Lecture 30

Code Generation

Awanish Pandey

Department of Computer Science and Engineering
Indian Institute of Technology
Roorkee

April 11, 2025

Rearranging order of the code

- `a+e*(c+d)`

Rearranging order of the code

- `a+e*(c+d)`
- Minimum registers required for correct execution?

Rearranging order of the code

- `a+e*(c+d)`
- Minimum registers required for correct execution?

1. $R_0 = a$

Rearranging order of the code

- `a+e*(c+d)`
- Minimum registers required for correct execution?

1. $R_0 = a$
2. $R_1 = c$

Rearranging order of the code

- `a+e*(c+d)`
- Minimum registers required for correct execution?

1. $R_0 = a$
2. $R_1 = c$
3. $R_2 = d$

Rearranging order of the code

- `a+e*(c+d)`
- Minimum registers required for correct execution?

1. $R_0 = a$
2. $R_1 = c$
3. $R_2 = d$
4. $R_1 = R_1 + R_2$

Rearranging order of the code

- `a+e*(c+d)`
- Minimum registers required for correct execution?

1. $R_0 = a$
2. $R_1 = c$
3. $R_2 = d$
4. $R_1 = R_1 + R_2$
5. $R_2 = e$

Rearranging order of the code

- `a+e*(c+d)`
- Minimum registers required for correct execution?

1. $R_0 = a$
2. $R_1 = c$
3. $R_2 = d$
4. $R_1 = R_1 + R_2$
5. $R_2 = e$
6. $R_2 = R_2 * R_1$

Rearranging order of the code

- `a+e*(c+d)`
- Minimum registers required for correct execution?

1. $R_0 = a$
2. $R_1 = c$
3. $R_2 = d$
4. $R_1 = R_1 + R_2$
5. $R_2 = e$
6. $R_2 = R_2 * R_1$
7. $R_0 = R_0 + R_2$

Rearranging order of the code

- `a+e*(c+d)`
- Minimum registers required for correct execution?

1. $R_0 = a$
2. $R_1 = c$
3. $R_2 = d$
4. $R_1 = R_1 + R_2$
5. $R_2 = e$
6. $R_2 = R_2 * R_1$
7. $R_0 = R_0 + R_2$

1. $R_0 = c$

Rearranging order of the code

- `a+e*(c+d)`
- Minimum registers required for correct execution?

1. $R_0 = a$
2. $R_1 = c$
3. $R_2 = d$
4. $R_1 = R_1 + R_2$
5. $R_2 = e$
6. $R_2 = R_2 * R_1$
7. $R_0 = R_0 + R_2$

1. $R_0 = c$
2. $R_1 = d$

Rearranging order of the code

- `a+e*(c+d)`
- Minimum registers required for correct execution?

1. $R_0 = a$
2. $R_1 = c$
3. $R_2 = d$
4. $R_1 = R_1 + R_2$
5. $R_2 = e$
6. $R_2 = R_2 * R_1$
7. $R_0 = R_0 + R_2$

1. $R_0 = c$
2. $R_1 = d$
3. $R_0 = R_0 + R_1$

Rearranging order of the code

- `a+e*(c+d)`
- Minimum registers required for correct execution?

1. $R_0 = a$
2. $R_1 = c$
3. $R_2 = d$
4. $R_1 = R_1 + R_2$
5. $R_2 = e$
6. $R_2 = R_2 * R_1$
7. $R_0 = R_0 + R_2$

1. $R_0 = c$
2. $R_1 = d$
3. $R_0 = R_0 + R_1$
4. $R_1 = e$

Rearranging order of the code

- `a+e*(c+d)`
- Minimum registers required for correct execution?

1. $R_0 = a$
2. $R_1 = c$
3. $R_2 = d$
4. $R_1 = R_1 + R_2$
5. $R_2 = e$
6. $R_2 = R_2 * R_1$
7. $R_0 = R_0 + R_2$

1. $R_0 = c$
2. $R_1 = d$
3. $R_0 = R_0 + R_1$
4. $R_1 = e$
5. $R_1 = R_1 * R_0$

Rearranging order of the code

- `a+e*(c+d)`
- Minimum registers required for correct execution?

