Transformations: Inlining

- Estimate whether inlining is beneficial
  - Savings of avoided call/computations/branches; cost of increased size
- Copy original function in place of the call
  - Split basic block containing function call

Simple Transformations: Inlining

- Estimate whether inlining is beneficial
  - Savings of avoided call/computations/branches; cost of increased size

- Copy original function in place of the call
  - Split basic block containing function call
- Replace returns with branches and ϕ-node to/at continuation point

Simple Transformations: Inlining

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

Simple Transformations: Mem2Reg and SROA

- Mem2reg: promote `alloca` to SSA values/phis
  - Condition: only `load`/`store`, no address taken
  - Essentially just SSA construction
- SROA: scalar replacement of aggregate
  - Separate structure fields into separate variables
  - Also promote them to SSA

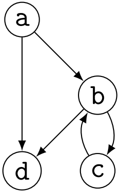
Loops

- Loop: maximal SCC L with at least one internal edge[16]
  (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 $L \setminus \{h\}$

[16] P Havlak. "Nesting of reducible and irreducible loops". In: *TOPLAS* 19.4 (1997), pp. 557–567. 🌐.

Loops

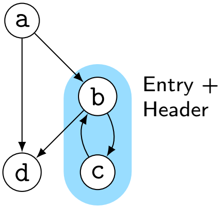
- Loop: maximal SCC L with at least one internal edge[16]
  (strongly connected component (SCC): all blocks reachable from each other)
  - Entry: block with an edge from outside of L
  - Header h: first entry found (might be ambiguous)
- Loop nested in L: loop in subgraph $L \setminus \{h\}$



[16] P Havlak. "Nesting of reducible and irreducible loops". In: *TOPLAS* 19.4 (1997), pp. 557–567.

Loops

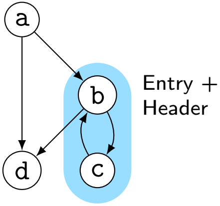
- Loop: maximal SCC L with at least one internal edge[16]
  (strongly connected component (SCC): all blocks reachable from each other)
  - Entry: block with an edge from outside of L
  - Header h: first entry found (might be ambiguous)
- Loop nested in L: loop in subgraph $L \setminus \{h\}$

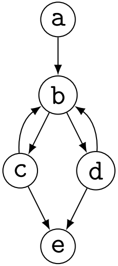


[16] P Havlak. "Nesting of reducible and irreducible loops". In: *TOPLAS* 19.4 (1997), pp. 557–567.

Loops

- Loop: maximal SCC L with at least one internal edge[16]
  (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 $L \setminus \{h\}$

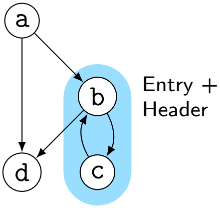


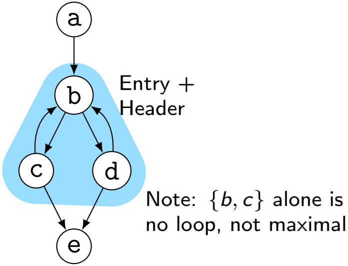


[16] P Havlak. "Nesting of reducible and irreducible loops". In: *TOPLAS* 19.4 (1997), pp. 557–567.

Loops

- Loop: maximal SCC L with at least one internal edge[16]
  (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 $L \setminus \{h\}$

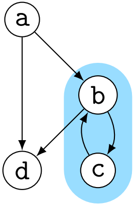


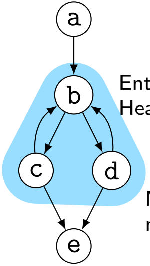


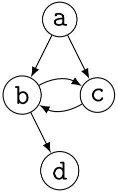
[16] P Havlak. "Nesting of reducible and irreducible loops". In: *TOPLAS* 19.4 (1997), pp. 557–567.

Loops

- Loop: maximal SCC L with at least one internal edge[16]
  (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 $L \setminus \{h\}$



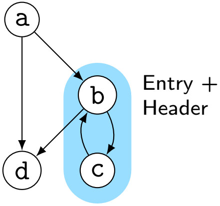


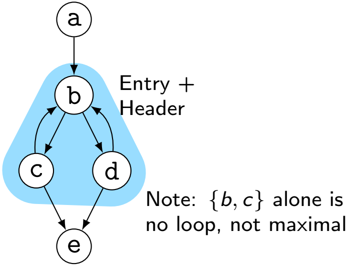


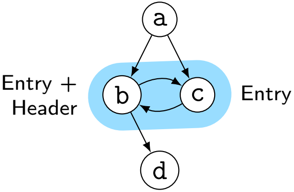
[16] P Havlak. "Nesting of reducible and irreducible loops". In: *TOPLAS* 19.4 (1997), pp. 557–567.

Loops

- Loop: maximal SCC L with at least one internal edge[16]
  (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 $L \setminus \{h\}$







[16] P Havlak. "Nesting of reducible and irreducible loops". In: *TOPLAS* 19.4 (1997), pp. 557–567.

Natural Loops

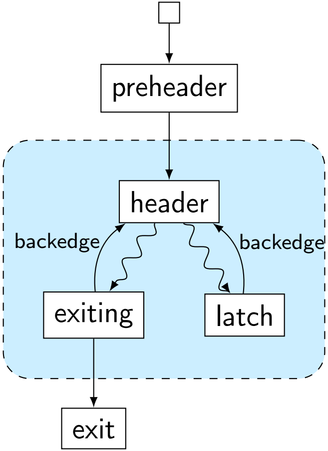
- Natural Loop: loop with single entry
  - ⇒ Header is unique
  - ⇒ Header dominates all block
  - ⇒ Loop is reducible

- Backedge: edge from block to header
- Predecessor: block with edge into loop
- Preheader: unique predecessor

Formal Definition

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

CFG is reducible iff all loops are reducible



Finding Natural Loops

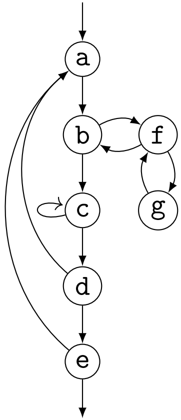
- Modified version[17] of Tarjan's algorithm[18]
- Iterate over dominator tree in post order
- Each block: find predecessors dominated by the block
  - None $\rightsquigarrow$ no loop header, continue
  - Any $\rightsquigarrow$ loop header, these edges *must* be backedges
- Walk through predecessors until reaching header again
  - All blocks on the way must be part of the loop body
  - Might encounter nested loops, update loop parent

[17] G Ramalingam. "Identifying loops in almost linear time". In: *TOPLAS* 21.2 (1999), pp. 175–188. 🌐.

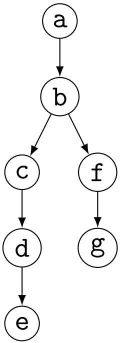
[18] R Tarjan. "Testing flow graph reducibility". In: *STOC*. 1973, pp. 96–107. 🌐.

Finding Natural Loops: Example

Control Flow Graph



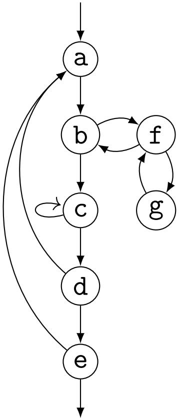
Dominator Tree



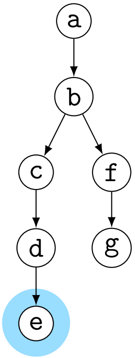
Loop Info

Finding Natural Loops: Example

Control Flow Graph



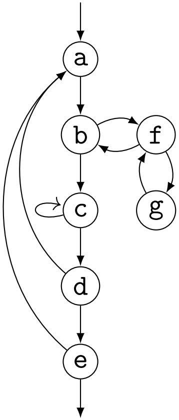
Dominator Tree



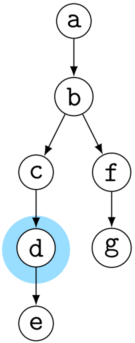
Loop Info

Finding Natural Loops: Example

Control Flow Graph



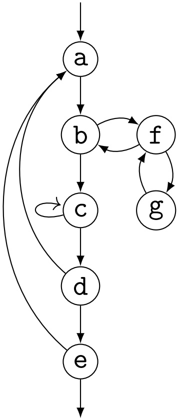
Dominator Tree



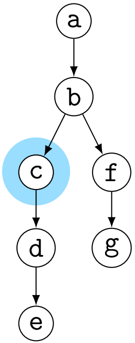
Loop Info

Finding Natural Loops: Example

Control Flow Graph



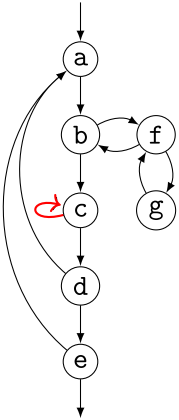
Dominator Tree



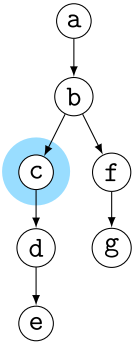
Loop Info

Finding Natural Loops: Example

Control Flow Graph



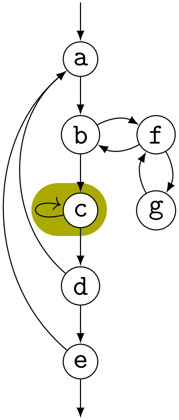
Dominator Tree



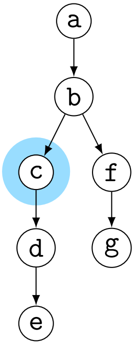
Loop Info

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



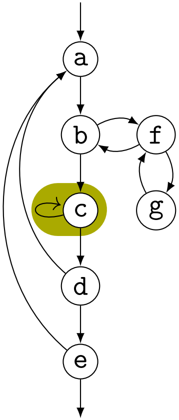
Loop Info

Loop A: {c}

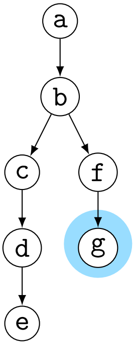
header: c; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



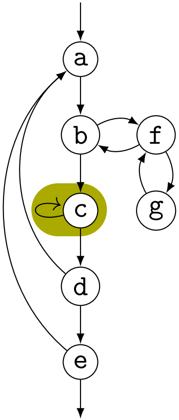
Loop Info

Loop A: {c}

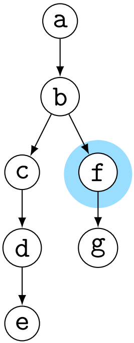
header: c; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



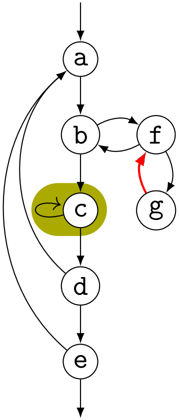
Loop Info

Loop A: {c}

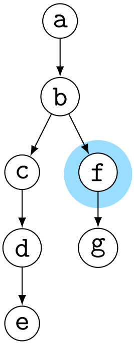
header: c; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



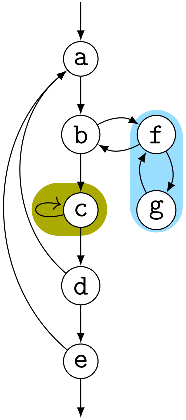
Loop Info

Loop A: {c}

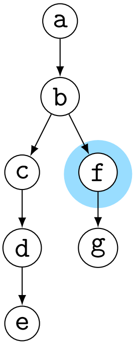
header: c; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



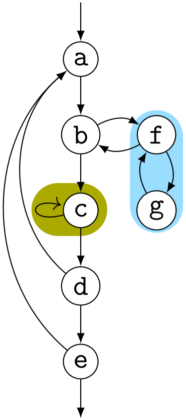
Loop Info

Loop A: {c}
header: c; parent: NULL

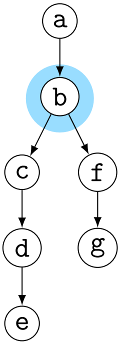
Loop B: {f,g}
header: f; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



Loop Info

Loop A: {c}

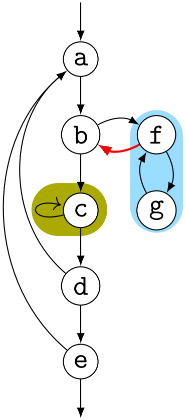
header: c; parent: NULL

Loop B: {f,g}

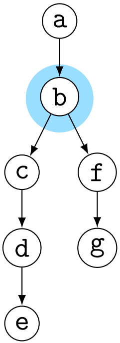
header: f; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



Loop Info

Loop A: {c}

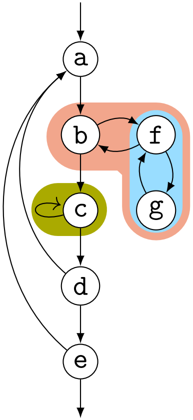
header: c; parent: NULL

Loop B: {f,g}

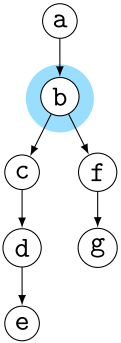
header: f; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



Loop Info

Loop A: `{c}`

header: `c`; parent: NULL

Loop B: `{f,g}`

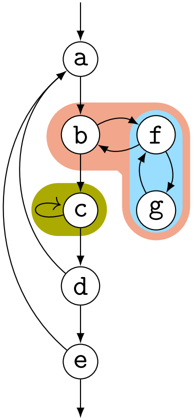
header: `f`; parent: C

Loop C: `{b,f,g}`

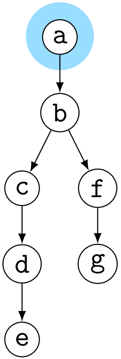
header: `b`; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



Loop Info

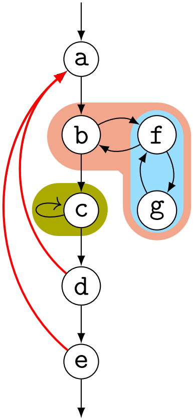
Loop A: `{c}`
header: `c`; parent: NULL

Loop B: `{f,g}`
header: `f`; parent: C

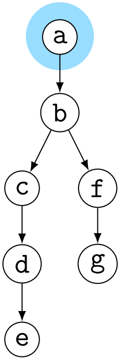
Loop C: `{b,f,g}`
header: `b`; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



Loop Info

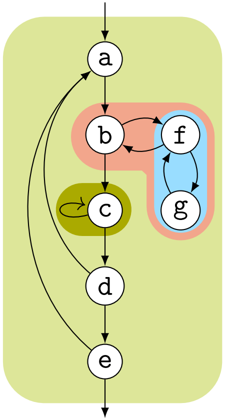
Loop A: `{c}`
header: `c`; parent: NULL

Loop B: `{f,g}`
header: `f`; parent: C

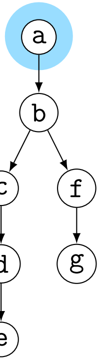
Loop C: `{b,f,g}`
header: `b`; parent: NULL

Finding Natural Loops: Example

Control Flow Graph



Dominator Tree



Loop Info

Loop A: {c}
header: c; parent: D

Loop B: {f,g}
header: f; parent: C

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

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

Loop Invariant Code Motion (LICM)

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

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

Loop Invariant Code Motion (LICM)

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

↑ Hoist:[19] iterate over blocks of loop in reverse post-order
  - For each movable inst., check for loop-defined operands
  - If not, move to preheader (create one, if not existent)
  - Otherwise, add inst. to set of values defined inside loop

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

Loop Invariant Code Motion (LICM)

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

↑ Hoist:[19] iterate over blocks of loop in reverse post-order
  - For each movable inst., check for loop-defined operands
  - If not, move to preheader (create one, if not existent)
  - Otherwise, add inst. to set of values defined inside loop

↓ Sink: Iterate over blocks of loop in post-order
  - For each movable inst., check for users inside loop
  - If none, move to unique exit (if existent)

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

Transformations and Analyses in LLVM: Passes

- Transformations and analyses organized in *passes*
- Pass can operate on Module/(CGSCC)/Function/Loop

Transformations and Analyses in LLVM: Passes

- Transformations and analyses organized in *passes*
- Pass can operate on Module/(CGSCC)/Function/Loop

- Analysis pass: takes input IR and returns analysis result
  - May also use results of other analyses; results are cached
- Transformation pass: takes input IR and returns preserved analyses
  - Can use analyses, which are re-run when outdated

Transformations and Analyses in LLVM: Passes

- Transformations and analyses organized in *passes*
- Pass can operate on Module/(CGSCC)/Function/Loop

- Analysis pass: takes input IR and returns analysis result
  - May also use results of other analyses; results are cached
- Transformation pass: takes input IR and returns preserved analyses
  - Can use analyses, which are re-run when outdated

