Lecture 31

Code Generation

Awanish Pandey

Department of Computer Science and Engineering
Indian Institute of Technology
Roorkee

April 22, 2025

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
    - ⋆ The Larger of the labels 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
    - ⋆ The Larger of the labels 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
    - The Larger of the labels 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
    - ⋆ The Larger of the labels 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 ≥ 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
    - ⋆ The Larger of the labels 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 ≥ 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`

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

- Interior node with label k and children with unequal labels.

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

- Interior node with label k and children with unequal labels.
  - Recursively generate code for the big child, using base b. Result will be in R_{b+k-1}

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

- Interior node with label k and children with unequal labels.
  - Recursively generate code for the big child, using base b. Result will be in R_{b+k-1}
  - Recursively generate code for the little child (assume label m), using base b and result will be in R_{b+m-1}.

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

- Interior node with label k and children with unequal labels.
  - Recursively generate code for the big child, using base b. Result will be in R_{b+k-1}
  - Recursively generate code for the little child (assume label m), using base b and result will be in R_{b+m-1}.
  - Generate `OP` R_{b+k-1}, R_{b+m-1}, R_{b+k-1} or `OP` R_{b+k-1}, R_{b+k-1}, R_{b+m-1} depending on big child is on left or right.

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

- Interior node with label k and children with unequal labels.
  - Recursively generate code for the big child, using base b. Result will be in R_{b+k-1}
  - Recursively generate code for the little child (assume label m), using base b and result will be in R_{b+m-1}.
  - Generate `OP` R_{b+k-1}, R_{b+m-1}, R_{b+k-1} or `OP` R_{b+k-1}, R_{b+k-1}, R_{b+m-1} depending on big child is on left or right.
- Interior node with label k and both the children has same label

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

- Interior node with label k and children with unequal labels.
  - Recursively generate code for the big child, using base b. Result will be in R_{b+k-1}
  - Recursively generate code for the little child (assume label m), using base b and result will be in R_{b+m-1}.
  - Generate `OP` R_{b+k-1}, R_{b+m-1}, R_{b+k-1} or `OP` R_{b+k-1}, R_{b+k-1}, R_{b+m-1} depending on big child is on left or right.
- Interior node with label k and both the children has same label
  - Recursively generate code for right child using base $b+1$. Result will be stored in R_{b+k-1}

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

- Interior node with label k and children with unequal labels.
  - Recursively generate code for the big child, using base b. Result will be in R_{b+k-1}
  - Recursively generate code for the little child (assume label m), using base b and result will be in R_{b+m-1}.
  - Generate `OP` R_{b+k-1}, R_{b+m-1}, R_{b+k-1} or `OP` R_{b+k-1}, R_{b+k-1}, R_{b+m-1} depending on big child is on left or right.
- Interior node with label k and both the children has same label
  - Recursively generate code for right child using base $b+1$. Result will be stored in R_{b+k-1}
  - Recursively generate code for left child, using base b, result will be in R_{b+k-2}

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

- Interior node with label k and children with unequal labels.
  - Recursively generate code for the big child, using base b. Result will be in R_{b+k-1}
  - Recursively generate code for the little child (assume label m), using base b and result will be in R_{b+m-1}.
  - Generate `OP` R_{b+k-1}, R_{b+m-1}, R_{b+k-1} or `OP` R_{b+k-1}, R_{b+k-1}, R_{b+m-1} depending on big child is on left or right.
- Interior node with label k and both the children has same label
  - Recursively generate code for right child using base $b+1$. Result will be stored in R_{b+k-1}
  - Recursively generate code for left child, using base b, result will be in R_{b+k-2}
  - Generate the instruction `OP` R_{b+k-1}, R_{b+k-2}, R_{b+k-1}

Optimal Code Generation in case of tree (insufficient registers)

Start from the root of the tree with base $b = 1$. Assume the number of registers are $r >= 2$. For node N with label r or less, use same algo.

Optimal Code Generation in case of tree (insufficient registers)

Start from the root of the tree with base $b = 1$. Assume the number of registers are $r >= 2$. For node N with label r or less, use same algo.

- Recursively generate code for the big child, using base $b = 1$. The result will be stored in R_r

Optimal Code Generation in case of tree (insufficient registers)

Start from the root of the tree with base $b = 1$. Assume the number of registers are $r >= 2$. For node N with label r or less, use same algo.

- Recursively generate code for the big child, using base $b = 1$. The result will be stored in R_r
- Generate `ST` t_k , R_r

Optimal Code Generation in case of tree (insufficient registers)

Start from the root of the tree with base $b = 1$. Assume the number of registers are $r >= 2$. For node N with label r or less, use same algo.

- Recursively generate code for the big child, using base $b = 1$. The result will be stored in R_r
- Generate `ST` t_k , R_r
- Generate code for the little child

Optimal Code Generation in case of tree (insufficient registers)

Start from the root of the tree with base $b = 1$. Assume the number of registers are $r >= 2$. For node N with label r or less, use same algo.

- Recursively generate code for the big child, using base $b = 1$. The result will be stored in R_r
- Generate `ST` t_k , R_r
- Generate code for the little child
  - if its label (assume j) $\geq$ r, pick base $b = 1$

Optimal Code Generation in case of tree (insufficient registers)

Start from the root of the tree with base $b = 1$. Assume the number of registers are $r >= 2$. For node N with label r or less, use same algo.