1. $R_0 = a$
2. $R_1 = c$
3. $R_2 = d$
4. $R_1 = R_1 + R_2$
5. $R_2 = e$
6. $R_2 = R_2 * R_1$
7. $R_0 = R_0 + R_2$

1. $R_0 = c$
2. $R_1 = d$
3. $R_0 = R_0 + R_1$
4. $R_1 = e$
5. $R_1 = R_1 * R_0$
6. $R_0 = a$

Rearranging order of the code

- `a+e*(c+d)`
- Minimum registers required for correct execution?

1. $R_0 = a$
2. $R_1 = c$
3. $R_2 = d$
4. $R_1 = R_1 + R_2$
5. $R_2 = e$
6. $R_2 = R_2 * R_1$
7. $R_0 = R_0 + R_2$

1. $R_0 = c$
2. $R_1 = d$
3. $R_0 = R_0 + R_1$
4. $R_1 = e$
5. $R_1 = R_1 * R_0$
6. $R_0 = a$
7. $R_0 = R_0 + R_1$

Register allocation

- When should we do?

Register allocation

- When should we do?
  - Instruction Selection

Register allocation

- When should we do?
  - Instruction Selection
  - Instruction Reordering

Register allocation

- When should we do?
  - Instruction Selection
  - Instruction Reordering
- Type of allocator

Register allocation

- When should we do?
  - Instruction Selection
  - Instruction Reordering
- Type of allocator
  - Using usages count

Register allocation

- When should we do?
  - Instruction Selection
  - Instruction Reordering
- Type of allocator
  - Using usages count
  - Graph coloring based (Near optimal/ Build interferance graph)

Register allocation

- When should we do?
  - Instruction Selection
  - Instruction Reordering
- Type of allocator
  - Using usages count
  - Graph coloring based (Near optimal/ Build interferance graph)
  - linear scan (Fast)

Register allocation

- When should we do?
  - Instruction Selection
  - Instruction Reordering
- Type of allocator
  - Using usages count
  - Graph coloring based (Near optimal/ Build interferance graph)
  - linear scan (Fast)
  - Local register allocation

Problem in Simple Code Generator

Problem in Simple Code Generator

- Visibility of only current instruction.

Problem in Simple Code Generator

- Visibility of only current instruction.
- DAG of the basic block will give more visibility

Problem in Simple Code Generator

- Visibility of only current instruction.
- DAG of the basic block will give more visibility
    - It stores data dependencies

Problem in Simple Code Generator

- Visibility of only current instruction.
- DAG of the basic block will give more visibility
    - It stores data dependencies
    - Helps in common sub-expression elimination

Problem in Simple Code Generator

- Visibility of only current instruction.
- DAG of the basic block will give more visibility
  - It stores data dependencies
  - Helps in common sub-expression elimination
  - Useful in identifying code generation order

Problem in Simple Code Generator

- Visibility of only current instruction.
- DAG of the basic block will give more visibility
  - It stores data dependencies
  - Helps in common sub-expression elimination
  - Useful in identifying code generation order
  - Useful in instruction reordering

Problem in Simple Code Generator

- Visibility of only current instruction.
- DAG of the basic block will give more visibility
  - It stores data dependencies
  - Helps in common sub-expression elimination
  - Useful in identifying code generation order
  - Useful in instruction reordering
- Code generation based on DAG

Problem in Simple Code Generator

- Visibility of only current instruction.
- DAG of the basic block will give more visibility
  - It stores data dependencies
  - Helps in common sub-expression elimination
  - Useful in identifying code generation order
  - Useful in instruction reordering
- Code generation based on DAG
  - Perform topological numbering for the DAG nodes