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

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);
}
```

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

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

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

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

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

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

Analyses and Transformations – Summary

- Program Transformation critical for performance improvement
- Code not necessarily better
- Analyses are important to drive transformations
  - Dominator tree, loop detection, value liveness
- Important optimizations
  - Dead code elimination, common sub-expression elimination, loop-invariant code motion
- Compilers often implement transformations as passes
- Analyses may be invalidated by transformations, needs tracking

Analyses and Transformations – Questions

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

Code Generation for Data Processing

Lecture 6: Instruction Selection

Alexis Engelke

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

Winter 2022/23

Code Generation – Overview

Code Generation – Overview

- Instruction Selection
  - Map IR to assembly
  - Keep code shape and storage; change operations

Code Generation – Overview

- Instruction Selection
  - Map IR to assembly
  - Keep code shape and storage; change operations
- Instruction Scheduling
  - Optimize order to hide latencies
  - Keep operations, may increases demand for registers

Code Generation – Overview

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

Instruction Selection (ISel) – Overview

- Find machine instructions to implement abstract IR
- Typically separated from scheduling and register allocation

- Input: IR code with abstract instructions
- Output: lower-level IR code with target machine instructions

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

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

ISel – Typical Constraints

ISel – Typical Constraints

- Target offers multiple ways to implement operations
  - `imul x, 2`, `add x, x`, `shl x, 1`, `lea x, [x+x]`

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

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

ISel – Typical Constraints

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

Optimal ISel

- Find *most performant* instruction sequence with same semantics (?)
  - I.e., there no program with better "performance" exists
  - Performance = instructions associated with specific costs

[20] DR Koes and SC Goldstein. "Near-optimal instruction selection on DAGs". In: *CGO*. 2008, pp. 45–54.

Optimal ISel

- Find *most performant* instruction sequence with same semantics (?)
  - I.e., there no program with better "performance" exists
  - Performance = instructions associated with specific costs
- Problem: optimal code generation is **undecidable**

[20] DR Koes and SC Goldstein. "Near-optimal instruction selection on DAGs". In: *CGO*. 2008, pp. 45–54.

Optimal ISel

- Find *most performant* instruction sequence with same semantics (?)
  - I.e., there no program with better "performance" exists
  - Performance = instructions associated with specific costs
- Problem: optimal code generation is **undecidable**
- Alternative: optimal *tiling* of IR with machine code instrs
  - IR as dataflow graph, instr. tiles to optimally cover graph
  - $\mathcal{NP}$-complete[20]

[20] DR Koes and SC Goldstein. "Near-optimal instruction selection on DAGs". In: *CGO*. 2008, pp. 45–54.

Avoiding ISel Altogether

Avoiding ISel Altogether

Use an interpreter

+ Fast "compilation time", easy to implement
– Slow execution time

- Best if code is executed once

Macro Expansion

- Expand each IR operation with corresponding machine instrs

`%5 = add %1, 12345`	⟶	`%5a = movz 12345` `%5 = add %1, %5a`
`%6 = and %2, 7`	⟶	`%6 = and %2, 7`

`%7 = shl %5, %6`	⟶	`%7a = lsl %5, %6`
		`%7b = cmp %6, 64`
		`%7 = csel %7a, xzr, %7b, lo`

Macro Expansion

- Oldest approach, historically also does register allocation
  - Also possible by walking AST

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

Macro Expansion

- Oldest approach, historically also does register allocation
  - Also possible by walking AST

+ Very fast, linear time, simple to implement, easy to port
− Inefficient and large output code

- Used by, e.g., LLVM FastISel, Go, GCC

Peephole Optimization

- Plain macro expansion leads to suboptimal results
- Idea: replace inefficient instruction sequences[21]

- Originally: physical window over assembly code
  - Replace with more efficient instructions having same effects
  - Possibly with allocated registers

- Extension: do expansion before register allocation[22]
  - Expand IR into Register Transfer Lists (RTL) with temporary registers
  - While *combining*, ensure that each RTL can be implemented as single instr.

[21] WM McKeeman. "Peephole optimization". In: *CACM* 8.7 (1965), pp. 443–444. .

[22] JW Davidson and CW Fraser. "Code selection through object code optimization". In: *TOPLAS* 6.4 (1984), pp. 505–526. .

Peephole Optimization

- Originally covered only adjacent instructions
- Can also use logical window of data dependencies
  - Problem: instructions with multiple uses
  - Needs more sophisticated matching schemes for data deps.
    $\Rightarrow$ Tree-pattern matching

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.
    $\Rightarrow$ Tree-pattern matching

\+ Fast, also allows for target-specific sequences

− Pattern set grows large, limited potential

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.
    $\Rightarrow$ Tree-pattern matching

\+ Fast, also allows for target-specific sequences

– Pattern set grows large, limited potential

- Widely used today at different points during compilation

ISel as Graph Covering – High-level Intuition

- Idea: represent program as data flow graph

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)*

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

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

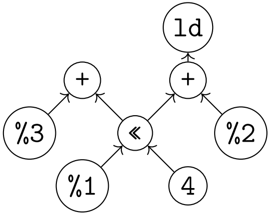
Tree Covering: Converting SSA into Trees

- SSA form:

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

  live-out: %6, %7

- Data flow graph:



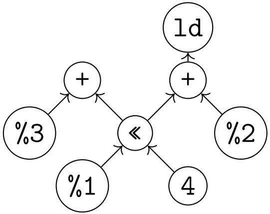
Tree Covering: Converting SSA into Trees

- SSA form:

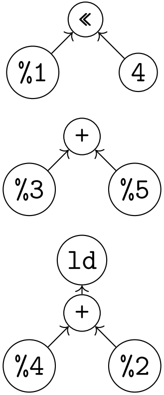
```
%4 = shl %1, 4
%5 = add %2, %4
%6 = add %3, %4
%7 = load %5
```

live-out: %6, %7

- Data flow graph:



- Method 1:
  Edge Splitting



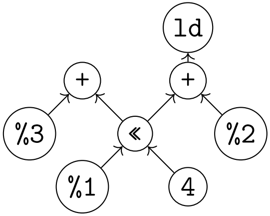
Tree Covering: Converting SSA into Trees

- SSA form:

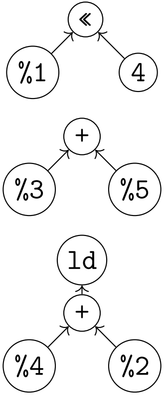
```
%4 = shl %1, 4
%5 = add %2, %4
%6 = add %3, %4
%7 = load %5
```

live-out: %6, %7

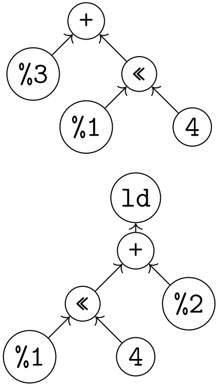
- Data flow graph:



- Method 1:
  Edge Splitting



- Method 2:
  Node Duplication



Tree Covering: Patterns

	Pattern	Cost	Instruction
P_0	$GP_{R1} \rightarrow$ «(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	$GP_{R1} \rightarrow +(GP_{R2},$ «$(GP_{R3}, K_1))$	2	`add` R_1, R_2, R_3, `lsl` #K_1
P_3	$GP_{R1} \rightarrow +($«$(GP_{R2}, K_1), GP_{R2})$	2	`add` R_1, R_3, R_2, `lsl` #K_1
P_4	$GP_{R1} \rightarrow$ `ld`(GP_{R2})	2	`ldr` R_1, [R_2]
P_5	$GP_{R1} \rightarrow$ `ld`$(+(GP_{R2}, GP_{R3}))$	2	`ldr` R_1, [R_2, R_3]
P_6	$GP_{R1} \rightarrow$ `ld`$(+(GP_{R2},$ «$(GP_{R3}, K_1)))$	3	`ldr` R_1, [R_2, R_3, `lsl` #K_1]
P_7	$GP_{R1} \rightarrow$ `ld`$(+($«$(GP_{R2}, K_1), GP_{R3}))$	3	`ldr` R_1, [R_3, R_2, `lsl` #K_1]
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
⋮	⋮	⋮	⋮

Tree Covering: Greedy/Maximal Munch

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

+ Easy to implement, fast

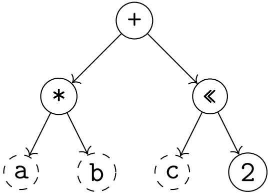
Tree Covering: Greedy/Maximal Munch

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

\+ Easy to implement, fast

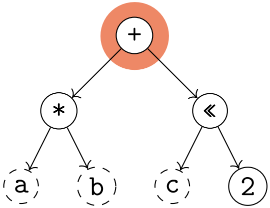
– Result might be non-optimum

Tree Covering: Greedy/Maximal Munch – Example



Matching Patterns:

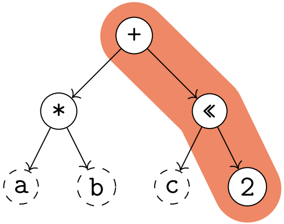
Tree Covering: Greedy/Maximal Munch – Example



Matching Patterns:

- +: P_1 – cost 1 – covered nodes: 1

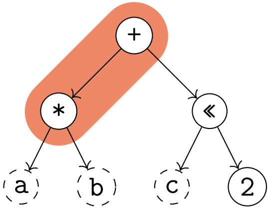
Tree Covering: Greedy/Maximal Munch – Example



Matching Patterns:

- +: P_1 – cost 1 – covered nodes: 1
- +: P_2 – cost 2 – covered nodes: 3

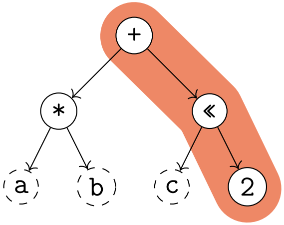
Tree Covering: Greedy/Maximal Munch – Example



Matching Patterns:

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

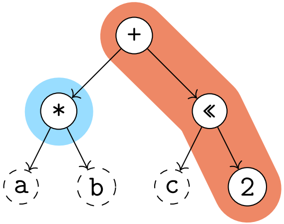
Tree Covering: Greedy/Maximal Munch – Example



Matching Patterns:

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

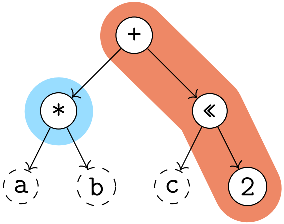
Tree Covering: Greedy/Maximal Munch – Example



Matching Patterns:

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

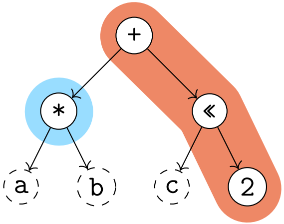
Tree Covering: Greedy/Maximal Munch – Example



Matching Patterns:

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

Tree Covering: Greedy/Maximal Munch – Example

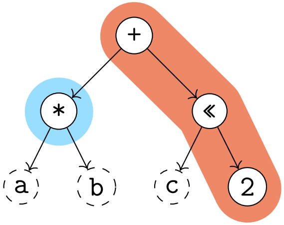


Matching Patterns:

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

Total cost: 5

Tree Covering: Greedy/Maximal Munch – Example



Matching Patterns:

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

Total cost: 5

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

Tree Covering: with LR-Parsing?

- Can we use (LR-)parsing for instruction selection?

[23] RS Glanville and SL Graham. "A new method for compiler code generation". In: *POPL*. 1978, pp. 231–254. .

Tree Covering: with LR-Parsing

- Can we use (LR-)parsing for instruction selection? Yes![23]
  - Pattern set = grammar; IR (in prefix notation) = input

Advantages

[23] RS Glanville and SL Graham. "A new method for compiler code generation". In: *POPL*. 1978, pp. 231–254.

Tree Covering: with LR-Parsing

- Can we use (LR-)parsing for instruction selection? Yes![23]
  - Pattern set = grammar; IR (in prefix notation) = input

Advantages

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

Disadvantages

[23] RS Glanville and SL Graham. "A new method for compiler code generation". In: *POPL*. 1978, pp. 231–254.

Tree Covering: with LR-Parsing

- Can we use (LR-)parsing for instruction selection? Yes![23]
  - 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

[23] RS Glanville and SL Graham. "A new method for compiler code generation". In: *POPL*. 1978, pp. 231–254.

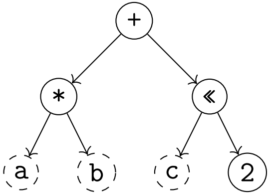
Tree Covering: Dynamic Programming[24]

- Step 1: compute cost matrix, bottom-up for all nodes
  - Matrix: tree node × non-terminal
    (different patterns might yield different non-terminals)
  - 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 non-terminal, follow rules in matrix

- Time linear w.r.t. tree size

[24] AV Aho, M Ganapathi, and SWK Tjiang. "Code generation using tree matching and dynamic programming". In: *TOPLAS* 11.4 (1989), pp. 491–516. .

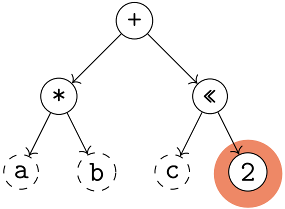
Tree Covering: Dynamic Programming – Example



Node: 2
Pattern:
Pat. Cost:
Cost Sum:

		Node	+	*	«	2
GP	Cost		∞	∞	∞	∞
	Pattern					

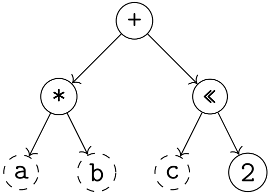
Tree Covering: Dynamic Programming – Example



Node:	2
Pattern:	P_{10}: $GP \to K_1$
Pat. Cost:	1
Cost Sum:	1

		Node	+	*	«	2
GP	Cost		∞	∞	∞	1
	Pattern					P_{10}

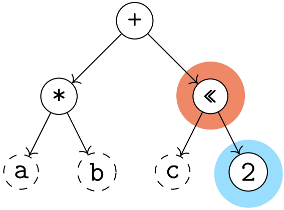
Tree Covering: Dynamic Programming – Example



Node: «
Pattern:
Pat. Cost:
Cost Sum:

		Node	+	*	«	2
GP	Cost		∞	∞	∞	1
	Pattern					P_{10}

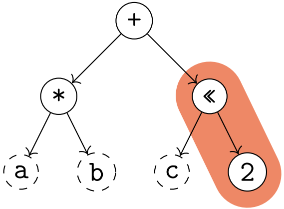
Tree Covering: Dynamic Programming – Example



Node:	«
Pattern:	$P_?$: $GP \rightarrow$ «(GP, GP)
Pat. Cost:	1
Cost Sum:	2

		Node	+	*	«	2
GP	Cost		∞	∞	2	1
	Pattern				$P_?$	P_{10}

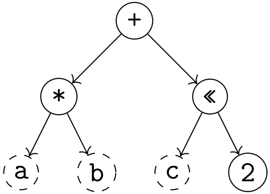
Tree Covering: Dynamic Programming – Example



Node:	«
Pattern:	P_1: $GP \rightarrow$ «(GP, K_1)
Pat. Cost:	1
Cost Sum:	2

		Node	+	*	«	2
GP	Cost		∞	∞	1	1
	Pattern				P_1	P_{10}

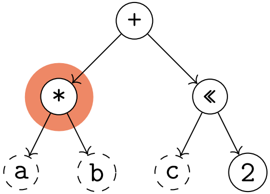
Tree Covering: Dynamic Programming – Example



Node: $*$
Pattern:
Pat. Cost:
Cost Sum:

		Node	+	*	«	2
GP	Cost		∞	∞	1	1
	Pattern				P_1	P_{10}

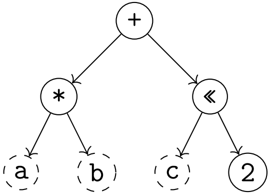
Tree Covering: Dynamic Programming – Example



Node: $*$
Pattern: P_8: $GP \rightarrow *(GP, GP)$
Pat. Cost: 3
Cost Sum: 3

		Node	+	*	«	2
GP	Cost		∞	3	1	1
	Pattern			P_8	P_1	P_{10}

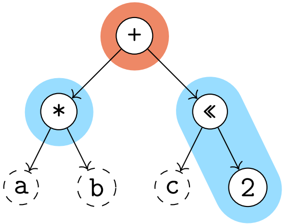
Tree Covering: Dynamic Programming – Example



Node: +
Pattern:
Pat. Cost:
Cost Sum:

		Node	+	*	«	2
GP	Cost		∞	3	1	1
	Pattern			P_8	P_1	P_{10}

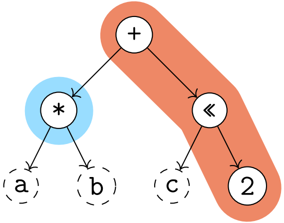
Tree Covering: Dynamic Programming – Example



Node: +
Pattern: P_1: $GP \rightarrow +(GP, GP)$
Pat. Cost: 1
Cost Sum: 5

	Node	+	*	«	2
GP	Cost	5	3	1	1
	Pattern	P_1	P_8	P_1	P_{10}

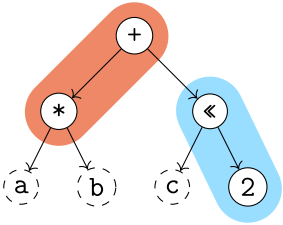
Tree Covering: Dynamic Programming – Example



Node: +
Pattern: P_2: $GP \rightarrow +(GP, «(GP, K_1)$
Pat. Cost: 2
Cost Sum: 5

		Node	+	*	«	2
GP	Cost		5	3	1	1
	Pattern		P_1	P_8	P_1	P_{10}

Tree Covering: Dynamic Programming – Example



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

		Node	+	*	«	2
GP	Cost		4	3	1	1
	Pattern		P_9	P_8	P_1	P_{10}

Tree Covering: Dynamic Programming – Off-line Analysis

- Cost analysis can actually be *precomputed*[25]

- Idea: annotate each node with a state based on child states
- Lookup node label from precomputed table (one per non-terminal)

- Significantly improves compilation time
- But: Tables can be large, need to cover all possible (sub-)trees

- Variation: dynamically compute and cache state tables[26]

[25] 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.

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

Tree Covering

Tree Covering

+ Efficient: linear time to find local optimum
+ Better code than pure macro expansion
+ Applicable to many ISAs

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`)

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 ☹

[27] DR Koes and SC Goldstein. "Near-optimal instruction selection on DAGs". In: *CGO*. 2008, pp. 45–54.

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[27]

[27] DR Koes and SC Goldstein. "Near-optimal instruction selection on DAGs". In: *CGO*. 2008, pp. 45–54.

DAG Covering: Adapting Dynamic Programming I[28]

- 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/non-terminal combinations: avoid redundant emit

[28] MA Ertl. "Optimal code selection in DAGs". In: *POPL*. 1999, pp. 242–249. .

DAG Covering: Adapting Dynamic Programming I[28]

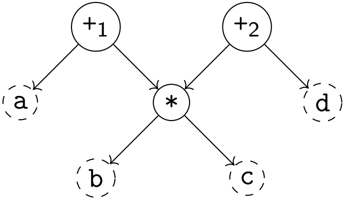
- Step 1: compute cost matrix, bottom-up for all nodes
  - As before; make sure to visit each node once
- Step 2: iterate over DAG top-down
  - Respect that multiple roots exist: start from all roots
  - Mark visited node/non-terminal combinations: avoid redundant emit

\+ Linear time

– Generally not optimal, only for specific grammars

[28] MA Ertl. "Optimal code selection in DAGs". In: *POPL*. 1999, pp. 242–249. .

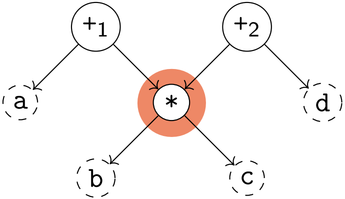
DAG Covering: Adapting Dynamic Programming I – Example



Node: *
Pattern:
Pat. Cost:
Cost Sum:

		Node	$+_2$	$+_1$	*
GP	Cost		∞	∞	∞
	Pattern				

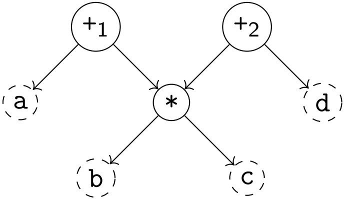
DAG Covering: Adapting Dynamic Programming I – Example



Node:	$*$
Pattern:	P_8: $GP \to *(GP,\ GP)$
Pat. Cost:	3
Cost Sum:	3

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

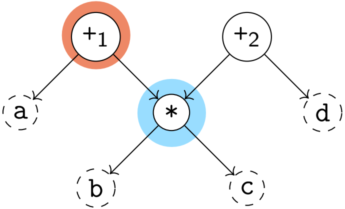
DAG Covering: Adapting Dynamic Programming I – Example



Node: $+_1$
Pattern:
Pat. Cost:
Cost Sum:

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

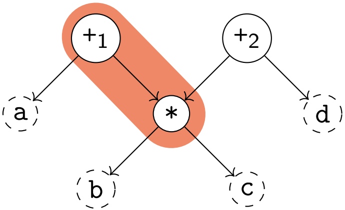
DAG Covering: Adapting Dynamic Programming I – Example



Node: $+_1$
Pattern: P_1: $GP \rightarrow +(GP, GP)$
Pat. Cost: 1
Cost Sum: 4

		Node	$+_2$	$+_1$	$*$
GP	Cost		∞	4	3
	Pattern			P_1	P_8

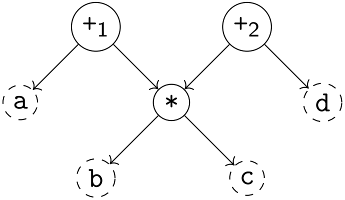
DAG Covering: Adapting Dynamic Programming I – Example



Node:	$+_1$
Pattern:	P_9: $GP \rightarrow +(*(GP, GP), GP)$
Pat. Cost:	3
Cost Sum:	3

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

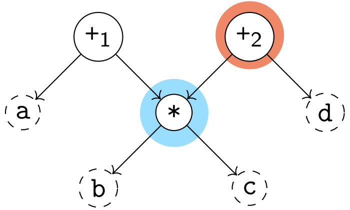
DAG Covering: Adapting Dynamic Programming I – Example



Node: $+_2$
Pattern:
Pat. Cost:
Cost Sum:

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

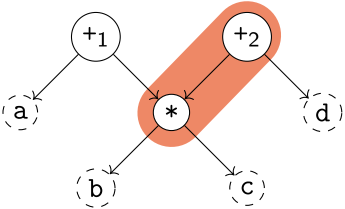
DAG Covering: Adapting Dynamic Programming I – Example



Node: $+_2$
Pattern: P_1: $GP \rightarrow +(GP,\ GP)$
Pat. Cost: 1
Cost Sum: 4

		Node	$+_2$	$+_1$	*
GP	Cost		4	3	3
	Pattern		P_1	P_9	P_8

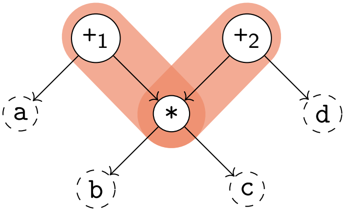
DAG Covering: Adapting Dynamic Programming I – Example



Node: $+_2$
Pattern: P_9: $GP \rightarrow +(*(GP, GP), GP)$
Pat. Cost: 3
Cost Sum: 3

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

DAG Covering: Adapting Dynamic Programming I – Example



Total cost: 6

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

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

DAG Covering: Adapting Dynamic Programming II[29]

- 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

[29] DR Koes and SC Goldstein. "Near-optimal instruction selection on DAGs". In: *CGO*. 2008, pp. 45–54. 🌐.

DAG Covering: Adapting Dynamic Programming II[29]

- 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

[29] DR Koes and SC Goldstein. "Near-optimal instruction selection on DAGs". In: *CGO*. 2008, pp. 45–54. 🌐.

DAG Covering: ILP[30]

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

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

[30] DR Koes and SC Goldstein. "Near-optimal instruction selection on DAGs". In: *CGO*. 2008, pp. 45–54. .

DAG Covering: ILP[30]

- 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

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

\+ Optimal result

– Practicability beyond small programs questionable (at best)

[30] DR Koes and SC Goldstein. "Near-optimal instruction selection on DAGs". In: *CGO*. 2008, pp. 45–54.

DAG Covering: Greedy/Maximal Munch

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

DAG Covering: Greedy/Maximal Munch

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

+ Easy to implement, reasonably fast

– Result often non-optimal

DAG Covering: Greedy/Maximal Munch

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

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

- Used by: LLVM SelectionDAG

Graph Covering

- Idea: lift limitation of DAGs, cover entire function graphs
- Better handling of predication and VLIW bundling
  - E.g., hoisting instructions from a conditional block
- Allows to handle instructions that expand to multiple blocks
  - `switch`, `select`, etc.

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

Graph Covering

- Idea: lift limitation of DAGs, cover entire function graphs
- Better handling of predication and VLIW bundling
  - E.g., hoisting instructions from a conditional block
- Allows to handle instructions that expand to multiple blocks
  - `switch`, `select`, etc.

- May need new IR to model control flow in addition to data flow

- In practice: only used by adapting methods showed for DAGs
- Used by: Java HotSpot Server, LLVM GlobalISel (all tree-covering)

Flawed Assumptions

Flawed Assumptions

- Cost model is fundamentally flawed

$\Rightarrow$ "Optimal" ISel doesn't really mean anything

Flawed Assumptions

- Cost model is fundamentally flawed

⇒ "Optimal" ISel doesn't really mean anything

- Out-of-order execution: costs are not linear
  - Instructions executed in parallel, might execute for free
  - Possible contention of functional units
- Register allocator will modify instructions
- "Bad" instructions boundaries increase register requirements
  - More stack spilling $\rightsquigarrow$ much slower code!

LLVM Back-end: Overview

LLVM Back-end: Overview

- LLVM-IR $\rightarrow$ 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.

LLVM Back-end: Overview

- LLVM-IR $\rightarrow$ 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.)

LLVM Back-end: Overview

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

LLVM MIR Example

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

```
# YAML with name, registers, frame info
body: |
  bb.0 (%ir-block.0):
    liveins: $x0, $x1, $x2

