CS412/413

Introduction to Compilers
Radu Rugina

Lecture 24: Induction Variable Optimizations
27 Mar 02

Induction Variables

- Two categories of induction variables
- Basic induction variables: only incremented in loop body
  \( i = i + c \)
  where \( c \) is a constant (positive or negative)
- Derived induction variables: expressed as a linear function of an induction variable
  \( k = c^i + d \)
  where:
  - \( j \) is basic induction variable
  - \( j \) is derived induction variable in the family of \( i \), and:
    1. No definition of \( j \) outside the loop reaches definition of \( k \)
    2. \( i \) is not defined between the definitions of \( j \) and \( k \)

Families of Induction Variables

- Each basic induction variable defines a family of induction variables
  - Each variable in the family of \( i \) is a linear function of \( i \)
- A variable \( k \) is in the family of basic variable \( i \) iff:
  1. \( k = i \) (the basic variable itself)
  2. \( k \) is a linear function of other variables in the family of \( i \):
     \[ k = c^i + d, \text{ where } j \in \text{Family}(i) \]
- A triple \( <i, a, b> \) denotes an induction variable \( k \) in the family of \( i \) such that:
  - \( k = ia + b \)
  - Triple for basic variable \( i \) is \( <i, 1, 0> \)

Dataflow Analysis Formulation

- Detection of induction variables: can formulate problem using the dataflow analysis framework
  - Analyze loop sub-graph, except the back edge
  - Analysis is similar to constant folding
- Dataflow information: a function \( f \) that assigns a triple to each variable:
  \[ f(i) = <c, a, b>, \text{ if } k \text{ is an induction variable in family of } i \]
  \[ f(k) = \perp : k \text{ is not an induction variable} \]
  \[ f(k) = T : \text{don't know if } k \text{ is an induction variable} \]

Dataflow Analysis Formulation

- Meet operation: if \( f_1 \) and \( f_2 \) are two functions, then:
  \[ (f_1 \land f_2)(v) = \begin{cases} <c, a, b> \text{ if } f_1(v) = f_2(v) = <c, a, b> \\ \perp, \text{ otherwise} \end{cases} \]
  (in other words, use a lattice)
- Initialization:
  - Detect all basic induction variables
  - At loop header: \( f(i) = <1, 0> \) for each basic variable \( i \)
- Transfer function:
  - Consider \( f \) is information before induction \( I \)
  - Compute information \( f \) after \( I \)
Dataflow Analysis Formulation

- For a definition \( k = j \cdot c \), where \( k \) is not basic induction variable
  \[ F(v) = \langle a, b, c \rangle, \text{ if } v = k \text{ and } F(\beta) = \langle a, b, c \rangle \]
  \[ F(v) = F(v), \text{ otherwise} \]

- For a definition \( k = j \cdot c \), where \( k \) is not basic induction variable
  \[ F(v) = \langle a, b, c, b \cdot c \rangle, \text{ if } v = k \text{ and } F(\beta) = \langle a, b, c \rangle \]
  \[ F(v) = F(v), \text{ otherwise} \]

- For any other instruction and any variable \( k \) in def[1] :
  \[ F(v) = \langle a, b, c \rangle, \text{ if } v = k \text{ and } F(\beta) = \langle a, b, c \rangle \]
  \[ F(v) = F(v), \text{ otherwise} \]

Strength Reduction

- Basic idea: replace expensive operations (multiplications) with cheaper ones (additions) in definitions of induction variables
  \[ s = 3^j+1; \]
  \[ \text{while } (<10) \{ \]
  \[ j = 3^j; \]
  \[ a[j] = a[j] - 2; \]
  \[ i = i + 2; \]
  \[ i = i + 2; \]
  \[ \} \]

- Benefit: cheaper to compute \( s = 3^j+1 \) than \( j = 3^j \)
  - \( s = 3^j+1 \) requires an addition
  - \( j = 3^j \) requires a multiplication

General Algorithm

- Algorithm:
  - For each induction variable \( j \) with type \( \langle a, b, c \rangle \)
    - create a new variable \( s \)
    - replace definition of \( j \) with \( s \)
    - immediately after \( i = i + c \), insert \( s = s + a \cdot c \)
      (here \( a \cdot c \) is constant)
    - insert \( s = a \cdot c + b \) into preheader
  - Correctness: this transformation maintains the invariant that \( s = a \cdot c + b \)

Strength Reduction

- Gives opportunities for copy propagation, dead code elimination
  \[ s = 3^j+1; \]
  \[ \text{while } (<10) \{ \]
  \[ j = 3^j; \]
  \[ a[j] = a[j] - 2; \]
  \[ i = i + 2; \]
  \[ i = i + 2; \]
  \[ \} \]
  \[ s = 3^j+1; \]
  \[ \text{while } (<10) \{ \]
  \[ j = 3^j; \]
  \[ a[j] = a[j] - 2; \]
  \[ i = i + 2; \]
  \[ i = i + 2; \]
  \[ \} \]

Induction Variable Elimination

- Idea: eliminate each basic induction variable whose only uses are in loop test conditions and in their own definitions
  - rewrite loop test to eliminate induction variable
    \[ s = 3^j+1; \]
    \[ \text{while } (<10) \{ \]
    \[ a[s] = a[s] - 2; \]
    \[ i = i + 2; \]
    \[ s = s + 6; \]
    \[ \} \]

