CS415 Compilers
ILOC, Code Shape, and Instruction Scheduling

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

• Recitations and office hours start this week
• Office hours will be posted soon
• Please go to https://www.cs.rutgers.edu/courses/415/classes/spring_2022_kremer/ to download lecture slides
• Lecture videos for first three lectures are/will be available on canvas https://rutgers.instructure.com/courses/160913
• Please go to piazza for questions https://rutgers.instructure.com/courses/160913/external_tools/1590
• Reminder: Get ilab account
Implications

• Use an intermediate representation (IR)
• Front end maps legal source code into IR
• Back end maps IR into target machine code

Typically, front end is $O(n)$ or $O(n \log n)$, while back end is NP-complete
Part of the compiler’s back end

Critical properties
• Produce correct code that uses \( k \) (or fewer) registers
• Minimize added loads and stores
• Minimize space used to hold spilled values
• Operate efficiently
  \( O(n), O(n \log_2 n), \) maybe \( O(n^2) \), but not \( O(2^n) \)
Part of the compiler’s back end

Backend - Instruction Scheduling

Instruction Selection → Register Allocation → Instruction Scheduling → Machine code

$\text{IR} \rightarrow m \text{ register} \rightarrow k \text{ register} \rightarrow \text{Machine code}$
Local Instruction Scheduling

Readings: EaC 12.1-12.3, Appendix A (ILOC)

Definition

A **basic block** is a maximal length segment of straight-line (i.e., branch free) code. Control can only enter at first instruction of basic block and exit after last instruction.

*Local*: within single basic block  
*Global*: across procedures/functions
**Instruction Scheduling**

**Motivation**
- Instruction latency (pipelining)
  several cycles to complete instructions; instructions can be issued every cycle
- Instruction-level parallelism (VLIW, superscalar)
  execute multiple instructions per cycle

**Issues**
- Reorder instructions to reduce execution time
- Static schedule - insert NOPs to preserve correctness
- Dynamic schedule - hardware pipeline stalls
- Preserve correctness, improve performance
- Interactions with other optimizations (register allocation!)
Motivation

- Instruction latency (pipelining)
  several cycles to complete instructions; instructions can be issued every cycle
- Instruction-level parallelism (VLIW, superscalar)
  execute multiple instructions per cycle

Issues

- Reorder instructions to reduce execution time
- Static schedule - insert NOPs to preserve correctness
- Dynamic schedule - hardware pipeline stalls
- Preserve correctness, improve performance
- Interactions with other optimizations (register allocation!)
- Note: After register allocation, code shape contains real, not virtual registers ==> register may be redefined
Source code

\[
A = 5; \\
B = 6; \\
C = A + B;
\]
Source code

A = 5;
B = 6;
C = A + B;

Assume A, B, C are integer values of 4 bytes
address(A) = 1024 + offset(A) = 1028
address(B) = 1024 + offset(B) = 1032
address(C) = 1024 + offset(C) = 1036

This convention is used in activation records or stack frames. We use it here for consistency.

More general:
address(X) = base_address + offset(X)
Instruction scheduling on basic blocks in “ILOC”

- Pseudo-code for a simple, abstracted RISC machine generated by the instruction selection process
- Simple, compact data structures
- Here: we only use a small subset of ILOC

Naïve Representation:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadl</td>
<td>2</td>
<td></td>
<td>r1</td>
</tr>
<tr>
<td>loadAl</td>
<td>r0</td>
<td>@y</td>
<td>r2</td>
</tr>
<tr>
<td>add</td>
<td>r1</td>
<td>r2</td>
<td>r3</td>
</tr>
<tr>
<td>loadAl</td>
<td>r0</td>
<td>@x</td>
<td>r4</td>
</tr>
<tr>
<td>sub</td>
<td>r4</td>
<td>r3</td>
<td>r5</td>
</tr>
</tbody>
</table>

Quadruples:

- table of $k \times 4$ small integers
- simple record structure
- easy to reorder
- all names are explicit

ILOC is described in Appendix A of EAC.

ILOC simulator “sim” is available on ilab:
~uli/cs415/ILOC_Simulator/sim
Memory Model / Code Shape

ILOC: EaC Appendix A

<table>
<thead>
<tr>
<th>Source code</th>
<th>ILOC code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 5;</td>
<td>loadI 5 (\Rightarrow r1)</td>
</tr>
<tr>
<td>B = 6;</td>
<td>// compute address of A in r2</td>
</tr>
<tr>
<td>C = A + B;</td>
<td>store r1 (\Rightarrow r2) // content(A) = r1</td>
</tr>
<tr>
<td></td>
<td>loadI 6 (\Rightarrow r3)</td>
</tr>
<tr>
<td></td>
<td>// compute address of B in r4</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>store r3 (\Rightarrow r4) // content(B) = r3</td>
</tr>
<tr>
<td></td>
<td>add r1, r3 (\Rightarrow r5)</td>
</tr>
<tr>
<td></td>
<td>// compute address of C in r6</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>store r5 (\Rightarrow r6) // content(C) = r1 + r3</td>
</tr>
</tbody>
</table>

Is this code correct?
**Memory Model / Code Shape**

**Source code**

```plaintext
foo (var A, B)
A = 5;
B = 6;
C = A + B;
end foo;

X = 1
call foo(X,X);
print C;
```

**ILOC code**

```plaintext
loadI 5 \rightarrow r1
// compute address of A in r2
store r1 \rightarrow r2  // content(A) = r1
loadI 6 \rightarrow r3
// compute address of B in r4
store r3 \rightarrow r4  // content(B) = r3
add r1, r3 \rightarrow r5
// compute address of C in r6
store r5 \rightarrow r6  // content(C) = r1 + r3
```

**Memory layout**

```
0
.
.
1024
.

byte data addresses
```

**Is this code correct?**

*Incorrect for call-by-reference!*

---

*cs415, spring 22*  
*Lecture 2*
Aliasing: Two variables or source-code names may refer to the same memory location.

Examples:
- formal call-by-reference parameters a and b
- pointers a->f and b->f
- array elements: a(i, j) and a(k, l)

Challenge: When is it safe to keep a variable’s value in a register across STORE instructions, i.e., while other STORE instructions are executed?
Memory Model / Code Shape

- **register-register model**
  - Values that may safely reside in registers are assigned to a unique virtual register (alias analysis)
  - Register allocation/assignment maps virtual registers to limited set of physical registers
  - Register allocation/assignment pass needed to make code “work”

- **memory-memory model**
  - All values reside in memory, and are only kept in registers as briefly as possible (load operands from memory, perform computation, store result into memory)
  - Register allocation/assignment has to try to identify cases where values can be safely kept in registers
  - Safety verification is hard at the low levels of program abstraction
  - Even without register allocation/assignment, code will “work”
More instruction scheduling
EaC 12.1 – 12.3