    %2:gpr64 = COPY $x2
    %1:gpr64 = COPY $x1
    %0:gpr64 = COPY $x0
    %3:gpr64 = MADDXrrr %0, %1, $xzr
    %4:gpr64 = ADDXrs killed %3, %2, 2
    $x0 = COPY %4
    RET_ReallyLR implicit $x0
```

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

LLVM: Instruction Selectors

LLVM: Instruction Selectors

FastISel

- Uses macro expansion
- Low compile-time
- Code quality poor

- Only common cases
- Otherwise: fallback to SelectionDAG

- Default for -O0

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

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

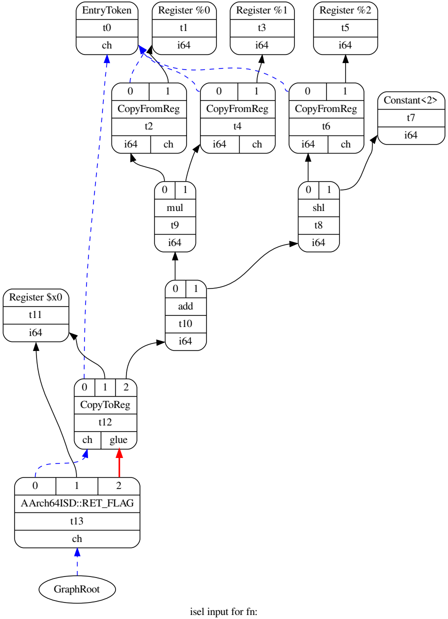
GlobalISel

- Conv. to generic-MIR then legalize to MIR
- Reuses SD patterns
- Faster than SelDAG

- Few architectures
- Handles many cases, SelDAG-fallback

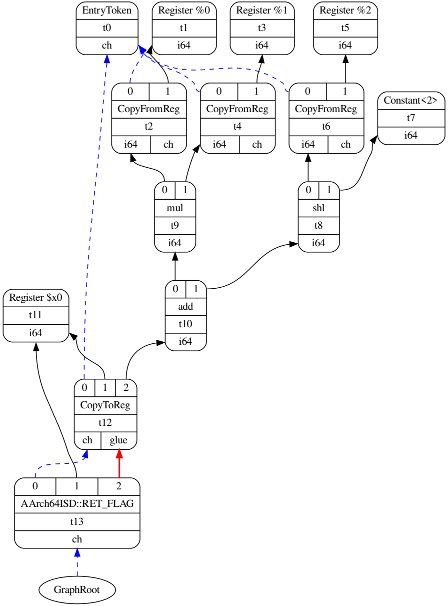
LLVM SelectionDAG: IR to ISelDAG

- Construct DAG for basic block
  - EntryToken as ordering chain



LLVM SelectionDAG: IR to ISelDAG

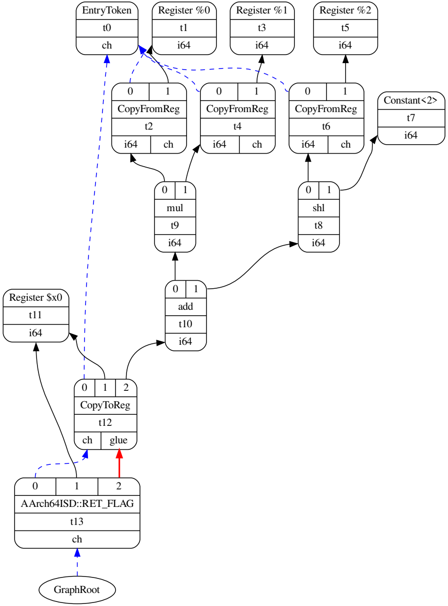
- Construct DAG for basic block
  - EntryToken as ordering chain
- Legalize data types
  - Integers: promote or expand into multiple
  - Vectors: widen or split (or scalarize)



isel input for fn:

LLVM SelectionDAG: IR to ISelDAG

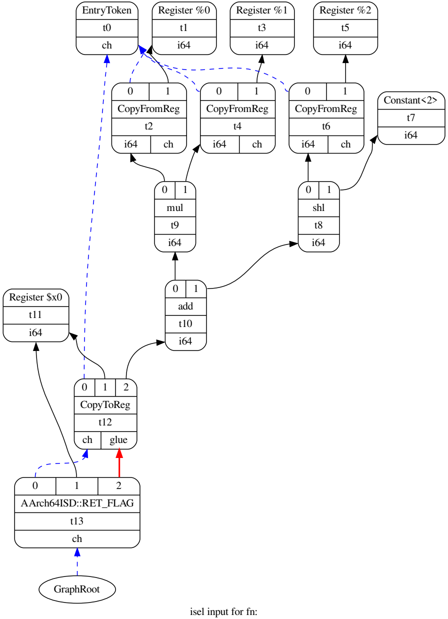
- Construct DAG for basic block
  - EntryToken as ordering chain
- Legalize data types
  - Integers: promote or expand into multiple
  - Vectors: widen or split (or scalarize)
- Legalize operations
  - E.g., conditional move, etc.



isel input for fn:

LLVM SelectionDAG: IR to ISelDAG

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



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

Note: needs LLVM debug build

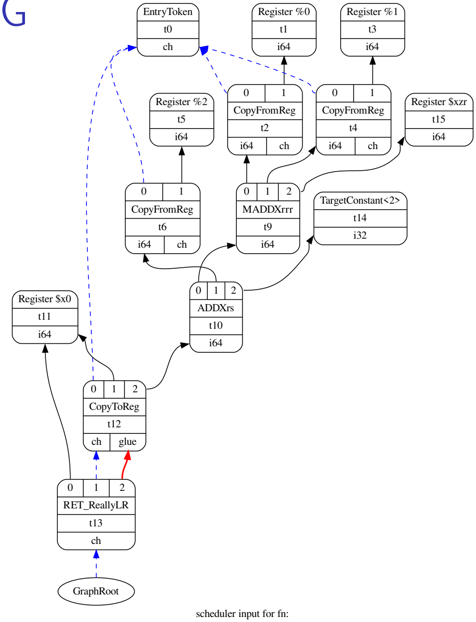
LLVM SelectionDAG: ISelDAG to DAG

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

- Scheduling: linearization of graph

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

Note: needs LLVM debug build



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

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?

Code Generation for Data Processing

Lecture 7: Register Allocation

Alexis Engelke

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

Winter 2022/23

Register Allocation

- Map unlimited/virtual registers to limited/architectural registers

Register Allocation

- Map unlimited/virtual registers to limited/architectural registers
- Assign a register to every value
  - Outputs get a (new) register, input operands often require registers

Register Allocation

- Map unlimited/virtual registers to limited/architectural registers
- Assign a register to every value
  - Outputs get a (new) register, input operands often require registers
- When running out of registers, move values to stack
  - Stack *spilling* – save value register from to stack memory

Register Allocation

- Map unlimited/virtual registers to limited/architectural registers
- Assign a register to every value
  - Outputs get a (new) register, input operands often require registers
- When running out of registers, move values to stack
  - Stack *spilling* – save value register from to stack memory
- ϕ-nodes: ensure all inputs are assigned to same location

Register Allocation

- Map unlimited/virtual registers to limited/architectural registers

- Assign a register to every value
  - Outputs get a (new) register, input operands often require registers
- When running out of registers, move values to stack
  - Stack *spilling* – save value register from to stack memory
- ϕ-nodes: ensure all inputs are assigned to same location

- Goal: produce correct code, minimize extra load/stores
  - Regalloc affects performance in orders of magnitude

Register Allocation: Overview Example

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

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

- May also insert copy and stack spilling instructions

Simplest thing that could possibly work

Simplest thing that could possibly work

- Idea: allocate a one stack slot for every SSA variable/argument

Simplest thing that could possibly work

- Idea: allocate a one stack slot for every SSA variable/argument

- Load all instruction operands into registers right before
- Perform instruction
- Write result back to stack slot for that SSA variable

Simplest thing that could possibly work

- Idea: allocate a one stack slot for every SSA variable/argument

- Load all instruction operands into registers right before
- Perform instruction
- Write result back to stack slot for that SSA variable

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

Regalloc Example 1

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

```
gauss(%0)

  %2 = SUBXri %0, 1

  %3 = MADDXrrr %0, %2, 0

  %4 = MOVXconst 2

  %5 = SDIVrr %3, %4

  ret %5
```

Regalloc Example 1

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

```
gauss(%0 : X0)


  %2 = SUBXri %0, 1


  %3 = MADDXrrr %0, %2, 0

  %4 = MOVXconst 2


  %5 = SDIVrr %3, %4


  ret %5
```

Regalloc Example 1

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

```
gauss(%0 : X0)
  %spills = alloca 8
  STRXi %0, %spills, 0

  %2 = SUBXri %0, 1



  %3 = MADDXrrr %0, %2, 0

  %4 = MOVXconst 2



  %5 = SDIVrr %3, %4


  ret %5
```

Regalloc Example 1

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

```
gauss(%0 : X0)
  %spills = alloca 8
  STRXi %0, %spills, 0
  %10 = LDRXi %spills, 0 : X0
  %2 = SUBXri %10, 1



  %3 = MADDXrrr %0, %2, 0

  %4 = MOVXconst 2




  %5 = SDIVrr %3, %4


  ret %5
```

Regalloc Example 1

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

```
gauss(%0 : X0)
  %spills = alloca 8
  STRXi %0, %spills, 0
  %10 = LDRXi %spills, 0 : X0
  %2 = SUBXri %10, 1 : X0



  %3 = MADDXrrr %0, %2, 0

  %4 = MOVXconst 2



  %5 = SDIVrr %3, %4


  ret %5
```

Regalloc Example 1

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

```
gauss(%0 : X0)
  %spills = alloca 16
  STRXi %0, %spills, 0
  %10 = LDRXi %spills, 0 : X0
  %2 = SUBXri %10, 1 : X0
  STRXi %2, %spills, 8


  %3 = MADDXrrr %0, %2, 0

  %4 = MOVXconst 2



  %5 = SDIVrr %3, %4


  ret %5
```

Regalloc Example 1

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

```
gauss(%0 : X0)
  %spills = alloca 16
  STRXi %0, %spills, 0
  %10 = LDRXi %spills, 0 : X0
  %2 = SUBXri %10, 1 : X0
  STRXi %2, %spills, 8
  %11 = LDRXi %spills, 0 : X0
  %12 = LDRXi %spills, 8 : X1
  %3 = MADDXrrr %0, %2, 0

  %4 = MOVXconst 2



  %5 = SDIVrr %3, %4


  ret %5
```

Regalloc Example 1

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

```
gauss(%0 : X0)
  %spills = alloca 16
  STRXi %0, %spills, 0
  %10 = LDRXi %spills, 0 : X0
  %2 = SUBXri %10, 1 : X0
  STRXi %2, %spills, 8
  %11 = LDRXi %spills, 0 : X0
  %12 = LDRXi %spills, 8 : X1
  %3 = MADDXrrr %11, %12, 0 : X0

  %4 = MOVXconst 2



  %5 = SDIVrr %3, %4


  ret %5
```

Regalloc Example 1

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

```
gauss(%0 : X0)
  %spills = alloca 24
  STRXi %0, %spills, 0
  %10 = LDRXi %spills, 0 : X0
  %2 = SUBXri %10, 1 : X0
  STRXi %2, %spills, 8
  %11 = LDRXi %spills, 0 : X0
  %12 = LDRXi %spills, 8 : X1
  %3 = MADDXrrr %11, %12, 0 : X0
  STRXi %3, %spills, 16
  %4 = MOVXconst 2




  %5 = SDIVrr %3, %4


  ret %5
```

Regalloc Example 1

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

```
gauss(%0 : X0)
  %spills = alloca 24
  STRXi %0, %spills, 0
  %10 = LDRXi %spills, 0 : X0
  %2 = SUBXri %10, 1 : X0
  STRXi %2, %spills, 8
  %11 = LDRXi %spills, 0 : X0
  %12 = LDRXi %spills, 8 : X1
  %3 = MADDXrrr %11, %12, 0 : X0
  STRXi %3, %spills, 16
  %4 = MOVXconst 2 : X0



  %5 = SDIVrr %3, %4


  ret %5
```

Regalloc Example 1

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

```
gauss(%0 : X0)
  %spills = alloca 32
  STRXi %0, %spills, 0
  %10 = LDRXi %spills, 0 : X0
  %2 = SUBXri %10, 1 : X0
  STRXi %2, %spills, 8
  %11 = LDRXi %spills, 0 : X0
  %12 = LDRXi %spills, 8 : X1
  %3 = MADDXrrr %11, %12, 0 : X0
  STRXi %3, %spills, 16
  %4 = MOVXconst 2 : X0
  STRXi %4,i %spills, 24


  %5 = SDIVrr %3, %4


  ret %5
```

Regalloc Example 1

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

```
gauss(%0 : X0)
  %spills = alloca 32
  STRXi %0, %spills, 0
  %10 = LDRXi %spills, 0 : X0
  %2 = SUBXri %10, 1 : X0
  STRXi %2, %spills, 8
  %11 = LDRXi %spills, 0 : X0
  %12 = LDRXi %spills, 8 : X1
  %3 = MADDXrrr %11, %12, 0 : X0
  STRXi %3, %spills, 16
  %4 = MOVXconst 2 : X0
  STRXi %4,i %spills, 24
  %13 = LDRXi %spills, 16 : X0
  %14 = LDRXi %spills, 24 : X1
  %5 = SDIVrr %13, %14


  ret %5
```

Regalloc Example 1

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

```
gauss(%0 : X0)
  %spills = alloca 32
  STRXi %0, %spills, 0
  %10 = LDRXi %spills, 0 : X0
  %2 = SUBXri %10, 1 : X0
  STRXi %2, %spills, 8
  %11 = LDRXi %spills, 0 : X0
  %12 = LDRXi %spills, 8 : X1
  %3 = MADDXrrr %11, %12, 0 : X0
  STRXi %3, %spills, 16
  %4 = MOVXconst 2 : X0
  STRXi %4,i %spills, 24
  %13 = LDRXi %spills, 16 : X0
  %14 = LDRXi %spills, 24 : X1
  %5 = SDIVrr %13, %14 : X0


  ret %5
```

Regalloc Example 1

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

```
gauss(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %10 = LDRXi %spills, 0 : X0
  %2 = SUBXri %10, 1 : X0
  STRXi %2, %spills, 8
  %11 = LDRXi %spills, 0 : X0
  %12 = LDRXi %spills, 8 : X1
  %3 = MADDXrrr %11, %12, 0 : X0
  STRXi %3, %spills, 16
  %4 = MOVXconst 2 : X0
  STRXi %4,i %spills, 24
  %13 = LDRXi %spills, 16 : X0
  %14 = LDRXi %spills, 24 : X1
  %5 = SDIVrr %13, %14 : X0
  STRXi %5, %spills, 32

  ret %5
```

Regalloc Example 1

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

```
gauss(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %10 = LDRXi %spills, 0 : X0
  %2 = SUBXri %10, 1 : X0
  STRXi %2, %spills, 8
  %11 = LDRXi %spills, 0 : X0
  %12 = LDRXi %spills, 8 : X1
  %3 = MADDXrrr %11, %12, 0 : X0
  STRXi %3, %spills, 16
  %4 = MOVXconst 2 : X0
  STRXi %4,i %spills, 24
  %13 = LDRXi %spills, 16 : X0
  %14 = LDRXi %spills, 24 : X1
  %5 = SDIVrr %13, %14 : X0
  STRXi %5, %spills, 32
  %15 = LDRXi %spills, 32 : X0
  ret %15
```

Handling PHI Nodes

- ϕ-node needs to become register or stack slot
  - Simplest thing that could possibly work: PHI becomes stack slot
- Remember: ϕ-nodes are executed on the edge

Handling PHI Nodes

- ϕ-node needs to become register or stack slot
  - Simplest thing that could possibly work: PHI becomes stack slot
- Remember: ϕ-nodes are executed on the edge

- Idea: predecessors write their value to that location at the end
  - First pass: define/allocate storage for ϕ-node, but ignore inputs
  - Second pass: insert move operations at end of predecessors

Regalloc Example 2

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

Pass 1

```
identity(%0)


  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]

  %4 = ADDXri %3, 1



  %5 = CMPXrr_BLS %4, %0
  br %5, %2, %6
6:
  ret %3
```

Regalloc Example 2

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

Pass 1

```
identity(%0 : X0)
  %spills = alloca 8
  STRXi %0, %spills, 0
  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]

  %4 = ADDXri %3, 1



  %5 = CMPXrr_BLS %4, %0
  br %5, %2, %6
6:
  ret %3
```

Regalloc Example 2

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

Pass 1

```
identity(%0 : X0)
  %spills = alloca 16
  STRXi %0, %spills, 0
  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]

  %4 = ADDXri %3, 1



  %5 = CMPXrr_BLS %4, %0
  br %5, %2, %6
6:
  ret %3
```

Regalloc Example 2

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

Pass 1

```
identity(%0 : X0)
  %spills = alloca 16
  STRXi %0, %spills, 0
  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]
  %10 = LDRXi %spills, 8 : X0
  %4 = ADDXri %10, 1 : X0



  %5 = CMPXrr_BLS %4, %0
  br %5, %2, %6
6:
  ret %3
```

Regalloc Example 2

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

Pass 1

```
identity(%0 : X0)
  %spills = alloca 24
  STRXi %0, %spills, 0
  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]
  %10 = LDRXi %spills, 8 : X0
  %4 = ADDXri %10, 1 : X0
  STRXi %4, %spills, 16


  %5 = CMPXrr_BLS %4, %0
  br %5, %2, %6
6:
  ret %3
```

Regalloc Example 2

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

Pass 1

```
identity(%0 : X0)
  %spills = alloca 24
  STRXi %0, %spills, 0
  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]
  %10 = LDRXi %spills, 8 : X0
  %4 = ADDXri %10, 1 : X0
  STRXi %4, %spills, 16
  %11 = LDRXi %spills, 16 : X0
  %12 = LDRXi %spills, 0 : X1
  %5 = CMPXrr_BLS %11, %12
  br %5, %2, %6
6:
  ret %3
```

Regalloc Example 2

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

Pass 1

```
identity(%0 : X0)
  %spills = alloca 24
  STRXi %0, %spills, 0
  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]
  %10 = LDRXi %spills, 8 : X0
  %4 = ADDXri %10, 1 : X0
  STRXi %4, %spills, 16
  %11 = LDRXi %spills, 16 : X0
  %12 = LDRXi %spills, 0 : X1
  %5 = CMPXrr_BLS %11, %12
  br %5, %2, %6
6:%13 = LDRXi %spills, 8 : X0
  ret %13
```

Regalloc Example 2

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

Pass 2

```
identity(%0 : X0)
  %spills = alloca 24
  STRXi %0, %spills, 0
  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]
  %10 = LDRXi %spills, 8 : X0
  %4 = ADDXri %10, 1 : X0
  STRXi %4, %spills, 16
  %11 = LDRXi %spills, 16 : X0
  %12 = LDRXi %spills, 0 : X1
  %5 = CMPXrr_BLS %11, %12
  br %5, %2, %6
6:%13 = LDRXi %spills, 8 : X0
  ret %13
```

Regalloc Example 2

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

Pass 2

```
identity(%0 : X0)
  %spills = alloca 24
  STRXi %0, %spills, 0
  %c0 = MOVXconst 0 : X0
  STRXi %c, %spills, 8
  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]
  %10 = LDRXi %spills, 8 : X0
  %4 = ADDXri %10, 1 : X0
  STRXi %4, %spills, 16
  %11 = LDRXi %spills, 16 : X0
  %12 = LDRXi %spills, 0 : X1
  %5 = CMPXrr_BLS %11, %12
  br %5, %2, %6
6:%13 = LDRXi %spills, 8 : X0
  ret %13
```

Regalloc Example 2

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

Pass 2

```
identity(%0 : X0)
  %spills = alloca 24
  STRXi %0, %spills, 0
  %c0 = MOVXconst 0 : X0
  STRXi %c, %spills, 8
  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]
  %10 = LDRXi %spills, 8 : X0
  %4 = ADDXri %10, 1 : X0
  STRXi %4, %spills, 16
  %14 = LDRXi %spills, 16 : X0
  STRXi %14, %spills, 8
  %11 = LDRXi %spills, 16 : X0
  %12 = LDRXi %spills, 0 : X1
  %5 = CMPXrr_BLS %11, %12
  br %5, %2, %6
6:%13 = LDRXi %spills, 8 : X0
  ret %13
```

Regalloc Example 2

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

```
identity(%0 : X0)
  %spills = alloca 24
  STRXi %0, %spills, 0
  %c0 = MOVXconst 0 : X0
  STRXi %c, %spills, 8
  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]
  %10 = LDRXi %spills, 8 : X0
  %4 = ADDXri %10, 1 : X0
  STRXi %4, %spills, 16
  %14 = LDRXi %spills, 16 : X0
  STRXi %14, %spills, 8
  %11 = LDRXi %spills, 16 : X0
  %12 = LDRXi %spills, 0 : X1
  %5 = CMPXrr_BLS %11, %12
  br %5, %2, %6
6:%13 = LDRXi %spills, 8 : X0
  ret %13
```

Regalloc Example 2

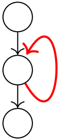
```
identity(%0)
  br %2
2:
  %3 = phi [ 0, %1 ], [ %4, %2 ]
  %4 = ADDXri %3, 1
  %5 = CMPXrr_BLS %4, %0
  br %5, %2, %6
6:
  ret %3
```

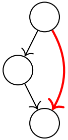
```
identity(%0 : X0)
  %spills = alloca 24
  STRXi %0, %spills, 0
  %c0 = MOVXconst 0 : X0
  STRXi %c, %spills, 8
  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]
  %10 = LDRXi %spills, 8 : X0
  %4 = ADDXri %10, 1 : X0
  STRXi %4, %spills, 16
  %14 = LDRXi %spills, 16 : X0
  STRXi %14, %spills, 8
  %11 = LDRXi %spills, 16 : X0
  %12 = LDRXi %spills, 0 : X1
  %5 = CMPXrr_BLS %11, %12
  br %5, %2, %6
6:%13 = LDRXi %spills, 8 : X0
  ret %13
```

- Original value lost in %6!

Critical Edges

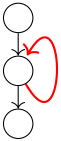
- Critical edge: edge from block with mult. succs. to block with mult. preds.
- Problem: cannot place move on such edges
  - When placing in predecessor, they would also execute for other successor
    $\Rightarrow$ unnecessary and – worse – incorrect

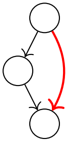




Critical Edges

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





- *Break* critical edges: insert an empty block

Critical Edges

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

- *Break* critical edges: insert an empty block

Regalloc Example 2 – Attempt 2

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

```
identity(%0)


  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]

  %4 = ADDXri %3, 1



  %5 = CMPXrr_BLS %4, %0
  br %5, %6, %7
6:br %2
7:
  ret %3
```

Pass 1

Regalloc Example 2 – Attempt 2

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

```
identity(%0 : X0)
  %spills = alloca 8
  STRXi %0, %spills, 0
  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]

  %4 = ADDXri %3, 1



  %5 = CMPXrr_BLS %4, %0
  br %5, %6, %7
6:br %2
7:
  ret %3
```

Pass 1

Regalloc Example 2 – Attempt 2

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

```
identity(%0 : X0)
  %spills = alloca 16
  STRXi %0, %spills, 0
  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]

  %4 = ADDXri %3, 1



  %5 = CMPXrr_BLS %4, %0
  br %5, %6, %7
6:br %2
7:
  ret %3
```

Pass 1

Regalloc Example 2 – Attempt 2

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

Pass 1

```
identity(%0 : X0)
  %spills = alloca 16
  STRXi %0, %spills, 0
  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]
  %10 = LDRXi %spills, 8 : X0
  %4 = ADDXri %10, 1 : X0



  %5 = CMPXrr_BLS %4, %0
  br %5, %6, %7
6:br %2
7:
  ret %3
```

Regalloc Example 2 – Attempt 2

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