Problem in Simple Code Generator

- Visibility of only current instruction.
- DAG of the basic block will give more visibility
    - It stores data dependencies
    - Helps in common sub-expression elimination
    - Useful in identifying code generation order
    - Useful in instruction reordering
- Code generation based on DAG
    - Perform topological numbering for the DAG nodes
    - Generate code for nodes in reverse topological ordering

Problem in Simple Code Generator

- Visibility of only current instruction.
- DAG of the basic block will give more visibility
    - It stores data dependencies
    - Helps in common sub-expression elimination
    - Useful in identifying code generation order
    - Useful in instruction reordering
- Code generation based on DAG
    - Perform topological numbering for the DAG nodes
    - Generate code for nodes in reverse topological ordering
    - Optimal code generation is NP-hard

Optimal Code Generation in case of tree (assuming sufficient registers)

Optimal Code Generation in case of tree (assuming sufficient registers)

- Ershov Numbers: Numbers of registers needed to evaluate a node without storing any temporaries.

Optimal Code Generation in case of tree (assuming sufficient registers)

- Ershov Numbers: Numbers of registers needed to evaluate a node without storing any temporaries.
  - Label all leaves as 1.

Optimal Code Generation in case of tree (assuming sufficient registers)

- Ershov Numbers: Numbers of registers needed to evaluate a node without storing any temporaries.
    - Label all leaves as 1.
    - The label of an interior node with one child is the label of its child

Optimal Code Generation in case of tree (assuming sufficient registers)

- Ershov Numbers: Numbers of registers needed to evaluate a node without storing any temporaries.
    - Label all leaves as 1.
    - The label of an interior node with one child is the label of its child
    - The label of an interior node with two children

Optimal Code Generation in case of tree (assuming sufficient registers)

- Ershov Numbers: Numbers of registers needed to evaluate a node without storing any temporaries.
  - Label all leaves as 1.
  - The label of an interior node with one child is the label of its child
  - The label of an interior node with two children
    - Larger of the label of its children if those labels are different.

Optimal Code Generation in case of tree (assuming sufficient registers)

- Ershov Numbers: Numbers of registers needed to evaluate a node without storing any temporaries.
  - Label all leaves as 1.
  - The label of an interior node with one child is the label of its child
  - The label of an interior node with two children
    - ★ Larger of the label of its children if those labels are different.
    - ★ One plus the label of its children if both the labels are same.

Optimal Code Generation in case of tree (assuming sufficient registers)

- Ershov Numbers: Numbers of registers needed to evaluate a node without storing any temporaries.
  - Label all leaves as 1.
  - The label of an interior node with one child is the label of its child
  - The label of an interior node with two children
    - Larger of the label of its children if those labels are different.
    - One plus the label of its children if both the labels are same.
- Recursive algo, start from root node. If a node is labelled as k then only k registers will be used

Optimal Code Generation in case of tree (assuming sufficient registers)

- Ershov Numbers: Numbers of registers needed to evaluate a node without storing any temporaries.
    - Label all leaves as 1.
    - The label of an interior node with one child is the label of its child
    - The label of an interior node with two children
        - Larger of the label of its children if those labels are different.
        - One plus the label of its children if both the labels are same.
- Recursive algo, start from root node. If a node is labelled as k then only k registers will be used
- There is a "base" b $\geq$ 1 for registers use. so that actual registers used are $R_b, R_{b+1}, \cdots, R_{b+k-1}$and the result always appears in R_{b+k-1}

Optimal Code Generation in case of tree (assuming sufficient registers)

- Ershov Numbers: Numbers of registers needed to evaluate a node without storing any temporaries.
  - Label all leaves as 1.
  - The label of an interior node with one child is the label of its child
  - The label of an interior node with two children
    - Larger of the label of its children if those labels are different.
    - One plus the label of its children if both the labels are same.
- Recursive algo, start from root node. If a node is labelled as k then only k registers will be used
- There is a "base" b $\geq$ 1 for registers use. so that actual registers used are $R_b, R_{b+1}, \cdots, R_{b+k-1}$and the result always appears in R_{b+k-1}
- For a leaf operand x, if base is b generate the instruction `LD` R_b`, x`