- Recursively generate code for the big child, using base $b = 1$. The result will be stored in R_r
- Generate `ST` t_k , R_r
- Generate code for the little child
  - if its label (assume j) $\geq$ r, pick base $b = 1$
  - $j < r$, pick b = r - j

Optimal Code Generation in case of tree (insufficient registers)

Start from the root of the tree with base $b = 1$. Assume the number of registers are $r >= 2$. For node N with label r or less, use same algo.

- Recursively generate code for the big child, using base $b = 1$. The result will be stored in R_r
- Generate `ST` t_k , R_r
- Generate code for the little child
  - if its label (assume j) $\geq$ r, pick base $b = 1$
  - $j < r$, pick b = r - j
  - Recursively do this and store the result in R_r

Optimal Code Generation in case of tree (insufficient registers)

Start from the root of the tree with base $b = 1$. Assume the number of registers are $r >= 2$. For node N with label r or less, use same algo.

- Recursively generate code for the big child, using base $b = 1$. The result will be stored in R_r
- Generate `ST` t_k , R_r
- Generate code for the little child
  - if its label (assume j) $\geq$ r, pick base $b = 1$
  - $j < r$, pick b = r - j
  - Recursively do this and store the result in R_r
- Generate `LD` R_{r-1} , t_k

Optimal Code Generation in case of tree (insufficient registers)

Start from the root of the tree with base $b = 1$. Assume the number of registers are $r >= 2$. For node N with label r or less, use same algo.

- Recursively generate code for the big child, using base $b = 1$. The result will be stored in R_r
- Generate `ST` t_k , R_r
- Generate code for the little child
  - if its label (assume j) $\geq$ r, pick base $b = 1$
  - $j < r$, pick b = r - j
  - Recursively do this and store the result in R_r
- Generate `LD` R_{r-1} , t_k
- Generate `OP` R_r , R_r , R_{r-1} or `OP` R_r , R_{r-1} , R_r depending on big child is on left or right.

Optimal Code Generation in case of tree (insufficient registers)

Start from the root of the tree with base $b = 1$. Assume the number of registers are $r >= 2$. For node N with label r or less, use same algo.

- Recursively generate code for the big child, using base $b = 1$. The result will be stored in R_r
- Generate `ST` t_k , R_r
- Generate code for the little child
  - if its label (assume j) $\geq$ r, pick base $b = 1$
  - $j < r$, pick b = r - j
  - Recursively do this and store the result in R_r
- Generate `LD` R_{r-1} , t_k
- Generate `OP` R_r , R_r , R_{r-1} or `OP` R_r , R_{r-1} , R_r depending on big child is on left or right.

Peephole Optimization

- Target code often contains redundant instructions and suboptimal constructs

Peephole Optimization

- Target code often contains redundant instructions and suboptimal constructs
- Examine a short sequence of target instruction (peephole) and replace by a shorter or faster sequence

Peephole Optimization

- Target code often contains redundant instructions and suboptimal constructs
- Examine a short sequence of target instruction (peephole) and replace by a shorter or faster sequence
- Peephole is a small moving window on the target systems

Peephole Optimization

- Target code often contains redundant instructions and suboptimal constructs
- Examine a short sequence of target instruction (peephole) and replace by a shorter or faster sequence
- Peephole is a small moving window on the target systems
- Example

Peephole Optimization

- Target code often contains redundant instructions and suboptimal constructs
- Examine a short sequence of target instruction (peephole) and replace by a shorter or faster sequence
- Peephole is a small moving window on the target systems
- Example
  - Redundant load and stores

Peephole Optimization

- Target code often contains redundant instructions and suboptimal constructs
- Examine a short sequence of target instruction (peephole) and replace by a shorter or faster sequence
- Peephole is a small moving window on the target systems
- Example
  - Redundant load and stores
  - Constant Propogation

Peephole Optimization

- Target code often contains redundant instructions and suboptimal constructs
- Examine a short sequence of target instruction (peephole) and replace by a shorter or faster sequence
- Peephole is a small moving window on the target systems
- Example
  - Redundant load and stores
  - Constant Propogation
  - Redundant jump sequences

Peephole Optimization

- Target code often contains redundant instructions and suboptimal constructs
- Examine a short sequence of target instruction (peephole) and replace by a shorter or faster sequence
- Peephole is a small moving window on the target systems
- Example
  - Redundant load and stores
  - Constant Propogation
  - Redundant jump sequences
  - Simplify algebraic expressions

Peephole Optimization

- Target code often contains redundant instructions and suboptimal constructs
- Examine a short sequence of target instruction (peephole) and replace by a shorter or faster sequence
- Peephole is a small moving window on the target systems
- Example
  - Redundant load and stores
  - Constant Propogation
  - Redundant jump sequences
  - Simplify algebraic expressions
  - Strength reduction

Code Optimization

- Criteria for code improving transformation

Code Optimization

- Criteria for code improving transformation
  - Preserve the meaning

Code Optimization

- Criteria for code improving transformation
  - Preserve the meaning
  - Must speed up the program

Code Optimization

- Criteria for code improving transformation
  - Preserve the meaning
  - Must speed up the program
  - Must be worth the effort

Code Optimization

- Criteria for code improving transformation
  - Preserve the meaning
  - Must speed up the program
  - Must be worth the effort
  - The analysis must be fast

Code Optimization

- Criteria for code improving transformation
  - Preserve the meaning
  - Must speed up the program
  - Must be worth the effort
  - The analysis must be fast
- Local transformation: within basic blocks

Code Optimization

- Criteria for code improving transformation
  - Preserve the meaning
  - Must speed up the program
  - Must be worth the effort
  - The analysis must be fast
- Local transformation: within basic blocks
- Global transformation: across basic blocks