```
identity(%0 : X0)
  %spills = alloca 24
  STRXi %0, %spills, 0
  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]
  %10 = LDRXi %spills, 8 : X0
  %4 = ADDXri %10, 1 : X0
  STRXi %4, %spills, 16


  %5 = CMPXrr_BLS %4, %0
  br %5, %6, %7
6:br %2
7:
  ret %3
```

Pass 1

Regalloc Example 2 – Attempt 2

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

Pass 1

```
identity(%0 : X0)
  %spills = alloca 24
  STRXi %0, %spills, 0
  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]
  %10 = LDRXi %spills, 8 : X0
  %4 = ADDXri %10, 1 : X0
  STRXi %4, %spills, 16
  %11 = LDRXi %spills, 16 : X0
  %12 = LDRXi %spills, 0 : X1
  %5 = CMPXrr_BLS %11, %12
  br %5, %6, %7
6:br %2
7:
  ret %3
```

Regalloc Example 2 – Attempt 2

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

```
identity(%0 : X0)
  %spills = alloca 24
  STRXi %0, %spills, 0
  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]
  %10 = LDRXi %spills, 8 : X0
  %4 = ADDXri %10, 1 : X0
  STRXi %4, %spills, 16
  %11 = LDRXi %spills, 16 : X0
  %12 = LDRXi %spills, 0 : X1
  %5 = CMPXrr_BLS %11, %12
  br %5, %6, %7
6:br %2
7:%13 = LDRXi %spills, 8 : X0
  ret %13
```

Pass 1

Regalloc Example 2 – Attempt 2

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

```
identity(%0 : X0)
  %spills = alloca 24
  STRXi %0, %spills, 0
  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]
  %10 = LDRXi %spills, 8 : X0
  %4 = ADDXri %10, 1 : X0
  STRXi %4, %spills, 16
  %11 = LDRXi %spills, 16 : X0
  %12 = LDRXi %spills, 0 : X1
  %5 = CMPXrr_BLS %11, %12
  br %5, %6, %7
6:br %2
7:%13 = LDRXi %spills, 8 : X0
  ret %13
```

Pass 2

Regalloc Example 2 – Attempt 2

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

```
identity(%0 : X0)
  %spills = alloca 24
  STRXi %0, %spills, 0
  %c0 = MOVXconst 0 : X0
  STRXi %c, %spills, 8
  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]
  %10 = LDRXi %spills, 8 : X0
  %4 = ADDXri %10, 1 : X0
  STRXi %4, %spills, 16
  %11 = LDRXi %spills, 16 : X0
  %12 = LDRXi %spills, 0 : X1
  %5 = CMPXrr_BLS %11, %12
  br %5, %6, %7
6:br %2
7:%13 = LDRXi %spills, 8 : X0
  ret %13
```

Pass 2

Regalloc Example 2 – Attempt 2

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

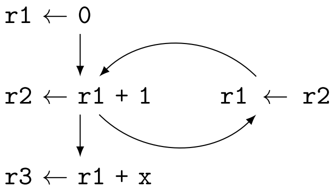
Pass 2

```
identity(%0 : X0)
  %spills = alloca 24
  STRXi %0, %spills, 0
  %c0 = MOVXconst 0 : X0
  STRXi %c, %spills, 8
  br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]
  %10 = LDRXi %spills, 8 : X0
  %4 = ADDXri %10, 1 : X0
  STRXi %4, %spills, 16
  %11 = LDRXi %spills, 16 : X0
  %12 = LDRXi %spills, 0 : X1
  %5 = CMPXrr_BLS %11, %12
  br %5, %6, %7
6:%14 = LDRXi %spills, 16 : X0
  STRXi %14, %spills, 8
  br %2
7:%13 = LDRXi %spills, 8 : X0
  ret %13
```

Handling Critical Edges

Breaking Edges

- Insert new block for moves

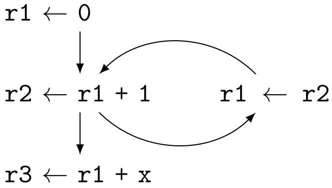


Handling Critical Edges

Breaking Edges

- Insert new block for moves

+ Simple, no analyses needed
− Bad performance in loops



Handling Critical Edges

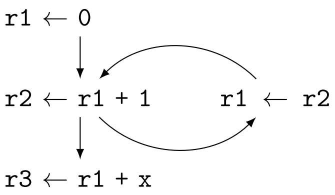
Breaking Edges

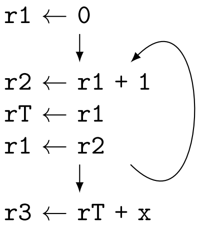
- Insert new block for moves

+ Simple, no analyses needed
− Bad performance in loops

Copy Used Values

- Move values still used to new reg.





Handling Critical Edges

Breaking Edges

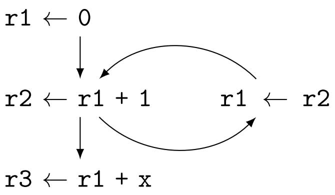
- Insert new block for moves

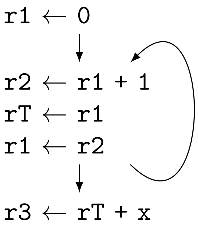
+ Simple, no analyses needed
– Bad performance in loops

Copy Used Values

- Move values still used to new reg.

+ Performance might be better
– Needs more registers





Regalloc Example 3

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

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0



  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32



  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3

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

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0



  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32



  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3

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

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8


  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32
  %14 = LDRXi %spills, 40 : X0; STRXi %14, %spills, 8


  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3

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

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8


  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32
  %14 = LDRXi %spills, 40 : X0; STRXi %14, %spills, 8


  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3

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

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16

  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32
  %14 = LDRXi %spills, 40 : X0; STRXi %14, %spills, 8
  %15 = LDRXi %spills, 24 : X0; STRXi %15, %spills, 16

  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3

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

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16

  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32
  %14 = LDRXi %spills, 40 : X0; STRXi %14, %spills, 8
  %15 = LDRXi %spills, 24 : X0; STRXi %15, %spills, 16

  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3

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

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16
  %c1 = MOVXconst 0 : X0; STRXi %c1, %spills, 16
  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32
  %14 = LDRXi %spills, 40 : X0; STRXi %14, %spills, 8
  %15 = LDRXi %spills, 24 : X0; STRXi %15, %spills, 16
  %16 = LDRXi %spills, 16 : X0; STRXi %16, %spills, 24
  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3

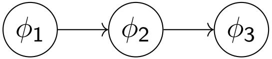
```
odd(%0)
  br %2
2:
  %3 = phi [ %0, %1 ], [ %8, %7 ]
  %4 = phi [ 1, %1 ], [ %5, %7 ]
  %5 = phi [ 0, %1 ], [ %4, %7 ]
  %6 = CBNZX(%3)
  br %6, %7, %9
7:
  %8 = SUBXri %3, 1
  br %2
9:
  ret %4
```

- Value of ϕ node lost!

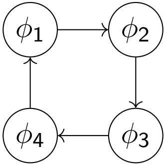
```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16
  %c1 = MOVXconst 0 : X0; STRXi %c1, %spills, 16
  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32
  %14 = LDRXi %spills, 40 : X0; STRXi %14, %spills, 8
  %15 = LDRXi %spills, 24 : X0; STRXi %15, %spills, 16
  %16 = LDRXi %spills, 16 : X0; STRXi %16, %spills, 24
  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

PHI Cycles

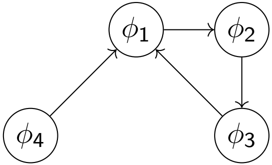
- Problem: ϕ-nodes can depend on each other
- Can be chains (ordering matters) or cycles (need to be broken)
- Note: only ϕ-nodes defined in same block are relevant/problematic



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



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



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

Handling PHI Cycles

Handling PHI Cycles

1. Compute number of other ϕ nodes reading other ϕ on same edge

Handling PHI Cycles

1. Compute number of other ϕ nodes reading other ϕ on same edge
2. For each ϕ with 0 readers: handle node/chain
   - No readers $\rightsquigarrow$ start of chain
   - Handling node may unblock next element in chain

Handling PHI Cycles

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

Handling PHI Cycles

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

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

Regalloc Example 3 – Attempt 2

Edge %1 → %2

Critical ϕ:

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0



  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32



  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3 – Attempt 2

Edge %1 → %2

Critical ϕ:

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0



  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32




  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3 – Attempt 2

Edge %1 → %2

Critical ϕ:

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8


  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32




  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3 – Attempt 2

Edge %1 → %2

Critical ϕ:

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8


  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32




  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3 – Attempt 2

Edge %1 → %2

Critical ϕ:

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16

  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32




  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3 – Attempt 2

Edge %1 → %2

Critical ϕ:

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16

  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32




  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3 – Attempt 2

Edge %1 $\rightarrow$ %2

Critical ϕ:

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16
  %c1 = MOVXconst 0 : X0; STRXi %c1, %spills, 16
  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32




  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3 – Attempt 2

Edge %7 → %2

Critical ϕ:

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16
  %c1 = MOVXconst 0 : X0; STRXi %c1, %spills, 16
  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32




  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3 – Attempt 2

Edge %7 → %2

Critical ϕ:

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16
  %c1 = MOVXconst 0 : X0; STRXi %c1, %spills, 16
  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32
  %14 = LDRXi %spills, 40 : X0; STRXi %14, %spills, 8



  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3 – Attempt 2

Edge %7 → %2

Critical ϕ:

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16
  %c1 = MOVXconst 0 : X0; STRXi %c1, %spills, 16
  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32
  %14 = LDRXi %spills, 40 : X0; STRXi %14, %spills, 8



  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3 – Attempt 2

Edge %7 → %2

Critical ϕ:

- %4

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16
  %c1 = MOVXconst 0 : X0; STRXi %c1, %spills, 16
  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32
  %14 = LDRXi %spills, 40 : X0; STRXi %14, %spills, 8



  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3 – Attempt 2

Edge %7 → %2

Critical ϕ:

- %4

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16
  %c1 = MOVXconst 0 : X0; STRXi %c1, %spills, 16
  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32
  %14 = LDRXi %spills, 40 : X0; STRXi %14, %spills, 8



  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3 – Attempt 2

Edge %7 → %2

Critical ϕ:

- %4
- %5

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16
  %c1 = MOVXconst 0 : X0; STRXi %c1, %spills, 16
  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32
  %14 = LDRXi %spills, 40 : X0; STRXi %14, %spills, 8



  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3 – Attempt 2

Edge %7 → %2

Critical ϕ:

- %4
- %5

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16
  %c1 = MOVXconst 0 : X0; STRXi %c1, %spills, 16
  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32
  %14 = LDRXi %spills, 40 : X0; STRXi %14, %spills, 8



  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3 – Attempt 2

Edge %7 → %2

Critical ϕ:

- %4 #readers: 1
- %5 #readers: 1

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16
  %c1 = MOVXconst 0 : X0; STRXi %c1, %spills, 16
  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32
  %14 = LDRXi %spills, 40 : X0; STRXi %14, %spills, 8



  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3 – Attempt 2

Edge %7 → %2

Critical ϕ:

- %4 #readers: 1 – broken
- %5 #readers: 1

Action: break %4

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16
  %c1 = MOVXconst 0 : X0; STRXi %c1, %spills, 16
  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32
  %14 = LDRXi %spills, 40 : X0; STRXi %14, %spills, 8



  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3 – Attempt 2

Edge %7 → %2

Critical ϕ:

- %4 #readers: 1 – broken
- %5 #readers: 0

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16
  %c1 = MOVXconst 0 : X0; STRXi %c1, %spills, 16
  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32
  %14 = LDRXi %spills, 40 : X0; STRXi %14, %spills, 8
  %15 = LDRXi %spills, 24 : X1


  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3 – Attempt 2

Edge %7 → %2

Critical ϕ:

- %4 #readers: 1 – broken
- %5 #readers: 0

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16
  %c1 = MOVXconst 0 : X0; STRXi %c1, %spills, 16
  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32
  %14 = LDRXi %spills, 40 : X0; STRXi %14, %spills, 8
  %15 = LDRXi %spills, 24 : X1


  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3 – Attempt 2

Edge %7 → %2

Critical ϕ:

- %4 #readers: 0 – broken
- %5 #readers: 0

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16
  %c1 = MOVXconst 0 : X0; STRXi %c1, %spills, 16
  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32
  %14 = LDRXi %spills, 40 : X0; STRXi %14, %spills, 8
  %15 = LDRXi %spills, 24 : X1
  %16 = LDRXi %spills, 16 : X0; STRXi %16, %spills, 24

  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Regalloc Example 3 – Attempt 2

Edge %7 → %2

Critical ϕ:

- %4 #readers: 0 – broken
- %5 #readers: 0

```
odd(%0 : X0)
  %spills = alloca 40
  STRXi %0, %spills, 0
  %13 = LDRXi %spills, 0 : X0; STRXi %13, %spills, 8
  %c0 = MOVXconst 1 : X0; STRXi %c0, %spills, 16
  %c1 = MOVXconst 0 : X0; STRXi %c1, %spills, 16
  br %2
2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
  %4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
  %5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
  %10 = LDRXi %spills, 8 : X0
  %6 = CBNZX(%10)
  br %6, %7, %9
7:%11 = LDRXi %spills, 8 : X0
  %8 = SUBXri %12, 1 : X0; STRXi %8, %spills, 32
  %14 = LDRXi %spills, 40 : X0; STRXi %14, %spills, 8
  %15 = LDRXi %spills, 24 : X1
  %16 = LDRXi %spills, 16 : X0; STRXi %16, %spills, 24
  STRXi %15, %spills, 16
  br %2
9:%12 = LDRXi %spills, 24 : X0
  ret %12
```

Better Register Allocation

- Goal:

Better Register Allocation

- Goal: keep as many values in registers as possible
  - Less stack spilling $\Rightarrow$ better performance

Better Register Allocation

- Goal: keep as many values in registers as possible
  - Less stack spilling $\Rightarrow$ better performance
- Problem: register count (severely) limited

$\rightsquigarrow$ Are there enough registers? (otherwise: spilling)

$\rightsquigarrow$ Which register to choose?

$\rightsquigarrow$ Which register to kill and put on the stack?

Register Allocation: Research

- *Tons* of papers exist
- Papers are academic

Register Allocation: Research

- *Tons* of papers exist
- Papers often skip over important details
  - E.g., when spilling – using the value needs another register
  - E.g., temporary register for shuffling values
- Additional (ISA) constraints in practice: (incomplete list)
  - 2-address instructions with destructive source
  - Fixed registers for specific instructions
  - Computing the stack address may need yet another register
  - Different register classes, often just handled independently

Register Allocation: Research

- *Tons* of papers exist
- Papers often skip over important details
  - E.g., when spilling – using the value needs another register
  - E.g., temporary register for shuffling values
- Additional (ISA) constraints in practice: (incomplete list)
  - 2-address instructions with destructive source
  - Fixed registers for specific instructions
  - Computing the stack address may need yet another register
  - Different register classes, often just handled independently
- Implementations even of simple algorithms tend to be large and complex

Liveness Analysis

- *Live:* value still used afterwards
  - After last (possible) use in program flow, the value becomes dead
- *Live ranges:* set of ranges in program where value is live
  - Not necessarily contiguous, e.g. in case of branches
- *Live interval:* over-approximation of live ranges without holes
  - Depends on block order, reverse post-order often a good choice

Liveness Analysis on SSA[31]

- For each block *liveIn*: values that are needed at block entry
- Construct live ranges for each SSA value

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

[31] C Wimmer and M Franz. "Linear scan register allocation on SSA form". In: *CGO*. 2010, pp. 170–179.

Linear Scan Register Allocation[32]

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

[32] M Poletto and V Sarkar. "Linear scan register allocation". In: *TOPLAS* 21.5 (1999), pp. 895–913.

Linear Scan Register Allocation

Linear Scan Register Allocation

- + low compile-time, simple, used for JIT-compilers and Go
- − very suboptimal code, live intervals grossly over-approximated

- What's missing?

Linear Scan Register Allocation

+ low compile-time, simple, used for JIT-compilers and Go
− very suboptimal code, live intervals grossly over-approximated

- What's missing?
  - Registers to load spilled values and shuffle values
  - Register constraints (e.g., for insts. or function calls)

Linear Scan Register Allocation

+ low compile-time, simple, used for JIT-compilers and Go
− very suboptimal code, live intervals grossly over-approximated

- What's missing?
  - Registers to load spilled values and shuffle values
  - Register constraints (e.g., for insts. or function calls)
- Other disadvantage: once a value is spilled, it is always spilled
- Function calls: clobber lots of registers

Linear Scan – Adaption (Engelke, 2022)

- Run linear scan, but forcefully keep one free register before ϕ-nodes
  - For register constraints, forcefully evict value occupying the register

Linear Scan – Adaption (Engelke, 2022)

- Run linear scan, but forcefully keep one free register before ϕ-nodes
  - For register constraints, forcefully evict value occupying the register
- Emit spill code and add new live intervals
  - Spill store: immediately store to stack, adds short live interval
  - Spill loads: load operands to new reg., adds short live intervals

Linear Scan – Adaption (Engelke, 2022)

- Run linear scan, but forcefully keep one free register before ϕ-nodes
  - For register constraints, forcefully evict value occupying the register
- Emit spill code and add new live intervals
  - Spill store: immediately store to stack, adds short live interval
  - Spill loads: load operands to new reg., adds short live intervals
- Repeat until no extra intervals/spills are inserted

Making Linear Scan Non-Linear (and better)

Making Linear Scan Non-Linear (and better)

- Don't spill variable forever, but split life time once necessary[33]
  - When no register is free, spill a register, but only from this point on
  - On reload, keep copy in register (but keep stack slot until end)

[33] O Traub, G Holloway, and MD Smith. "Quality and speed in linear-scan register allocation". In: *SIGPLAN* 33.5 (1998), pp. 142–151. 🌐.

Making Linear Scan Non-Linear (and better)

- Don't spill variable forever, but split life time once necessary[33]
  - When no register is free, spill a register, but only from this point on
  - On reload, keep copy in register (but keep stack slot until end)
- Base spill decision on next use (instead of lifetime end)[34]
  - Additionally keep track of next use distance during analysis
  - Benefit: better spill decisions; downside: superlinear run-time

[33] O Traub, G Holloway, and MD Smith. "Quality and speed in linear-scan register allocation". In: *SIGPLAN* 33.5 (1998), pp. 142–151. 🌐.

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

Making Linear Scan Non-Linear (and better)

- Don't spill variable forever, but split life time once necessary[33]
  - When no register is free, spill a register, but only from this point on
  - On reload, keep copy in register (but keep stack slot until end)
- Base spill decision on next use (instead of lifetime end)[34]
  - Additionally keep track of next use distance during analysis
  - Benefit: better spill decisions; downside: superlinear run-time
- Propagate register preferences bottom-up[35]
  - Better assignment for function calls/fixed register operands

[33] O Traub, G Holloway, and MD Smith. "Quality and speed in linear-scan register allocation". In: *SIGPLAN* 33.5 (1998), pp. 142–151. 🌐.

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

[35] `https://github.com/golang/go/blob/5f7abe/src/cmd/compile/internal/ssa/regalloc.go` e.g. lines 2604–2636

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

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
- Easy case: all nodes have degree $\leq k$

Chaitin's Algorithm[36]

- Find node with fewer than k edges

- Remove it from the graph
- Recursively color the rest of the graph
- Add note back in and assign valid color

[36] GJ Chaitin. "Register allocation & spilling via graph coloring". In: *SIGPLAN* 17.6 (1982), pp. 98–101. 🌐.

Chaitin's Algorithm[36]

- Find node with fewer than k edges
  - If no such node exists: pick one and spill to stack
  - Selection based on heuristics
  - Update interference graph
- Remove it from the graph
- Recursively color the rest of the graph
- Add note back in and assign valid color

[36] GJ Chaitin. "Register allocation & spilling via graph coloring". In: *SIGPLAN* 17.6 (1982), pp. 98–101. 🌐.

Graph Coloring Approaches

Graph Coloring Approaches

+ Considerably better results than greedy algorithms

Graph Coloring Approaches

+ Considerably better results than greedy algorithms
– High run-time, even with heuristics

- Graph coloring in general is $\mathcal{NP}$-complete
- Often used in compilers (e.g., GCC, WebKit)

AD	IN2053 “Program Optimization” covers this more formally

Register Selection and Spilling

- Avoid spilling values in loops
- Avoid spilling values used immediately afterwards
- Prefer callee-saved register for values live across function calls
  - Function call clobbers caller-saved regs $\rightsquigarrow$ cheaper call

Register Selection and Spilling

- Avoid spilling values in loops
- Avoid spilling values used immediately afterwards
- Prefer callee-saved register for values live across function calls
  - Function call clobbers caller-saved regs $\rightsquigarrow$ cheaper call
- Spill slots can be reused for different values
  - Better use of stack, but higher complexity

Register Selection and Spilling

- Avoid spilling values in loops
- Avoid spilling values used immediately afterwards
- Prefer callee-saved register for values live across function calls
  - Function call clobbers caller-saved regs $\rightsquigarrow$ cheaper call

- Spill slots can be reused for different values
  - Better use of stack, but higher complexity

- Spilling to FP/vector registers...
  - Occasionally proposed, rarely done in practice

Stack Frame Allocation

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

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

Stack Frame Allocation

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

Block Ordering

- Order blocks to make use of fall-through in machine code
- Avoid sequences of `b.cond; b`
  - Sometimes cannot be avoided: conditional branches often have shorter range

- Block ordering has implications for branch prediction
  - Forward branches default to not-taken, backward taken
  - Unlikely blocks placed "out of the way" of the main execution path
  - Indirect branches are predicted as fall-through

Register Allocation – Summary

- Map unlimited virtual registers to restricted register set
- Responsible for:
  - Assigning registers to values
  - Deciding which registers to spill to stack
  - Deciding when to spill/unspill values
- ϕ-nodes require extra care, esp. for chains and cycles
- Liveness information is key information for register allocation
- Linear-time algorithms exist, but have suboptimal results
- Register allocation/spilling relies on heuristics in practice

Register Allocation – Questions

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

Code Generation for Data Processing

Lecture 8: Object Files, Linker, and Loader

Alexis Engelke

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

Winter 2022/23

Overview: Post-compilation

- Compiler emits object file
  - Somehow? Some format?

Overview: Post-compilation

- Compiler emits object file
  - Somehow? Some format?
- Linker merges object files and determines required shared libraries
  - Somehow resolves missing symbols?

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?

Overview: Post-compilation

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

Code Model and Position Independent Code

- Code Model = address constraints
- Allows for better code
  - Long addrs/offsets = more instrs.
- Exact constraints arch-specific

Code Model and Position Independent Code

- Code Model = address constraints
- Allows for better code
  - Long addrs/offsets = more instrs.
- Exact constraints arch-specific

- x86-64 SysV ABI:
  - Small: code and data max. 2 GiB
  - Medium: code max. 2 GiB
  - Large: no restrictions

Code Model and Position Independent Code

- Code Model = address constraints
- Allows for better code
  - Long addrs/offsets = more instrs.
- Exact constraints arch-specific

- x86-64 SysV ABI:
  - Small: code and data max. 2 GiB
  - Medium: code max. 2 GiB
  - Large: no restrictions

- non-PIC: absolute addresses fixed at link-time
  - Addrs can be encoded directly
  - Sometime slightly faster
  - Not possible for shared libs

Code Model and Position Independent Code

- Code Model = address constraints
- Allows for better code
  - Long addrs/offsets = more instrs.
- Exact constraints arch-specific

- x86-64 SysV ABI:
  - Small: code and data max. 2 GiB
  - Medium: code max. 2 GiB
  - Large: no restrictions

- non-PIC: absolute addresses fixed at link-time
  - Addrs can be encoded directly
  - Sometime 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)

Code Model and Position Independent Code

- Code Model = address constraints
- Allows for better code
  - Long addrs/offsets = more instrs.
- Exact constraints arch-specific

- x86-64 SysV ABI:
  - Small: code and data max. 2 GiB
  - Medium: code max. 2 GiB
  - Large: no restrictions

- non-PIC: absolute addresses fixed at link-time
  - Addrs can be encoded directly
  - Sometime slightly faster
  - Not possible for shared libs

- PIC: address random at load time
  - Offsets need be PC-relative
  - Addresses need fixup at load time (e.g., in jump tables)

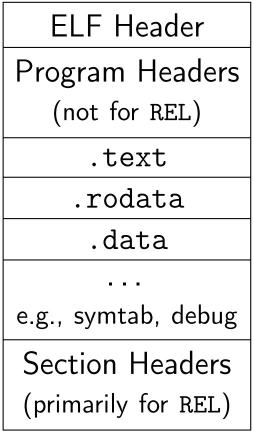
Compiler needs to know code model

Section 19

Object Files

Executable and Linkable Format (ELF)

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



ELF Header

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

ELF Sections

- Structures content of object files for linker
  - Linker later merges content sections of same "type"
- Some sections have "meta" information (e.g., symbols)

- `.text` –

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` –

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` –

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` –

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
- `.symtab` –

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
- `.symtab` – symbol table, references string table for names
- `.strtab` –

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
- `.symtab` – symbol table, references string table for names
- `.strtab` – string table for symbol names
- `.shstrtab` –

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
- `.symtab` – symbol table, references string table for names
- `.strtab` – string table for symbol names
- `.shstrtab` – string table for section header

ELF Section Header

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

Example: Section Headers

```c
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
```

Symbol Table

- Describes symbolic reference to object/function
- Names in associated string table, referenced by byte offset

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

Example: Symbol Table

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

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

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

```
Symbol table '.symtab' contains 7 entries:
 Num: Val  Size Type    Bind   Vis      Ndx Name
   0: 000     0 NOTYPE  LOCAL  DEFAULT  UND
   1: 000     0 FILE    LOCAL  DEFAULT  ABS <stdin>
   2: 000     0 SECTION LOCAL  DEFAULT    1 .text
   3: 000     1 FUNC    LOCAL  DEFAULT    1 bar
   4: 001     6 FUNC    GLOBAL DEFAULT    1 foo
   5: 007    19 FUNC    GLOBAL DEFAULT    1 func
   6: 000     0 NOTYPE  GLOBAL DEFAULT  UND external
```

Example: Writing Code to .text

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

```
0000000000000000 <bar>:
   0:   c3                 ret
0000000000000001 <foo>:
   1:   e8 ?? ?? ?? ?? call    ???
   6:   c3                 ret
0000000000000007 <func>:
   7:   48 83 ec 08        sub     rsp,0x8
   b:   e8 ?? ?? ?? ?? call    ???
  10:   e8 ?? ?? ?? ?? call    ???
  15:   48 83 c4 08        add     rsp,0x8
  19:   c3                 ret
```

Example: Writing Code to .text

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

```
0000000000000000 <bar>:
   0:   c3                  ret
0000000000000001 <foo>:
   1:   e8 fa ff ff ff      call   0 <bar>
   6:   c3                  ret
0000000000000007 <func>:
   7:   48 83 ec 08         sub    rsp,0x8
   b:   e8 ?? ?? ?? ??      call   ???
  10:   e8 ?? ?? ?? ??      call   ???
  15:   48 83 c4 08         add    rsp,0x8
  19:   c3                  ret
```

Example: Writing Code to .text

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

```
0000000000000000 <bar>:
   0:   c3               ret
0000000000000001 <foo>:
   1:   e8 fa ff ff ff call    0 <bar>
   6:   c3               ret
0000000000000007 <func>:
   7:   48 83 ec 08      sub     rsp,0x8
   b:   e8 00 00 00 00 call    10 <func+0x9>
      c: R_X86_64_PC32^a         foo-0x4
  10:   e8 ?? ?? ?? ?? call    ???
  15:   48 83 c4 08      add     rsp,0x8
  19:   c3               ret
```

[a] Recent GAS emits R_X86_64_PLT32, which is equivalent for local symbols.

Example: Writing Code to .text

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

```
0000000000000000 <bar>:
   0:   c3                  ret
0000000000000001 <foo>:
   1:   e8 fa ff ff ff call    0 <bar>
   6:   c3                  ret
0000000000000007 <func>:
   7:   48 83 ec 08         sub     rsp,0x8
   b:   e8 00 00 00 00 call    10 <func+0x9>
        c: R_X86_64_PC32^a         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
```

[a] Recent GAS emits R_X86_64_PLT32, which is equivalent for local symbols.

Example: Writing Code to .text

```c
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_PC32^a          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
```

[a] Recent GAS emits R_X86_64_PLT32, which is equivalent for local symbols.

Relocations

- Problem: symbol values unknown before linking
  - External symbols: unavailable; other section: distance unknown
- Idea: store *relocations* $\Rightarrow$ linker patches code/data

Relocations

- Problem: symbol values unknown before linking
  - External symbols: unavailable; other section: distance unknown
- Idea: store *relocations* $\Rightarrow$ linker patches code/data

- Relocation: quadruple of (offset in sec., type, symbol idx, addend)
- Contained in `REL`/`RELA`/`RELR` sections

Relocations

- Problem: symbol values unknown before linking
  - External symbols: unavailable; other section: distance 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

Relocation Types

- Types and meaning defined by psABI[37]

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

Name	Field	Calculation

Name	Field	Calculation

[37] x86-64: HJ Lu et al. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*. 2022.

Relocation Types

- Types and meaning defined by psABI[37]

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

Name	Field	Calculation	Name	Field	Calculation
`R_X86_64_PC32`	32	$S + A - P$			

[37] x86-64: HJ Lu et al. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*. 2022.

Relocation Types

- Types and meaning defined by psABI[37]

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_PC32`	32	$S + A - P$
`R_X86_64_PLT32`	32	$L + A - P$

Name	Field	Calculation

[37] x86-64: HJ Lu et al. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*. 2022. .

Relocation Types

- Types and meaning defined by psABI[37]

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	$Z + A$

[37] x86-64: HJ Lu et al. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*. 2022. 🌐.

Relocation Section

```
Section Headers:
  [Nr] Name        Type     Size   ES Flg Lk Inf Al
  [ 1] .text       PROGBITS 00001a 00  AX  0   0  1
  [ 2] .rela.text  RELA     000030 18   I 10   1  8
              sizeof(Elf64_Rela) --/    |  |   |
         I: info is section link ------/  |   |
                  link to 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
```

Relocations on RISC Architectures

- RISC architectures typically have *more* relocation types
  - Example: AArch64[38] 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

[38] Arm Ltd. *ELF for the Arm 64-bit Architecture (AArch64)*. (visited on 11/21/2022).

Branch Relocations

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

Branch Relocations

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

Branch Relocations

- Branches (often) have limited range; compiler must assume max. distance
- x86-64: ± 2 GiB range, if larger use `mov` and indirect jump
- AArch64: ± 128 MiB range $\rightsquigarrow$ executable sections must be <127 MiB linker will insert veneer between different `.text` sections
  - Veneer allowed to clobber inter-procedural scratch registers `x16`/`x17`
- *badly designed ISA*: ± 1 MiB range $\rightsquigarrow$ needs ind. jump *often*
  - Construct 20 bit first with `auipc`, insert low 12 bit in `jalr`

Branch Relocations

- Branches (often) have limited range; compiler must assume max. distance
- x86-64: $\pm$2 GiB range, if larger use `mov` and indirect jump
- AArch64: $\pm$128 MiB range $\rightsquigarrow$ executable sections must be $<$127 MiB
  linker will insert veneer between different `.text` sections
  - Veneer allowed to clobber inter-procedural scratch registers `x16`/`x17`
- *badly designed ISA*: $\pm$1 MiB range $\rightsquigarrow$ needs ind. jump *often*
  - Construct 20 bit first with `auipc`, insert low 12 bit in `jalr`
  - Add new *relax* reloc: linker optimizes/relaxes code if possible

Branch Relocations

- Branches (often) have limited range; compiler must assume max. distance
- x86-64: $\pm$2 GiB range, if larger use `mov` and indirect jump
- AArch64: $\pm$128 MiB range $\rightsquigarrow$ executable sections must be $<$127 MiB linker will insert veneer between different `.text` sections
  - Veneer allowed to clobber inter-procedural scratch registers `x16`/`x17`
- *badly designed ISA*: $\pm$1 MiB range $\rightsquigarrow$ needs ind. jump *often*
  - Construct 20 bit first with `auipc`, insert low 12 bit in `jalr`
  - Add new *relax* reloc: linker optimizes/relaxes code if possible
  - Changes code size, all relative offsets now need relocations, too

Branch Relocations

- Branches (often) have limited range; compiler must assume max. distance
- x86-64: $\pm$2 GiB range, if larger use `mov` and indirect jump
- AArch64: $\pm$128 MiB range $\rightsquigarrow$ executable sections must be $<$127 MiB
  linker will insert veneer between different `.text` sections
  - Veneer allowed to clobber inter-procedural scratch registers `x16`/`x17`
- *badly designed ISA*: $\pm$1 MiB range $\rightsquigarrow$ needs ind. jump *often*
  - Construct 20 bit first with `auipc`, insert low 12 bit in `jalr`
  - Add new *relax* reloc: linker optimizes/relaxes code if possible
  - Changes code size, all relative offsets now need relocations, too
  - Alignment guarantees need new, special `align` relocations

Section 20

Executable Files

Linker[39]

- Goal: combine multiple input files (.o/.so/.a) into executable or shared lib.

[39] Interesting blog on LLD: F Song. *Personal Blog*. (visited on 11/21/2022).

Linker[39]

- Goal: combine multiple input files (.o/.so/.a) into executable or shared lib.

1. Find and load all input files

[39] Interesting blog on LLD: F Song. *Personal Blog*. (visited on 11/21/2022).

Linker[39]

- 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

[39] Interesting blog on LLD: F Song. *Personal Blog*. (visited on 11/21/2022).

Linker[39]

- 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)

[39] Interesting blog on LLD: F Song. *Personal Blog*. (visited on 11/21/2022).

Linker[39]

- 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.

[39] Interesting blog on LLD: F Song. *Personal Blog*. (visited on 11/21/2022).

Linker[39]

- 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

[39] Interesting blog on LLD: F Song. *Personal Blog*. (visited on 11/21/2022).

Linker[39]

- 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.

[39] Interesting blog on LLD: F Song. *Personal Blog*. (visited on 11/21/2022).

Linker[39]

- 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!

[39] Interesting blog on LLD: F Song. *Personal Blog*. (visited on 11/21/2022).

ELF Executable File

- Entry in ELF header: entry address of the program
  - Typically provided by libc to call `__libc_start_main`
- Program headers: instructions for loading the program

ELF Executable File

- Entry in ELF header: entry address of the program
  - Typically provided by libc to call `__libc_start_main`
- Program headers: instructions for loading the program

- `PT_PHDR`: described program headers

ELF Executable File

- Entry in ELF header: entry address of the program
  - Typically provided by libc to call `__libc_start_main`
- Program headers: instructions for loading the program

- `PT_PHDR`: described program headers
- `PT_LOAD`: loadable segment
  - Specifies virtual address, file offset, file size/memory size, permission
  - `vaddr&(pgsize-1)==offset&(pgsize-1)` – kernel will just `mmap` the file
  - memory size > file size ⇒ filled up with zeros (for `.bss`)

ELF Executable File

- Entry in ELF header: entry address of the program
  - Typically provided by libc to call `__libc_start_main`
- Program headers: instructions for loading the program

- `PT_PHDR`: described program headers
- `PT_LOAD`: loadable segment
  - Specifies virtual address, file offset, file size/memory size, permission
  - `vaddr&(pgsize-1)==offset&(pgsize-1)` – kernel will just `mmap` the file
  - 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

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

Loading a Binary to Memory

Loading a Binary to Memory

- Load ELF header and program header

Loading a Binary to Memory

- Load ELF header and program header
- If `ET_DYN` ($\rightsquigarrow$ PIE), set random base added to all addresses

Loading a Binary to Memory

- Load ELF header and program header
- If `ET_DYN` ($\rightsquigarrow$ 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)

Loading a Binary to Memory

- Load ELF header and program header
- If ET_DYN ($\rightsquigarrow$ 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

Loading a Binary to Memory

- Load ELF header and program header
- If `ET_DYN` ($\rightsquigarrow$ 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)

Loading a Binary to Memory

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

This is the kernel's job

Section 21

Linker Optimizations

Eliminating Duplicate Strings/Constants

- Sections in different object may contain same data, e.g. strings
  - Critical for debug info (file names, function names, etc.)
- Idea: linker finds and deduplicates strings and other constant data
- Precondition: relative order of entries irrelevant

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

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

Linker Garbage Collection

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

Linker Garbage Collection

- Problem: objects may contain unused functions
  - Compiler can't know whether function is used
- Idea: put all function into separate sections, drop unused sections

- Sections are considered as inseparable units
- GC roots:

Linker Garbage Collection

- Problem: objects may contain unused functions
  - Compiler can't know whether function is used
- Idea: put all function into separate sections, drop unused sections

- Sections are considered as inseparable units
- GC roots: exported symbols, init functions, ...
- Iteratively mark all referenced sections, drop unmarked sections

- Downside:

Linker Garbage Collection

- Problem: objects may contain unused functions
  - Compiler can't know whether function is used
- Idea: put all function into separate sections, drop unused sections

- Sections are considered as inseparable units
- GC roots: exported symbols, init functions, ...
- Iteratively mark all referenced sections, drop unmarked sections

- Downside: may need longer relocations $\rightsquigarrow$ possibly less efficient code
- GCC/Clang `-ffunction-sections`, `ld --gc-sections`

Identical Code Folding

- Problem: objects may contain duplicate code
  - Same function compiled in many objs, e.g. template instantiation

Identical Code Folding

- Problem: objects may contain duplicate code
  - Same function compiled in many objs, e.g. template instantiation
- Idea: deduplicate read-only sections (same flags, contents, relocations(!))

- Hash all sections and their relocations, remove duplicates
- Repeat until convergence
  - Only after folding `foo1` and `foo2`, these become equivalent:

```
int funcA(void) { foo1(); } int funcB(void) { foo2(); }
```

- Caution: function pointers may be guaranteed to be different
- LLD has more aggressive deduplication

Link-Time Optimization

- Problem: Compilers still suck

Link-Time Optimization

- Problem: no optimizations across object files
  - Inlining, constant propagation+cloning, specialized call conv., ...
  - Optimization across language boundaries

Link-Time Optimization

- Problem: no optimizations across object files
  - Inlining, constant propagation+cloning, specialized call conv., ...
  - Optimization across language boundaries
- Idea 1: glue all source code together, compile with `-fwhole-program`
  - Downside: single core, problematic with same-name `static` functions

Link-Time Optimization

- Problem: no optimizations across object files
  - Inlining, constant propagation+cloning, specialized call conv., ...
  - Optimization across language boundaries

- Idea 1: glue all source code together, compile with `-fwhole-program`
  - Downside: single core, problematic with same-name `static` functions
- Idea 2: Use static binary optimization during linking (severely limited)

Link-Time Optimization

- Problem: no optimizations across object files
  - Inlining, constant propagation+cloning, specialized call conv., ...
  - Optimization across language boundaries

- Idea 1: glue all source code together, compile with `-fwhole-program`
  - Downside: single core, problematic with same-name `static` functions
- Idea 2: Use static binary optimization during linking (severely limited)
- Idea 3: dump IR into object, glue IR together (`-flto`)
  - Done as very first step at link-time

- LTO is widely used and highly effective

Section 22

Static Libraries

Static Libraries

- Archive of relocatable object files
- Header often contains index mapping symbol to object file
- Linker takes only object files that are needed
- Code/data copied into final executable

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

Static Libraries

- Archive of relocatable object files
- Header often contains index mapping symbol to object file
- Linker takes only object files that are needed
- Code/data copied into final executable

+ Simple and fast, no ABI problems, no extra library needed at run-time

− Larger executable files, library changes need relinking

Section 23

Shared Libraries

Shared Libraries

- Problem: code duplication, large executables, recompile needed for changes
- Idea: *share* code between different executables

Shared Libraries

- Problem: code duplication, large executables, recompile needed for changes
- Idea: *share* code between different executables

- Executable references functions/objects in shared library
  - Shared libraries can refer to other shared libraries, too
  - Linker needs to retain dynamic relocations and symbols
    (dynamic symbol = externally visible symbol)
- Run-time loader links executable and libraries program start
  - Find and load libraries from different paths, resolve all relocations

Shared Libraries: Changes in Compiler

Shared Libraries: Changes in Compiler

None :-)

(almost)

Shared Libraries: Changes in Compiler

None $\smile$
(almost)

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

Shared Libraries: Changes in Linker

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`)

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

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

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, ...

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

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
```

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

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

```
00401030 <func@plt>:
  401030: ff 25 8a 2f 00 00 jmp    QWORD PTR [rip+0x2f8a] # GOT slot
```

- PLT can be disabled (`-fno-plt`): indirect jump is duplicated
  - Compiler emits indirect calls/jumps instead of near calls to PLT
  - Linker cannot convert into near jump if target is in same DSO

PT_DYNAMIC segment

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

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

Symbol Lookup

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

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

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)

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[40]

[40] A Roenky. *ELF: better symbol lookup via DT_GNU_HASH.* (visited on 12/14/2022)

Miscellaneous Things

Miscellaneous Things

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

Miscellaneous Things

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

Miscellaneous Things

- Purpose of all these dynamic entries
- Symbols: versioning and visibility
- Constructors/destructors: called at load/unload of DSO
- Indirect functions (ifunc)
  - Function to dynamically determine actual address of symbol
  - Used e.g. for determining `memcpy` variant based on CPU features

- Dynamic loading of DSOs (`dlopen`)

Object Files, Linker, and Loader – Summary

- Compiler needs to know code model to emit proper asm code/relocations
- ELF format used for relocatable files, executables and shared libraries
- ELF relocatables structured in sections and have static relocations
- ELF dynamic executables grouped in segments and have dynamic relocations
  - Need dynamic loader to resolve dynamic relocations and shared libraries
- Linker combines relocatable files into executables or shared libraries
- Linker can perform further optimizations

Object Files, Linker, and Loader – Questions

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

Code Generation for Data Processing

Lecture 9: Unwinding and Debuginfo

Alexis Engelke

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

Winter 2022/23

Motivation: Meta-Information on Program

Motivation: Meta-Information on Program

- Machine code suffices for execution

Motivation: Meta-Information on Program

- Machine code suffices for execution $\rightarrow$ not true

- Needs program headers and entry point
- Linking with shared libraries needs dynamic symbols and interpreter

Motivation: Meta-Information on Program

- Machine code suffices for execution $\rightarrow$ 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.

Adding Meta-Information with GCC

Adding Meta-Information with GCC

-g

Adding Meta-Information with GCC

`-g`

`-fexceptions`

Adding Meta-Information with GCC

`-g`

`-fexceptions`

`-fasynchronous-unwind-tables`

Adding Meta-Information with GCC

`-g`

`-fexceptions`

`-fasynchronous-unwind-tables`

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

Stack Unwinding

- Needed for exceptions (`_Unwind_RaiseException`) or forced unwinding

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

Stack Unwinding

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

Stack Unwinding: Requirements

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

Stack Unwinding: Requirements

- Given: current register values in unwind function
- Need: iterate through stack frames
  - Get address of function of the stack frame
  - Get pc and sp for *this function*
  - Find personality function and language-specific data
  - Maybe get some registers from the stack frame
  - Update some registers with exception data
- Increased difficulty: stepping through signal handler

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`

Stack Unwinding: `setjmp/longjmp`

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

\+ Needs no extra information

– High overhead in non-exceptional case

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

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

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

Stack Unwinding: 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)

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

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

Stack Unwinding: 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)
- Still needs to find meta-information
- Need to distinguish prologue with wrong info

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

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

Stack Unwinding: Without Frame Pointer

- Given: `pc` and `sp` (bottom of stack frame/call frame)
  - In parent frames: $\mathit{retaddr} - 1$ ~`pc` and *CFA* ~`sp`
- Need to map `pc` to stack frame size
  - `sp`+*framesize* = *CFA* (canonical frame address – `sp` at `call`)
  - Stack frame size varies throughout function, e.g. prologue

Stack Unwinding: Without Frame Pointer

- Given: `pc` and `sp` (bottom of stack frame/call frame)
  - In parent frames: $\textit{retaddr} - 1 \sim$`pc` and $\textit{CFA} \sim$`sp`
- Need to map `pc` to stack frame size
  - `sp`$+\textit{framesize} = \textit{CFA}$ (canonical frame address – `sp` at `call`)
  - Stack frame size varies throughout function, e.g. prologue
- Case 1: some register used as frame pointer – CFA constant offset to `fp`
  - E.g., for variable stack frame size
- Case 2: no frame pointer: CFA is constant offset to `sp`

Stack Unwinding: Without Frame Pointer

- Given: `pc` and `sp` (bottom of stack frame/call frame)
  - In parent frames: $\mathit{retaddr} - 1 \sim$`pc` and $\mathit{CFA} \sim$`sp`
- Need to map `pc` to stack frame size
  - `sp`$+\mathit{framesize} = \mathit{CFA}$ (canonical frame address – `sp` at `call`)
  - Stack frame size varies throughout function, e.g. prologue

- Case 1: some register used as frame pointer – CFA constant offset to `fp`
  - E.g., for variable stack frame size
- Case 2: no frame pointer: CFA is constant offset to `sp`

⇝ Unwinding *must* restore register values
  - Other reg. can act as frame pointer, register saved in other register, ...
  - Need to know where return address is stored

Call Frame Information

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

Call Frame Information – Example 1

	CFA	rip	rbx	rbp	...
foo:					
0x0: push rbx					
0x1: mov ebx, edi					
0x3: call bar					
0x8: mov eax, ebx					
0xa: pop rbx					
0xb: ret					

Call Frame Information – Example 1

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

Call Frame Information – Example 1

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

Call Frame Information – Example 1

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

Call Frame Information – Example 1

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

Call Frame Information – Example 1

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

Call Frame Information – Example 1

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

Call Frame Information – Example 1

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

Call Frame Information – Example 1

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

Call Frame Information – Example 1

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

Call Frame Information – Example 1

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

Call Frame Information – Example 1

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

Call Frame Information – Example 1

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

Call Frame Information – Example 1

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

Call Frame Information – Example 1

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

Call Frame Information – Example 2

	CFA	rip	rbx	rbp	...
foo:					
0x0: push rbp					
0x1: mov rbp, rsp					
0x4: shl rdi, 4					
0x8: sub rsp, rdi					
0xb: mov rdi, rsp					
0xe: call bar					
0x13: leave					
0x14: ret					

Call Frame Information – Example 2

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

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					
0x8: sub rsp, rdi					
0xb: mov rdi, rsp					
0xe: call bar					
0x13: leave					
0x14: ret					

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					
0x14: ret					

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					

Call Frame Information – Example 2

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

Call Frame Information – Example 3

	CFA	rip	rbx	rbp	...
foo:					
0x0: sub rsp, 8					
0x4: test edi, edi					
0x6: js 0x12					
0x8: call positive					
0xd: add rsp, 8					
0x11: ret					
0x12: call negative					
0x17: add rsp, 8					
0x1a: ret					

Call Frame Information – Example 3

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

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					
0x8: call positive					
0xd: add rsp, 8					
0x11: ret					
0x12: call negative					
0x17: add rsp, 8					
0x1a: ret					

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					
0xd: add rsp, 8					
0x11: ret					
0x12: call negative					
0x17: add rsp, 8					
0x1a: ret					

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					
0x11: ret					
0x12: call negative					
0x17: add rsp, 8					
0x1a: ret					

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					
0x12: call negative					
0x17: add rsp, 8					
0x1a: ret					

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					
0x17:	add rsp, 8					
0x1a:	ret					

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					
0x1a:	ret					

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					

Call Frame Information – Example 3

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

Call Frame Information: Encoding

- Expanded table can be huge
- Contents change rather seldomly
  - Mainly in prologue/epilogue, but mostly constant in-between

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

Call Frame Information: Bytecode – Example 1

	CFA	rip	rbx
foo:			
0: push rbx			
1: mov ebx, edi			
3: call bar			
8: mov eax, ebx			
a: pop rbx			
b: ret			

```
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
```

Call Frame Information: Bytecode – Example 1

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

```
=> 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
```

Call Frame Information: Bytecode – Example 1

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

```
   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
```

Call Frame Information: Bytecode – Example 1

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

```
   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
```

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]	
3: call bar			
8: mov eax, ebx			
a: pop rbx			
b: ret			

```
   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
```

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			
8: mov eax, ebx			
a: pop rbx			
b: ret			

```
   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
```

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+16	[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
```

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
```

Call Frame Information: Bytecode – Example 2

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

```
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
```

Call Frame Information: Bytecode – Example 2

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

```
=> 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
```

Call Frame Information: Bytecode – Example 2

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

```
   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
```

Call Frame Information: Bytecode – Example 2

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

```
   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
```

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]	
4: shl rdi, 4			
8: sub rsp, rdi			
b: mov rdi, rsp			
e: call bar			
13: leave			
14: ret			

```
   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
```

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			
8: sub rsp, rdi			
b: mov rdi, rsp			
e: call bar			
13: leave			
14: ret			

```
   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
```

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	rsp+16	[CFA-8]	[CFA-16]
8: sub rsp, rdi			
b: mov rdi, rsp			
e: call bar			
13: leave			
14: ret			

```
   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
```

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			
b: mov rdi, rsp			
e: call bar			
13: leave			
14: ret			

```
   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
```

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	rbp+16	[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
```

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
```

Call Frame Information: Bytecode – Example 3

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

```
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: 9
DW_CFA_def_cfa_offset: +8
```

Call Frame Information: Bytecode – Example 3

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

```
=> 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: 9
   DW_CFA_def_cfa_offset: +8
```

Call Frame Information: Bytecode – Example 3

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

```
   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: 9
   DW_CFA_def_cfa_offset: +8
```

Call Frame Information: Bytecode – Example 3

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

```
   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: 9
   DW_CFA_def_cfa_offset: +8
```

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`		
`8: call positive`		
`d: add rsp, 8`		
`11: ret`		
`12: call negative`		
`17: add rsp, 8`		
`1a: ret`		

```
   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: 9
   DW_CFA_def_cfa_offset: +8
```

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+16	[CFA-8]
`12:   call negative`		
`17:   add rsp, 8`		
`1a:   ret`		

```
   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: 9
   DW_CFA_def_cfa_offset: +8
```

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+16	[CFA-8]
12: call negative		
17: add rsp, 8		
1a: ret		

```
   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: 9
   DW_CFA_def_cfa_offset: +8
```

Remember stack: {`CFA=rsp+16; rip=[CFA-8]`}

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		
17:	add rsp, 8		
1a:	ret		

```
   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: 9
   DW_CFA_def_cfa_offset: +8
```

Remember stack: {`CFA=rsp+16; rip=[CFA-8]`}

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+8	[CFA-8]
17: add rsp, 8		
1a: ret		

```
   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: 9
   DW_CFA_def_cfa_offset: +8
```

Remember stack: {`CFA=rsp+16; rip=[CFA-8]`}

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		
1a: ret		

```
   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: 9
   DW_CFA_def_cfa_offset: +8
```

Remember stack: {}

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+16	[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: 9
   DW_CFA_def_cfa_offset: +8
```

Remember stack: {}

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: 9
=> DW_CFA_def_cfa_offset: +8
```

Remember stack: {}

Call Frame Information: Bytecode

- DWARF[41] 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>`

[41] DWARF Debugging Information Committee. *DWARF Debugging Information Format Version 5*. Feb. 2017. .

Call Frame Information: .eh_frame_hdr

- Problem:

Call Frame Information: .eh_frame_hdr

- Problem: linear search over – possibly many – FDEs is slow
- Idea:

Call Frame Information: `.eh_frame_hdr`

- 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

Call Frame Information: `.eh_frame_hdr`

- Problem: linear search over – possibly many – FDEs is slow
- Idea: create binary search table over FDEs at link-time

- Ordered list of all function addresses and their FDE
- Unwinder does binary search to find matching FDE

- Separate program header entry: `PT_GNU_EH_FRAME`
- Unwinder needs loader support to find these
  - `_dl_find_object` or `dl_iterate_phdr`
- FDEs and indices are cached to avoid redundant lookups

Call Frame Information: Assembler Directives

- Compilers produces textual CFI
- Assembler encodes CFI into binary format
  - Allows for integration of annotated inline assembly
  - Inline-asm also needs CFI directives
- Register numbers specified by psABI

- Wrap function with `.cfi_startproc`/`.cfi_endproc`
- Many directives map straight to DWARF instructions
  - `.cfi_def_cfa_offset 16`; `.cfi_offset %rbp, -16`; `.cfi_def_cfa_register %rbp`

Call Frame Information: Assembler Directives – Example

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

gcc -O -S foo.c

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

Unwinding: Other Platforms

- Unwinding depends *strongly* on OS and architecture

Unwinding: Other Platforms

- Unwinding depends *strongly* on OS and architecture

- Linux uses DWARF
- Apple has modified version
- Windows has SEH with kernel-support for unwinding
- IBM AIX has their own format
- AArch32 has another custom format

Unwinding: Other Platforms

- Unwinding depends *strongly* on OS and architecture

- Linux uses DWARF
- Apple has modified version
- Windows has SEH with kernel-support for unwinding
- IBM AIX has their own format
- AArch32 has another custom format

- Additionally: minor differences for return address, stack handling, ...

Unwinding: Other Platforms

- Unwinding depends *strongly* on OS and architecture

- Linux uses DWARF
- Apple has modified version
- Windows has SEH with kernel-support for unwinding
- IBM AIX has their own format
- AArch32 has another custom format

- Additionally: minor differences for return address, stack handling, ...

Needs to work reliably for exception handling

Debugging: Wanted Features

Debugging: Wanted Features

- Get back trace
- Map address to source file/line

- Show global and local variables
  - Local variables need scope information, e.g. shadowing
  - Data type information, e.g. int, string, struct, enum

- Set break point at line/function
  - Might require multiple actual breakpoints: inlining, template expansion
- Step through program by line/statement

Debugging: Wanted Features

- Get back trace $\rightsquigarrow$ CFI
- Map address to source file/line

- Show global and local variables
  - Local variables need scope information, e.g. shadowing
  - Data type information, e.g. int, string, struct, enum

- Set break point at line/function
  - Might require multiple actual breakpoints: inlining, template expansion
- Step through program by line/statement

Line Table

- Map instruction to: file/line/column; start of stmt; start of basic block; is prologue/epilogue; ISA mode
- Table can be huge; idea:

Line Table

- Map instruction to: file/line/column; start of stmt; start of basic block; is prologue/epilogue; ISA mode
- Table can be huge; idea: encode as bytecode

- Extracted information are bytecode registers
- Conceptually similar to CFI encoding

- `llvm-dwarfdump -v --debug-line` or `readelf -wlL`

Debugging: Wanted Features

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

Debugging: Wanted Features

- Get back trace ⇝ CFI
- Map address to source file/line ⇝ Line Table

- Show global and local variables
  - Local variables need scope information, e.g. shadowing
  - Data type information, e.g. int, string, struct, enum

- Set break point at line/function ⇝ Line Table/??
  - Might require multiple actual breakpoints: inlining, template expansion
- Step through program by line/statement ⇝ Line Table

DWARF: Hierarchical Program Description

- Extensible, flexible, Turing-complete[42] 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

[42] J Oakley and S Bratus. "Exploiting the Hard-Working DWARF: Trojan and Exploit Techniques with No Native Executable Code". In: *WOOT*. 2011.

DWARF: Hierarchical Program Description

- Extensible, flexible, Turing-complete[42] 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, ...

[42] J Oakley and S Bratus. "Exploiting the Hard-Working DWARF: Trojan and Exploit Techniques with No Native Executable Code". In: *WOOT*. 2011.

DWARF: Data Types

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

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")
```

DWARF: Data Types

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

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

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

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

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

DWARF: Variables

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

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

DWARF: Expressions

- Very general way to describe location of value:

DWARF: Expressions

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

DWARF: Program Structure

- Follows structure of code

- Top-level: compilation unit
- Entries for namespaces, subroutines (functions)
  - Functions can contain inlined subroutines
- Lexical blocks to group variables
- Call sites and parameters

- Each node annotated with pc-range and source location

Debugging: Wanted Features

- Get back trace $\rightsquigarrow$ CFI
- Map address to source file/line $\rightsquigarrow$ Line Table

- Show global and local variables
  - Local variables need scope information, e.g. shadowing
  - Data type information, e.g. int, string, struct, enum

- Set break point at line/function $\rightsquigarrow$ Line Table/??
  - Might require multiple actual breakpoints: inlining, template expansion
- Step through program by line/statement $\rightsquigarrow$ Line Table

Debugging: Wanted Features

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

Other Debuginfo Formats

- DWARF is big despite compression
- Cannot run in time-constrained environments
  - 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

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

Unwinding and Debuginfo – Questions

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

Code Generation for Data Processing

Lecture 10: JIT Compilation and Sandboxing

Alexis Engelke

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

Winter 2022/23

JIT Compilation

- Ahead-of-Time compilation not always possible/sufficient

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[43]

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

JIT Compilation: Simple Approach

JIT Compilation: Simple Approach

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

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

JIT Compilation: Simple Approach

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

- Example: libgccjit

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

JIT: Allocating Memory

- `malloc()`

JIT: Allocating Memory

- `malloc()` – memory often non-executable
- `alloca()`

JIT: Allocating Memory

- `malloc()` – memory often non-executable
- `alloca()` – memory often non-executable
- `mmap(PROT_READ|PROT_WRITE|PROT_EXEC)`

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

JIT: Allocating Memory

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

JIT: Making Code Executable

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

JIT: Making Code Executable

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

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[44] (Linux), all non-x86 (Darwin): system call

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

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[44] (Linux), all non-x86 (Darwin): system call

- Skipping ICache flush: spurious, hard-to-debug problems

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

Code Generation: Differences AoT vs. JIT

Code Generation: Differences AoT vs. JIT

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

- JIT compiler and linker are often merged

JIT: Code Model

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

JIT: Relocations and Symbols

- JIT compiler must take care of relocations
  - Can try to directly process relocations during machine code gen.
  - Not always possible: cyclic dependencies
  - Option: behave like normal compiler with separate runtime linker

JIT: Relocations and Symbols

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

JIT: Memory Layout

- *Never* place code and (writable) data on same page
  - $W \oplus X$; and writes near code can trigger self-modifying code detection
  - Avoid many small allocations with one page each
  - But: editing existing code pages is problematic

JIT: Memory Layout

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

JIT: .eh_frame Registration (required for C++)

- Unwinder finds .eh_frame

JIT: .eh_frame Registration (required for C++)

- Unwinder finds .eh_frame using program headers
- Problem: JIT-compiled code has no program headers
- Idea: JIT compiler registers new code with runtime

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

JIT: GDB Debuginfo Registration (optional)

- GDB finds debug info from section headers of DSOs
- Problem: JIT-compiled code has no DSO

JIT: GDB Debuginfo Registration (optional)

- GDB finds debug info from section headers of DSOs
- Problem: JIT-compiled code has no DSO
- Idea: JIT compiler registers new code with debugger

- Define function `__jit_debug_register_code` and global var. `__jit_debug_descriptor`
  - Call function on update; GDB places breakpoint in function
  - Prevent function from being inlined
- Descriptor is linked list of in-memory object files
  - Needs relocations applied, also for debug info

- Users: LLVM, Wasmtime, HHVM, ...; consumers: GDB, LLDB

JIT: Linux `perf` Registration (optional)

- perf tracks binary through backing file of `mmap`

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

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

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`[45]
  - Text file; format: `startaddr size name\n`
- Approach 2: *needs an extra slide*

[45] `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/perf/Documentation/jit-interface.txt`

JIT: Linux perf JITDUMP format (optional)

- JIT-compiler dumps function name/address/size/code[46]
  - JITDUMP file: record list for each function, may contain debuginfo
  - File name must be `jit-<PID>.dump`

[46] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/perf/Documentation/jitdump-specification.txt

JIT: Linux `perf` JITDUMP format (optional)

- JIT-compiler dumps function name/address/size/code[46]
  - JITDUMP file: record list for each function, may contain debuginfo
  - File name must be `jit-<PID>.dump`
- JIT-compiler `mmap`s 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

[46] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/perf/Documentation/jitdump-specification.txt

JIT: Linux perf JITDUMP format (optional)

- JIT-compiler dumps function name/address/size/code[46]
  - JITDUMP file: record list for each function, may contain debuginfo
  - File name must be `jit-<PID>.dump`
- JIT-compiler `mmaps` part of the file as executable somewhere
  - Only use: perf keeps track of executable mappings $\rightsquigarrow$ mapping is JIT marker, s.t. perf can find the file later
- Need to run `perf report` with `-k 1` to use monotonic clock
- After profiling: `perf inject --jit -i perf.data -o jit.data`
  - Extracts functions from JITDUMP, each into its own ELF file
  - Changes mappings of profile to refer to newly created files

[46] `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/perf/Documentation/jitdump-specification.txt`

JIT: Linux perf JITDUMP format (optional)

- JIT-compiler dumps function name/address/size/code[46]
  - JITDUMP file: record list for each function, may contain debuginfo
  - File name must be `jit-<PID>.dump`
- JIT-compiler `mmaps` part of the file as executable somewhere
  - Only use: perf keeps track of executable mappings ⇝ mapping is JIT marker, s.t. perf can find the file later
- Need to run `perf report` with `-k 1` to use monotonic clock
- After profiling: `perf inject --jit -i perf.data -o jit.data`
  - Extracts functions from JITDUMP, each into its own ELF file
  - Changes mappings of profile to refer to newly created files
- `perf report -i jit.data` – Profit!

[46] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/perf/Documentation/jitdump-specification.txt

Compilation Time

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

Compilation Time

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

- Idea: adaptive compilation
- Incrementally spend more time on optimization

Compilation Time: Simple Approach

Compilation Time: Simple Approach

Caching

Compilation Time: Simple Approach

Caching

- Doesn't work on first execution

Adaptive Execution

- Execution tiers have different compile-time/run-time tradeoffs
  - Bytecode interpreter: very fast/slow
  - Fast compiler: medium/medium
  - Optimizing compiler: slow/fast

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 no next tier
  - Depends on profile information, e.g. only optimize hot code
  - Compile in background, switch when ready

Adaptive Execution: Switching Tiers

- Switching only possible at compiler-defined points
  - Needs to serialize relevant state for other tier

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

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

Adaptive Execution: Switching Tiers

- Switching only possible at compiler-defined points
  - Needs to serialize relevant state for other tier
- Simple approach: only switch at function boundaries
  - Simple, well-defined boundaries; unable to switch inside loop
- Complex approach: allow switching at loop headers/everywhere
  - Needs tracking of much more meta-information
  - All entry points need well-defined interface
  - All exit points need info to recover complete state
  - Severely limits optimizations; all loops become irreducible

- Using LLVM is possible, but not a good fit

Adaptive Execution: Partial Compilation and Speculation

- Observation: even in hot functions, many branches are rarely used
- Optimizing cold code is wasted time(/energy)

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

Adaptive Execution: Partial Compilation and Speculation

- Observation: even in hot functions, many branches are rarely used
- Optimizing cold code is wasted time(/energy)

- Observation (JS): functions often get called with same data type
- Specializing on structure allows removing string lookup for fields

- Idea: speculate on common path using profiling data
- Add check whether speculation holds; if not, use side-exit
  - Side-exit can be patched later with actual code
- Side-exit must serialize all relevant state for lower tier
  - "Deoptimization"

Sandboxing

Sandboxing

- Executing untrusted code without additional measures may harm system
- Untrusted input may expose vulnerabilities

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

Sandboxing

- Executing untrusted code without additional measures may harm system
- Untrusted input may expose vulnerabilities

- Goal 1: execute untrusted code without impacting security
  - Code in higher-level representation allows for further analyses but needs JIT compilation for performance
- Goal 2: limit impact potential of new vulnerabilities

- Other goals: portability, resource usage, performance, usability, language flexibility

Approach: Sandbox Operating System as-is

- Idea: put entire operating system in sandbox (“virtual machine”)
- Widely used in practice

- Virtualization needs hardware and OS support
  - CPU has hypervisor mode which controls guest OS;
    offers nested paging, hypercalls from guest OS to hypervisor

Approach: Sandbox Operating System as-is

- Idea: put entire operating system in sandbox ("virtual machine")
- Widely used in practice

- Virtualization needs hardware and OS support
  - CPU has hypervisor mode which controls guest OS;
    offers nested paging, hypercalls from guest OS to hypervisor

+ Good usability and performance
+ Strong isolation
– Rather high overhead on resource usage: completely new OS
– Inflexible and high start latency (seconds)

Approach: Sandbox Native Code as-is

- Idea: strongly restrict possibilities of native code

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

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

Approach: Sandbox Native Code as-is

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

Approach: Sandbox Native Code as-is

- Frequently and widely used (“container”)

Approach: Sandbox Native Code as-is

- Frequently and widely used (“container”)

+ Good usability and performance, low latency (milliseconds)
+ Finer grained control of resources

∼ Resource usage: often completely new user space

– Weak isolation: OS+CPU often bad at separation
  - Kernel has a fairly large interface, not hardened against bad actors
  - Privilege escalation happens not rarely

Approach: Sandbox Native Code with Modification

- 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

Approach: Sandbox Native Code with Modification

- Idea: enforce limitations on machine code
  - Define restrictions on machine code, e.g. no unbounded memory access
  - Modify compiler to comply with restrictions
  - Verify program at load time

- Google Native Client[47], originally x86-32, ported to x86-64 and ARM
- Designed as browser extension
- Native code shipped to browser, executed after validation

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

NaCl Constraints on i386

- Problem: dynamic code not verifiable

NaCl Constraints on i386

- Problem: dynamic code not verifiable
  - $\Rightarrow$ No self-modifying/dynamically generated code
- Problem: overlapping instructions

NaCl Constraints on i386

- Problem: dynamic code not verifiable
  - ⇒ No self-modifying/dynamically generated code
- Problem: overlapping instructions
  - ⇒ All "valid" instructions must be reachable in linear disassembly
  - ⇒ Direct jumps must target valid instructions
  - ⇒ No instruction may cross 32-byte boundary
  - ⇒ Indirect jumps/returns must be `and eax, -32; jmp eax`
- Problem: arbitrary memory access inside virtual memory

NaCl Constraints on i386

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

NaCl Constraints on i386

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

NaCl on non-i386 Systems

- Other architectures[48] use base register instead of segment offsets
  - Additional verification required

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

NaCl on non-i386 Systems

- Other architectures[48] use base register instead of segment offsets
  - Additional verification required
- Deprecated in 2017 in favor of WebAssembly

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

NaCl on non-i386 Systems

- Other architectures[48] use base register instead of segment offsets
  - Additional verification required
- Deprecated in 2017 in favor of WebAssembly

+ Nice idea, high performance (5–15% overhead)

∼ Instruction blacklist not a good idea

– Not portable, severe restrictions on emitted code

– High verification complexity, error-prone

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

Approach: Using Bytecode

- Idea: compile code to bytecode, JIT-compile on host
  - Benefit:

Approach: Using Bytecode

- Idea: compile code to bytecode, JIT-compile on host
  - Benefit: verification easy – all code generated by trusted compiler
  - Benefit: more portable

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

Approach: Using Bytecode

- Idea: compile code to bytecode, JIT-compile on host
  - Benefit: verification easy – all code generated by trusted compiler
  - Benefit: more portable

- Java applets
- PNaCl: bytecode version of NaCl

+ Fairly high performance, portable

∼ Heavy runtime environment
  - Especially criticized for Java applets

– Very high complexity and attack surface

Approach: Subset of JavaScript: asm.js

- Situation: fairly fast JavaScript JIT-compilers present
- Idea: use subset of JavaScript known to be compilable to efficient code
  - All browsers/JS engines support execution without further changes

Approach: Subset of JavaScript: asm.js

- Situation: fairly fast JavaScript JIT-compilers present
- Idea: use subset of JavaScript known to be compilable to efficient code
  - All browsers/JS engines support execution without further changes

- asm.js[49]: 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)`

[49] D Herman, L Wagner, and A Zakai. *asm.js*. 2014.

asm.js Example

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

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

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

  return +sum;
}
```

Example taken from the specification

Approach: Encode asm.js as Bytecode

- Parsing costs time, type restrictions increase code size
- Idea: encode asm.js source as bytecode

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

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

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

Approach: Encode asm.js as Bytecode

- Parsing costs time, type restrictions increase code size
- Idea: encode asm.js source as bytecode

- First attempt: encode abstract syntax tree in pre-order
- Second attempt: encode abstract syntax tree in post-order
- Third attempt: encode as stack machine

- ... and WebAssembly was born

Approach: Using Bytecode – WebAssembly

- Strictly-typed bytecode format encoding a stack machine
- Global variables and single, global array as memory

Approach: Using Bytecode – WebAssembly

- Strictly-typed bytecode format encoding a stack machine
- Global variables and single, global array as memory
- Functions have local variables
  - Parameters pre-populated in first local variables
  - No dynamic/addressable stack space! $\rightsquigarrow$ part of global memory used as stack
- Operations use implicit stack
  - Stack has well-defined size and types at each point in program

Approach: Using Bytecode – WebAssembly

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

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!

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

Approach: Use Verifiable Bytecode – eBPF

- Problem: want to ensure termination within certain time frame
- Problem: need to make sure *nothing* can go wrong – no sandbox!

- Idea: disallow loops and undefined register values, e.g. due to branch
  - Combinatorial explosion of possible paths, all need to be analyzed
  - No longer Turing-complete
- eBPF: allow user-space to hook into various Linux kernel parts
  - E.g. network, perf sampling, ...
- Strongly verified register machine
- JIT-compiled inside kernel

JIT Compilation and Sandboxing – Summary

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

JIT Compilation and Sandboxing – Questions

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

Code Generation for Data Processing

Lecture 11: Binary Translation

Alexis Engelke

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

Winter 2022/23

Motivation

- Run program on other architecture

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

Motivation

- Run program on other architecture
- Use-case: application compatibility
  - Other architecture with incompatible instruction encoding
  - Applications using unavailable ISA extensions[50]

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

Motivation

- Run program on other architecture
- Use-case: application compatibility
  - Other architecture with incompatible instruction encoding
  - Applications using unavailable ISA extensions[50]
- Use-case: architecture research
  - Development of new ISA extensions without existing hardware

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

ISA Emulation

- Simplest approach: interpreting machine code
  - Simulate individual instructions, don't generate new code

ISA Emulation

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

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

ISA Emulation

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

+ Simple, works almost anywhere, high correctness

– Very inefficient

Binary Translation

- Idea: translate guest machine code to host machine code
- Replace interpretation overhead with translation overhead

- Difficult: very rigid semantics, but few code constraints imposed
  - Self-modifying code, overlapping instructions, indirect jumps
  - Exceptions with well-defined states, status flags

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.

Static vs. Dynamic Binary Translation

Static BT

- Translate guest executable into host executable
- Do translation before execution

Dynamic BT

- Translate code on-the-fly during program execution
- Host code just lives in memory

Static vs. Dynamic Binary Translation

Static BT

- Translate guest executable into host executable
- Do translation before execution

+ Low runtime overhead
– Binaries tend to be huge
– Cannot handle all cases
  - E.g., JIT-compiled code

Dynamic BT

- Translate code on-the-fly during program execution
- Host code just lives in memory

+ Allows for high correctness
∼ Can use JIT optimizations
– Translation overhead at run-time

Static Binary Translation

- Goal: create new binary for host with same functionality

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

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

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

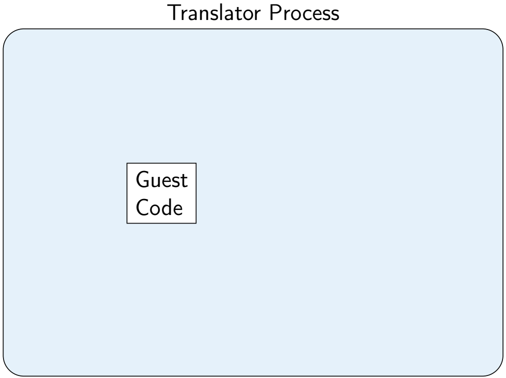
Static Binary Translation

- Goal: create new binary for host with same functionality

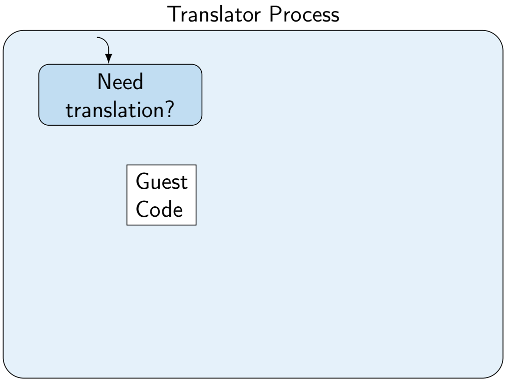
- Program may access its own code/data in various ways
  - Guest binary must be retained as-is in-place
- Indirect jumps problematic
  - Need prediction of all possible targets
  - Keeping lots of dynamically possible entries prohibits optimizations
- JIT-compiled/self-modifying code impossible to handle

- Purely static translation impossible for the general case

Dynamic Binary Translation

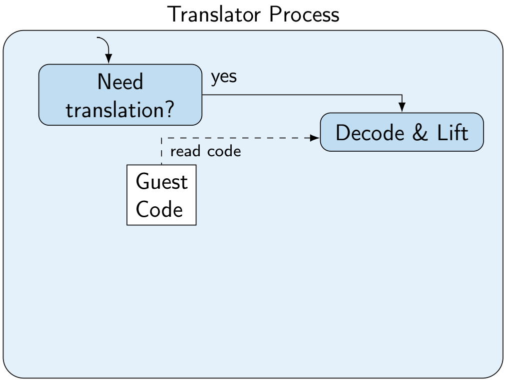


Dynamic Binary Translation



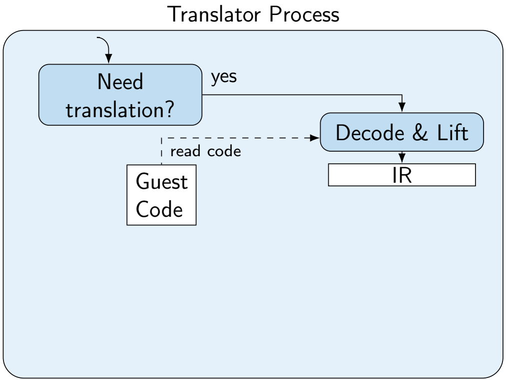
- Iteratively translate code chunks on-demand
  - Typically basic blocks

Dynamic Binary Translation



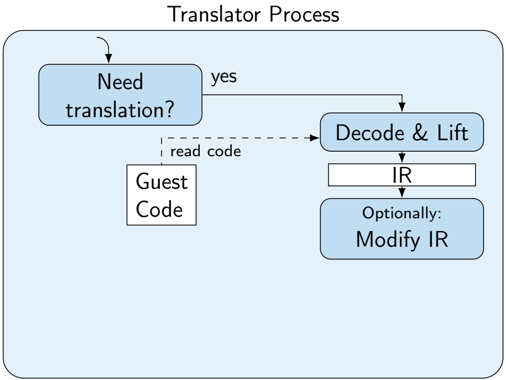
- Iteratively translate code chunks on-demand
  - Typically basic blocks

Dynamic Binary Translation



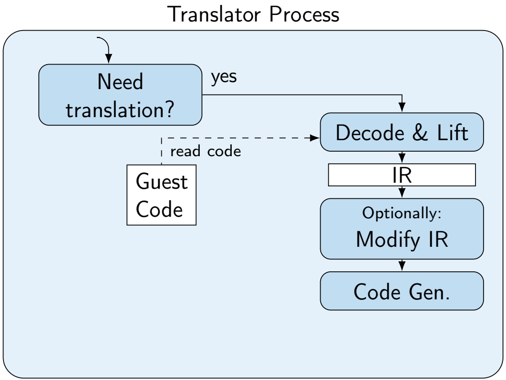
- Iteratively translate code chunks on-demand
  - Typically basic blocks

Dynamic Binary Translation



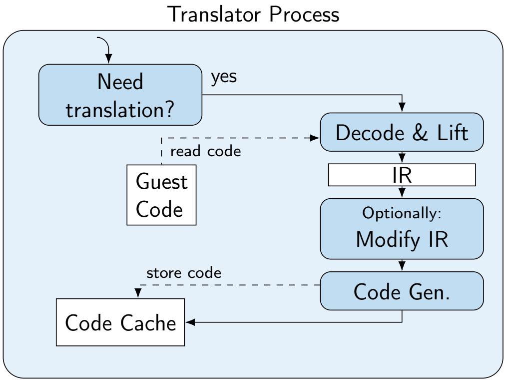
- Iteratively translate code chunks on-demand
  - Typically basic blocks

Dynamic Binary Translation



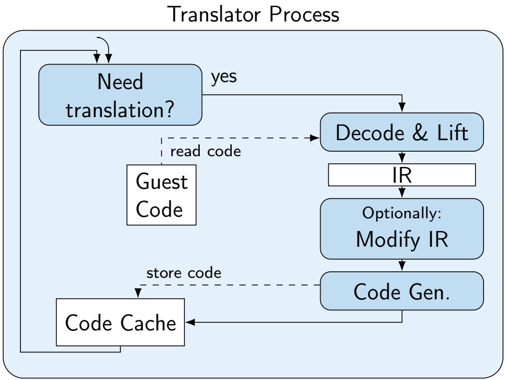
- Iteratively translate code chunks on-demand
  - Typically basic blocks

Dynamic Binary Translation



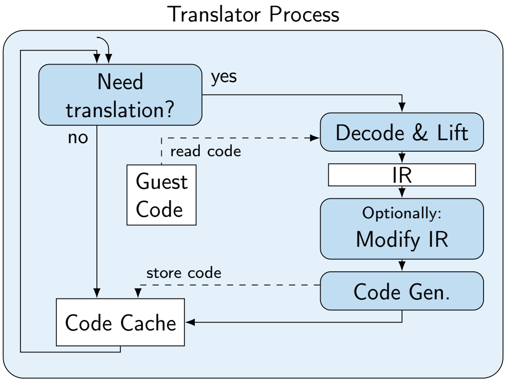
- Iteratively translate code chunks on-demand
  - Typically basic blocks
- Store new code in-memory for execution and later re-use
- Code executed in same address space as original
  - Guest code/data must be accessible

Dynamic Binary Translation



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

Dynamic Binary Translation



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

Dynamic Binary Translation: Code Fragment

RISC-V Code

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

Semantical representation

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

Translation Engine

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

Dynamic Binary Translation: Code Fragment

RISC-V Code

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

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
}
```

Guest State

- Guest CPU state must be completely emulated
  - Registers: general-purpose, floating-point, vector, ...
  - Flags, control registers, system registers, segments, TLS base

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

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

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

Guest State

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

Guest Interface

- User-space emulation: OS interface needs to be emulated
  - Mainly system calls, but also vDSO, memory maps, ...
  - Host libraries are hard to use: ABI differences (e.g. struct padding)
  - Syscall emulation tedious: different flag numbers, arguments, orders
    structs have different fields, alignments, padding bytes

Guest Interface

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

Dynamic Binary Translation: Optimizations

- Fully correct emulation of CPU (and OS) is slow
  - Every memory access is a potential page fault
  - Signals can be delivered at any instruction boundary
  - *many* other traps. . .
- But: these "special" features are used extremely rarely

- Idea: optimize for common case
- Aggressively trade correctness for performance

Translation Granularity

- Larger translation granules allow for more optimization
  - E.g., omit status flag computation; fold immediate construction

Translation Granularity

- Larger translation granules allow for more optimization
  - E.g., omit status flag computation; fold immediate construction

- Instruction: great for debugging
- Basic block: allows for some important opt.
  - Easy to detect (up to next branch), easy to translate (no control flow)
- Superblock: up to next unconditional jump
  - Reduces transfers between blocks in fallthrough case
  - Translated code not necessarily executed
- Function: follow all conditional control flow
  - Allows most optimizations, e.g. for loop induction variables
  - Complex codegen, ind. jumps problematic, lot of code never executed

Chaining

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

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)
```

Chaining: Limitations

Chaining: Limitations

- First execution still slow, patching adds overhead
  - Can speculatively translate continuations
  - Translation of possibly unneeded code adds overhead

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

Chaining: Limitations

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

Return Address Prediction

- Observation: function calls very often return ordinarily
  - Return is an indirect jump, *but* highly predictable
  - But: even for "normal" code, this is not always the case: `setjmp`/`longjmp`, exceptions

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

Return Address Prediction

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

Return Address Prediction in DBT

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

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

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

Return Address Prediction in DBT

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

Status Flags

- Observation: many status flags are rarely used
- But: eager computation can be expensive
  - E.g., x86 parity (PF) or auxiliary carry (AF)

Status Flags

- Observation: many status flags are rarely used
- But: eager computation can be expensive
  - E.g., x86 parity (PF) or auxiliary carry (AF)

- Idea: compute flags only when needed
- On flag computation, store operands needed for flag computation
- Flag usage in same block allows for optimizations
  - E.g., use idiomatic branches (`jle`, ...)
- Flag usage in different block: compute flags from operands
  - More expensive, but happens seldomly

Correct Binary Translation

- Goal 1: precise emulation – application works properly
- Goal 2: stealthness/isolation – application can't compromise DBT

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

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

Correct Binary Translation

- Goal 1: precise emulation – application works properly
- Goal 2: stealthness/isolation – application can't compromise DBT

- Problem: CPU and OS have huge and very-well-specified interfaces
  - ... and even if unspecified, software often depends on it
- Increased difficulty: different guest/host architectures
  - E.g., different page size or memory semantics
- Increased difficulty for user-space: different guest/host OS
  - Depending on syscall interface, nearly impossible (see WSL1)

POSIX Signals

- POSIX specifies signals, which can interrupt program at any point
- Kernel pushes signal frame to stack with user context and calls signal handler
- Signal handler can read/modify user context and continue execution

POSIX Signals

- POSIX specifies signals, which can interrupt program at any point
- Kernel pushes signal frame to stack with user context and calls signal handler
- Signal handler can read/modify user context and continue execution

- Synchronous signals: e.g., SIGSEGV, SIGBUS, SIGFPE, SIGILL
  - For example, due to page fault or FP exception
  - Delivered in response to "error" in current thread

- Asynchronous signals: e.g., SIGINT, SIGTERM, SIGCHILD
  - Delivered externally, e.g. using `kill`
  - Can be delivered to any thread at any time
  - (usually a bad idea to use them)

Correct DBT: Signals

- DBT must register signal handler and propagate signals
- Synchronous signals

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

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
  - $\rightsquigarrow$ Usually delivered when convenient
  - But: real-time signals have special semantics

Correct DBT: Memory Accesses

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

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

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)

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

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

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:

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:

Correct DBT: Memory Ordering

- CPUs (aggressively) reorder memory operations
  - x86: total store ordering – stores can be reordered after loads
  - Most others: weak ordering – everything can be reordered
- Relevant for multi-core systems: other thread can observe ordering
- Atomic operations and fences limit reordering (e.g., acq/rel/seqcst)

- Emulating weak memory on TSO: easy
- Emulating TSO on weak memory: hard
  - Can try to make all operations atomic
  - Atomic operations often need alignment guarantees (not on x86)
  - Only viable solution so far: insert fences everywhere

Correct DBT: Self-modifying Code

- Writable code regions (or with `MAP_SHARED`) can change at any time

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

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!

Correct DBT: Self-modifying Code

- Writable code regions (or with `MAP_SHARED`) can change at any time
- Idea: before translation, remap as read-only
- On page fault (SIGSEGV), remove relevant parts from code cache
  - Requires code cache segmentation and mapping of code to original page
- When executing possibly modifiable code: every store can change code!
- Doesn't easily work for shared memory, need to track this, too
  - Might be impossible when shared with other process

Correct DBT: Floating-point

- Floating-point arithmetic is standardized in IEE-754

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`

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)

Correct DBT: OS and CPU Specifics

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

Binary Translation – Summary

- ISA emulation often used for cross-ISA program execution
- Binary Translation allows for more performance than interpretation
- Static Binary Translation handles whole program ahead-of-time
- Dynamic Binary Translation translates code on-demand
- ISA often highly restricts optimization possibilities
- Optimizations typically very low-level
- Correct emulation of CPU/OS challenging due to large interface

Binary Translation – Questions

- What are use cases of binary translation?
- What is the difference between static and dynamic binary translation?
- Why is static BT strictly less powerful than dynamic BT?
- What are typical translation granularities for DBT?
- How to optimize control flow handling in DBT?
- Why is correct binary translation hard to optimize?
- What problem can occur when not emulating paging for user-space emulation?

Code Generation for Data Processing

Lecture 12: Query Compilation

Alexis Engelke

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

Winter 2022/23

Motivation: Fast Query Execution

Motivation: Fast Query Execution

- Databases are often used in latency-critical situations
  - Mostly transactional workload

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

Motivation: Fast Query Execution

- Databases are often used in latency-critical situations
  - Mostly transactional workload
- Databases are often used for analyzing large data sets
  - Mostly analytical workload; queries can be complex
  - Latency not that important, but through-put is
- Databases are also used for storing data streams
  - Streaming databases, e.g. monitoring sensors
  - Throughput is important; but queries often simple

Data Representation

- Relational algebra: set/bag of tuples
  - Tuple is sequence of data with different types
  - All tuples in one relation have same schema
  - Order does not matter
  - Duplicates might be possible (bags)
- Might have special values, e.g. `NULL`
- Values might be variably-sized, e.g. strings
- But: databases have *high* degree of freedom wrt. data representation

Query Plan

- Query often specified in “standardized format” (SQL)

Query Plan

- Query often specified in “standardized format” (SQL)
- SQL is transformed into (logical) query plan

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

Query Plan

- Query often specified in “standardized format” (SQL)

- SQL is transformed into (logical) query plan
- Logical query plan is optimized
  - E.g., selection push down, transforming cross products to joins, join ordering
- Physical query plan
  - Selection of actual implementation for operators
  - Determine use index structures, access paths, etc.

Query Plan: Subscripts

- Query plan strongly depends on query
- Operators have query-dependent subscripts
  - E.g., selection/join predicate, aggregation function, attributes
  - Implementation of these also depends on schema
- Can include arbitrarily complex expressions
- Examples: $\bowtie^{HJ}_{s.matrnr=h.matrnr}$, $\sigma_{a.x<5\cdot(b.y-a.z)}$

Subscripts: Execution

- Option:

Subscripts: Execution

- Option: keep as tree, interpret

Subscripts: Execution

- Option: keep as tree, interpret
  - \+ Simple, flexible
  - − Slow
- Option:

Subscripts: Execution

- Option: keep as tree, interpret
  - \+ Simple, flexible
  - – Slow
- Option: compile to bytecode

Subscripts: Execution

- Option: keep as tree, interpret
  - \+ Simple, flexible
  - – Slow
- Option: compile to bytecode
  - \+ More efficient
  - – More effort to implement, some compile-time
- Option:

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

Subscripts: Execution

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

SQL Expressions

- Arithmetic expressions are fairly simple
  - Need to respect data type and check for errors (e.g., overflow)
  - Numbers in SQL are (fixed-point) decimals
- String operations can be more complex

SQL Expressions

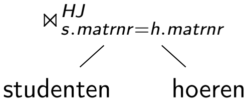
- Arithmetic expressions are fairly simple
  - Need to respect data type and check for errors (e.g., overflow)
  - Numbers in SQL are (fixed-point) decimals
- String operations can be more complex
  - `like` expressions
  - Regular expressions – strongly benefit from optimized execution
  - But: full-compilation may not be worth the effort
    often, calling runtime functions is beneficial
  - Support Unicode for increased complexity

Query Execution: Simplest Approach

$$\bowtie^{HJ}_{s.matrnr=h.matrnr}$$

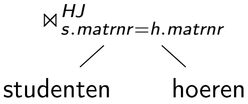
studenten   hoeren

Query Execution: Simplest Approach



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

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

Iterator Model[51]

- 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

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

Iterator Model[51]

- 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

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

Iterator Model: Example

```
struct TableScan : Iter {
  Table* table;
  Table::iterator it;
  void open() { it = table.begin(); }
  Tuple* next() {
    if (it != table.end())
      return *it++;
    return nullptr;
  } };
struct Select : Iter {
  Predicate p;
  Iter base;
  void open() { base.open(); }
  Tuple* next() {
    while (Tuple* t = base.next())
      if (p(t))
        return t;
    return nullptr;
  } };
```

Iterator Model: Example

```
struct TableScan : Iter {
  Table* table;
  Table::iterator it;
  void open() { it = table.begin(); }
  Tuple* next() {
    if (it != table.end())
      return *it++;
    return nullptr;
  } };
struct Select : Iter {
  Predicate p;
  Iter base;
  void open() { base.open(); }
  Tuple* next() {
    while (Tuple* t = base.next())
      if (p(t))
        return t;
    return nullptr;
  } };
```

```
struct Cross : Iter {
  Iter left, right;
  Tuple* curLeft = nullptr;
  void open() { left.open(); }
  Tuple* next() {
    while (true) {
      if (!curLeft) {
        if (!(curLeft = left.next()))
          return nullptr;
        right.open();
      }
      if (Tuple* tr = right.next())
        return concat(curLeft, tr);
      curLeft = nullptr;
    }
  }
};
```

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

Iterator Model

- "Pull-based" approach
- Widely used (e.g., Postgres)
- Often have separate function for `first()` or rewind

Iterator Model

- "Pull-based" approach
- Widely used (e.g., Postgres)
- Often have separate function for `first()` or rewind

\+ Fairly straight-forward to implement

\+ Avoids data copies, no dynamic compilation

– Only single tuple processed at a time, bad locality

– *Huge* amount virtual function calls

Push-based Model[52]

- 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

[52] T Neumann. "Efficiently compiling efficient query plans for modern hardware". In: *VLDB* 4.9 (2011), pp. 539–550.

Push-based Model: Example

```
struct TableScan {
  Table table;
  Consumer cons;
  void produce() {
    for (Tuple* t : table)
      cons.consume(t, this);
  }
};
struct Select {
  Predicate p;
  Producer prod;
  Consumer cons;
  void produce() { prod.produce(); }
  void consume(Tuple* t, Producer src) {
    if (p(t))
      cons.consume(t)
  }
};
```

Push-based Model: Example

```
struct TableScan {
  Table table;
  Consumer cons;
  void produce() {
    for (Tuple* t : table)
      cons.consume(t, this);
  }
};
struct Select {
  Predicate p;
  Producer prod;
  Consumer cons;
  void produce() { prod.produce(); }
  void consume(Tuple* t, Producer src) {
    if (p(t))
      cons.consume(t)
  }
};
```

```
struct Cross {
  Producer left, right;
  Consumer cons;
  Tuple* curLeft = nullptr;
  void produce() { left.produce(); }
  // Materializing one side might be better
  void consume(Tuple* t, Producer src) {
    if (src == left) {
      curLeft = t;
      right.produce();
    } else { // src == right
      cons.consume(concat(curLeft, t));
    }
  }
};
```

Push-based Model

- “Push-based” approach
- More recent approach

Push-based Model

- "Push-based" approach
- More recent approach

+ Fairly straight-forward, but less intuitive than iterator
+ Avoids data copies, no dynamic compilation
– Only single tuple processed at a time, bad locality
– *Huge* amount virtual function calls

Pull-based Model vs. Push-based Model[53]

- 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:

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

Pull-based Model vs. Push-based Model[53]

- 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:

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

Pull-based Model vs. Push-based Model[53]

- 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

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

Pipelining

- Some operators need materialized data for their operation
  - Pipeline breaker: operator materializes input
  - Full pipeline breaker: operator materializes complete input before producing
- Other operators can be *pipelined* (i.e., no materialization)

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)

Pipelining

- Some operators need materialized data for their operation
  - Pipeline breaker: operator materializes input
  - Full pipeline breaker: operator materializes complete input before producing
- Other operators can be *pipelined* (i.e., no materialization)

- Aggregations
- Join needs one side materialized (pipeline breaker on one side)
- Sorting needs all data (full pipeline breaker)

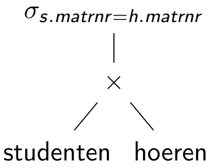
- System needs to take care of semantics, e.g. for memory management

Code Generation for Push-Based Model

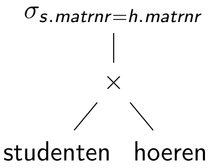
- Inlining code in push-based model yields nice code

- No virtual function calls
- Producer iterates over materialized tuples and loads relevant data
  - Tight loop over base table – data locality
- Operators of parent operators are applied inside the loop
- Pipeline breaker materializes result (e.g., into hash table)

Code Generation: Example



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);
      }
    }
  }
};
```

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

How to Generate Code

- Code generator executes produce/consume methods
  - Method bodies don't do actual operations, but construct code
  - E.g., call `IRBuilder`
  - Call to helper functions for complex operations
    e.g. hash table insert/lookup, string operations, memory allocation, etc.
- Resulting code doesn't contain produce/consume methods
  only loops that iterate over data
  - No overhead of function calls
- Generate (at most) one function per pipeline
  - Allows for parallel execution of different pipelines

What to Generate

- Code generation allows for substantial performance increase
  - *Fairly* popular, even in commercial systems, despite engineering effort
  - Competence in compiler engineering is a problem, though

What to Generate

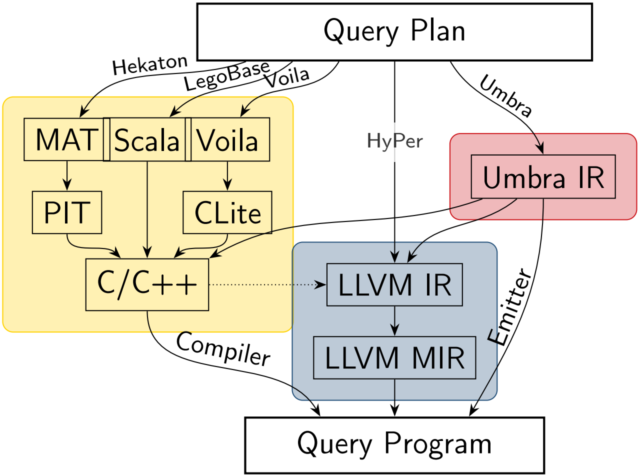
- Code generation allows for substantial performance increase
  - *Fairly* popular, even in commercial systems, despite engineering effort
  - Competence in compiler engineering is a problem, though
- Bytecode
  - Extremely popular: fairly simple, portable, and flexible

What to Generate

- Code generation allows for substantial performance increase
  - *Fairly* popular, even in commercial systems, despite engineering effort
  - Competence in compiler engineering is a problem, though

- Bytecode
  - Extremely popular: fairly simple, portable, and flexible
- Machine code through programming language (C, C++, Scala, . . .)
  - Also popular: no compiler knowledge required, but compile-times are bad
- Machine code through compiler IR (mostly LLVM)
- Machine code through specialized IR (Umbra only)

What to Generate



Case Study: Amazon Redshift[54]

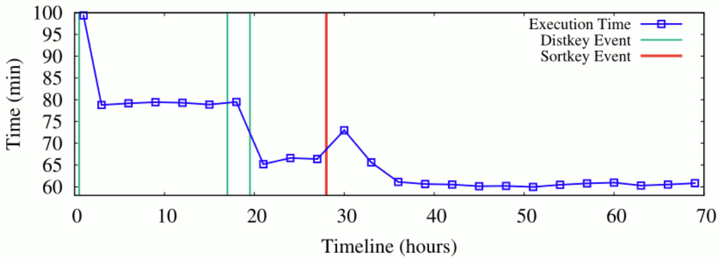
[54] N Armenatzoglou et al. "Amazon Redshift Re-invented". In: *SIGMOD*. 2022.

Case Study: Amazon Redshift[54]

"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."

[54] N Armenatzoglou et al. "Amazon Redshift Re-invented". In: *SIGMOD*. 2022.

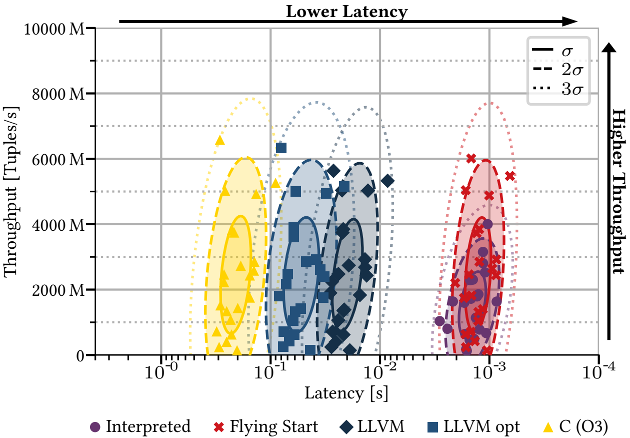
Case Study: Amazon Redshift[55]



"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).

[55] N Armenatzoglou et al. "Amazon Redshift Re-invented". In: *SIGMOD*. 2022.

Compile Times: Umbra



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

Vectorized Execution

- Problem: still only process single tuple at a time

Vectorized Execution

- Problem: still only process single tuple at a time
- Doesn't utilize vector extensions of CPUs

- Idea: process multiple tuples at once
  - Also allows eliminating data-dependent branches, which not well-predictable
  - Esp. relevant when selectivity is between 10–90%

- Use of SIMD instructions requires column-wise store
  - Row-wise store would require gather operation for each load
  - Gather is very expensive

Vectorized Execution: SIMD Instructions

- Obvious candidate:

Vectorized Execution: SIMD Instructions

- Obvious candidate: initial selection over tables
  - Load vector of elements, use SIMD operations for comparison
  - Write back compressed result to temporary location for use in subsequent operations
  - Special compress instructions (AVX-512, SVE) highly beneficial
- Other operations much more difficult to vectorize
  - Initial hash table lookup requires gather; collisions difficult
  - When many elements are masked out, performance suffers

Vectorized Execution

- Bytecode interpretation substantially benefits from vectorized execution
- Key benefit: less dispatch overhead
- Typically much larger “vectors” (>1000)

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

Query Compilation – Summary

- Databases have trade-off between low latency and high throughput
- Evaluation needed for operators and subscripts
- Subscripts easy to compile
- Operator execution: full materialization vs. pipelined execution
- Pull-based vs. push-based execution
- Push-based allows for good code generation
- Bytecode and programming languages are widely used in practice
- Vectorized execution improves performance without native code gen.

Query Compilation – Questions

- Why are low compile times important for databases?
- What is the difference between push-based and pull-based execution?
- Why does push-based execution allow for higher performance?
- How to generate code for a query?
- How does vectorized execution improve performance?
- Why do many database engines not use machine code generation?

Code Generation for Data Processing

Lecture 13: Vectorization

Alexis Engelke

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

Winter 2022/23

Parallel Data Processing

- Sequential execution has inherently limited performance
  - Clock rate, data path lengths, speed of light, ...
- Parallelism is the key to substantial and scalable perf. improvements
- Modern systems have many levels of parallelism:

Parallel Data Processing

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

Single Instruction, Multiple Data (SIMD)

- Idea: perform same operations on multiple data in parallel
- First computer with SIMD operations:

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[56]
- Wider use in HPC in 1970s with vector processors (Cray et al.)
  - Ultimately replaced by much more scalable distributed machines

[56] W Clark et al. *The Lincoln TX-2 Computer*. Apr. 1957. .

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[56]
- 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

[56] W Clark et al. *The Lincoln TX-2 Computer*. Apr. 1957. 🌐.

SIMD: Idea

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

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

- Horizontal operations work on neighbored elements

SIMD ISAs: Design

- Vectors are often implemented as fixed-size wide registers
  - Examples: ARM NEON 32×128-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

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

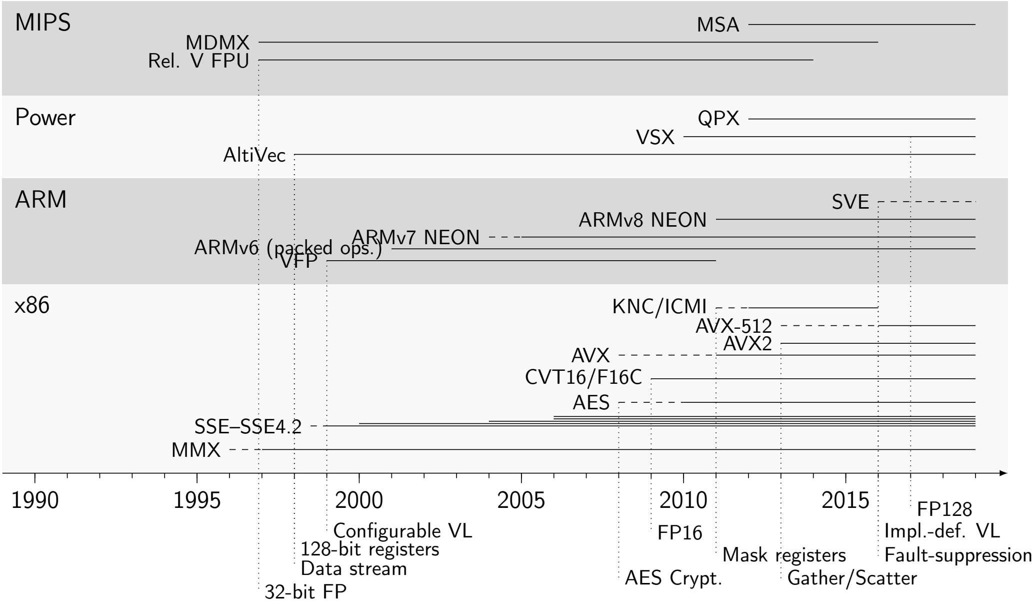
- Data types vary, e.g. i8/i16/i32/i64/f16/bf16/f32/f64/f128
  - Sometimes only conversion, sometime with saturating arithmetic

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


MIPS
MSA
MDMX
Rel. V FPU
Power
QPX
VSX
AltiVec
ARM
SVE
ARMv8 NEON
ARMv7 NEON
ARMv6 (packed ops.)
VFP
x86
KNC/ICMI
AVX-512
AVX2
AVX
CVT16/F16C
AES
SSE–SSE4.2
MMX
1990
1995
2000
2005
2010
2015
FP128
Impl.-def. VL
Fault-suppression
Configurable VL
FP16
Mask registers
128-bit registers
Data stream
AES Crypt.
Gather/Scatter
32-bit FP

SIMD: Use Cases

- Dense linear algebra: vector/matrix operations
  - Implementations: Intel MKL, OpenBLAS, ATLAS, . . .

SIMD: Use Cases

- Dense linear algebra: vector/matrix operations
  - Implementations: Intel MKL, OpenBLAS, ATLAS, ...
- Sparse linear algebra
  - Needs gather/scatter instructions

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

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

SIMD: Use Cases

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

SIMD ISAs: Usage Considerations

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)

SIMD ISAs: Usage Considerations

- Very easy to implement in hardware
  - Simple replication of functional units and larger vector registers
  - Too large vectors, however, also cause problems (AVX-512)
- Offer significant speedups for certain applications
  - With 4x parallelism, speed-ups of $\sim$3x are achievable
- Caveat:

SIMD ISAs: Usage Considerations

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

SIMD Programming: (Inline) Assembly

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

- "Popular" for optimized libraries

SIMD Programming: (Inline) Assembly

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

- "Popular" for optimized libraries

+ Allows for best performance
– Very tedious to write, manual register allocation, non-portable
– No optimization across boundaries

SIMD Programming: Intrinsics

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

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

– Tedious to write, non-portable

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:

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

Fault Suppression

- Variable-length vectors are problematic for buffers of unknown size
  - Example: NUL-terminated C strings
- Classical approach:

Fault Suppression

- Variable-length vectors are problematic for buffers of unknown size
  - Example: NUL-terminated C strings

- Classical approach: ensure alignment to prevent page faults
  - These types of out-of-bounds reads are guaranteed to be non-harmful
  - Downside:

Fault Suppression

- Variable-length vectors are problematic for buffers of unknown size
  - Example: NUL-terminated C strings

- Classical approach: ensure alignment to prevent page faults
  - These types of out-of-bounds reads are guaranteed to be non-harmful
  - Downside: needs loop peeling code for start/end
- More recent approach:

Fault Suppression

- Variable-length vectors are problematic for buffers of unknown size
  - Example: NUL-terminated C strings
- Classical approach: ensure alignment to prevent page faults
  - These types of out-of-bounds reads are guaranteed to be non-harmful
  - Downside: needs loop peeling code for start/end
- More recent approach: make hardware suppress exceptions
  - Option 1: specify that masked out lanes do not produce faults
  - Option 2: stop loading after first fault, store as mask in register
  - Downside 1:

Fault Suppression

- Variable-length vectors are problematic for buffers of unknown size
  - Example: NUL-terminated C strings
- Classical approach: ensure alignment to prevent page faults
  - These types of out-of-bounds reads are guaranteed to be non-harmful
  - Downside: needs loop peeling code for start/end
- More recent approach: make hardware suppress exceptions
  - Option 1: specify that masked out lanes do not produce faults
  - Option 2: stop loading after first fault, store as mask in register
  - Downside 1: increased complexity in hardware, may use microcode
  - Downside 2: permits speculative vectorization at cost of more instructions

SIMD Programming: Target-independent Vector Extensions

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

GCC Vector Extensions

```c
#include <stdint.h>

typedef uint32_t uint32x4_t
    __attribute__((vector_size(16)));

uint32x4_t
addvec(uint32x4_t a, uint32x4_t b) {
    return a + b;
}

uint32x4_t
modvec(uint32x4_t a, uint32x4_t b) {
    return a % b;
}
```

GCC Vector Extensions

```
#include <stdint.h>

typedef uint32_t uint32x4_t
    __attribute__((vector_size(16)));

uint32x4_t
addvec(uint32x4_t a, uint32x4_t b) {
    return a + b;
}

uint32x4_t
modvec(uint32x4_t a, uint32x4_t b) {
    return a % b;
}
```

```
addvec:
  paddd xmm0, xmm1
  ret
```

GCC Vector Extensions

```c
#include <stdint.h>

typedef uint32_t uint32x4_t
    __attribute__((vector_size(16)));

uint32x4_t
addvec(uint32x4_t a, uint32x4_t b) {
    return a + b;
}

uint32x4_t
modvec(uint32x4_t a, uint32x4_t b) {
    return a % b;
}
```

```asm
addvec:
  paddd xmm0, xmm1
  ret
```

```asm
modvec:
  movd ecx, xmm1
  movd eax, xmm0
  xor edx, edx
  pextrd edi, xmm1, 1
  div ecx
  pextrd eax, xmm0, 1
  pextrd ecx, xmm1, 2
  mov esi, edx
  xor edx, edx
  div edi
  pextrd eax, xmm0, 2
  mov r8d, edx
  xor edx, edx
  div ecx
  pextrd ecx, xmm1, 3
  pextrd eax, xmm0, 3
  movd xmm0, esi
  pinsrd xmm0, r8d, 1
  mov edi, edx
  xor edx, edx
  div ecx
  movd xmm1, edi
  pinsrd xmm1, edx, 1
```

SIMD Programming: Single Program, Multiple Data (SPMD)

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

SPMD: Example (OpenMP)

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

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

_ZGVxN4vv_foo:
  paddd xmm0, xmm1
  ret
```

SPMD: Example (OpenMP)

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

- Compiler generates version that operates on vector

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

_ZGVxN4vv_foo:
  paddd xmm0, xmm1
  ret
```

SPMD: Example (OpenMP)

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

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

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

SPMD: Example (OpenMP)

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

- Uniform: always same value

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

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

SPMD: Example (OpenMP) – `if/else`

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

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

SPMD: Example (OpenMP) – if/else

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

```asm
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
```

SPMD: Example (OpenMP) – if/else

```c
#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

```asm
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
```

SPMD to SIMD: Handling `if/else`

- Control flow solely depending on uniforms:

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

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[57]

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

SPMD Implementations on CPUs

- OpenMP SIMD functions
  - Need to be combined with `#pragma omp simd` loops
- Intel ispc[58] (Implicit SPMD Program Compiler)
  - Extension of C with keywords `uniform`, `varying`
  - Still active and interesting history[59]
- OpenCL on CPU
  - Very similar programming model
  - But: higher complexity for communicating with rest of application

[58] M Pharr and WR Mark. "ispc: A SPMD compiler for high-performance CPU programming". In: *InPar*. 2012, pp. 1–13.

[59] https://pharr.org/matt/blog/2018/04/30/ispc-all

SIMD Programming: SPMD on CPUs

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

SIMD Programming: SPMD on CPUs

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

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

SIMD Programming: Auto-vectorization

- Idea: programmer is too incompetent/busy, let compiler do vectorization
- Inherently difficult and problematic, after decades of research

SIMD Programming: Auto-vectorization

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

Auto-vectorization is Hard

- Biggest problem: data dependencies
  - Resolving loop-carried dependencies is difficult
- Memory aliasing
  - Overlapping arrays, or – worse – loop counter
- 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)

Auto-vectorization is Hard

- Biggest problem: data dependencies
  - Resolving loop-carried dependencies is difficult
- Memory aliasing
  - Overlapping arrays, or – worse – loop counter
- Loop body *might* impact loop count
- Function calls, e.g. for math functions
- Strided memory access (e.g., only every n-th element)
- Choosing vectorization level (outer loop *might* be better)

- Is vectorization profitable *at all*?
- Often black box to programmer, preventing fine-grained tuning

Auto-vectorization Strategies

- Inner Loop Vectorization: unroll innermost loop n times
  - Try to compact loop body into vectors with n lanes

Auto-vectorization Strategies

- Inner Loop Vectorization: unroll innermost loop n times
  - Try to compact loop body into vectors with n lanes
- Outer Loop Vectorization: unroll outer loop n times
  - Try to compact loop body into vectors with n lanes
  - Generally does not support diverging control flow in loop body

Auto-vectorization Strategies

- Inner Loop Vectorization: unroll innermost loop n times
  - Try to compact loop body into vectors with n lanes
- Outer Loop Vectorization: unroll outer loop n times
  - Try to compact loop body into vectors with n lanes
  - Generally does not support diverging control flow in loop body
- Superword-level Parallelism (SLP): packing series of scalar stores
  - Detect neighbored stores, try to fold operations into vectors

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