- When are induction variables used only in loop tests?
  - Usually, after strength reduction
  - Use algorithm from strength reduction even if definitions of induction variables don’t involve multiplications

Induction Variable Elimination

- Rewrite test condition using derived induction variables
  - Remove definition of basic induction variables (if not used after the loop)
    \[ s = 3^j+1; \]
    \[ \text{while } (<10) \{ \]
    \[ a[s] = a[s] - 2; \]
    \[ i = i + 2; \]
    \[ s = s + 6; \]
    \[ \} \]
**Induction Variable Elimination**

For each basic induction variable \( i \) whose only uses are
- The test condition \( i < u \)
- The definition of \( i = i + c \)

For each derived induction variable \( k \) in its family, with triple \( i; c; d \)
  - Replace test condition \( i < u \) with \( k < c u + d \)
  - Remove definition \( i = i + c \) if \( i \) is not live on loop exit

---

**Where We Are**

- Defined dataflow analysis framework
- Used it for several analyses
  - Live variables
  - Available expressions
  - Reaching definitions
  - Constant folding
- Loop transformations
  - Loop invariant code motion
  - Induction variables
- Next:
  - Pointer alias analysis

---

**Pointer Alias Analysis**

- Most languages use variables containing addresses
  - E.g. pointers (C/C++, references (Java), call-by-reference parameters (Pascal, C++, Fortran)
- **Pointer aliases**: multiple names for the same memory location, which occur when dereferencing variables that hold memory addresses
- **Problem**:
  - Don't know what variables read and written by accesses via pointer aliases (e.g. \(*p\), \(x = *p\), \(p.f\), \(x = p.f\), etc.)
  - Need to know accessed variables to compute dataflow information after each instruction

---

**Alias Analysis Problem**

- **Goal**:
  - For each variable \( v \) that may hold an address, compute the set \( \text{Pr}(v) \) of possible targets of \( v \)
    - \( \text{Pr}(v) \) is a set of variables (or objects)
    - \( \text{Pr}(v) \) includes stack- and heap-allocated variables (objects)
  - Is a "may" analysis: if \( x \in \text{Pr}(v) \), then \( v \) may hold the address of \( x \) in some execution of the program
  - **No alias information**: for each variable \( v \), \( \text{Pr}(v) = V \), where \( V \) is the set of all variables in the program

---

**Simple Alias Analyses**

- **Address-taken analysis**:
  - Consider \( AT \) = set of variables whose addresses are taken
  - Then, \( \text{Pr}(v) = AT \), for each pointer variable \( v \)
  - Addresses of heap variables are always taken at allocation sites (e.g. \( x = \text{new int}[2] \), \( x = \text{malloc}(8) \))
  - Hence \( AT \) includes all heap variables
- **Type-based alias analysis**:
  - If \( v \) is a pointer (or reference) to type \( T \), then \( \text{Pr}(v) \) is the set of all variables of type \( T \)
  - Example: \( p \) and \( q \) can be aliases only if \( p \) and \( q \) are references to objects of the same type
  - Works only for strongly-typed languages
**Dataflow Alias Analysis**

- **Dataflow analysis**: for each variable \( v \), compute points-to set \( \text{Pt}(v) \) at each program point.

- **Dataflow information**: set \( \text{Pt}(v) \) for each variable \( v \)
  - Can be represented as a graph \( G \subseteq 2^{V \times V} \)
  - Nodes = \( V \) (program variables)
  - There is an edge \( v \rightarrow u \) if \( u \in \text{Pt}(v) \)

\[
\begin{align*}
\text{Pt}(x) &= \{ y \} \\
\text{Pt}(y) &= \{ x, t \}
\end{align*}
\]

**Dataflow Alias Analysis**

- **Dataflow lattice**: \( (2^{V \times V}, \supseteq) \)
  - \( V \times V \) is set of all possible points-to relations
  - "may" analysis: top element is \( \supseteq \), meet operation is \( \cup \)

- **Transfer functions**: use standard dataflow transfer functions:
  \[\text{out}[I] = (\text{in}[I] \cup \text{kill}[I]) \cup \text{gen}[I]\]
  \[p = \text{addr} q \quad \text{kill}[I] = (p \times V) \quad \text{gen}[I] = ((p, p))\]
  \[p = q \quad \text{kill}[I] = (p \times V) \quad \text{gen}[I] = (p \times \text{Pt}(q))\]
  \[\text{if} = p \quad \text{kill}[I] = (p) \times \text{Pt}(p) \quad \text{gen}[I] = (p \times \text{Pt}(p))\]

- Transfer functions are monotonic, but not distributive!

**Alias Analysis Example**

- Program:
  \[x = \text{ba}; \quad y = \text{bb}; \quad c = \text{ci}; \quad \text{if}(I) = x = y; \quad \#x = c\]

- CFG:

- Points-to Graph (at the end of program):

**Alias Analysis Uses**

- Once alias information is available, use it in other dataflow analyses

- **Example**: Live variable analysis
  - Use alias information to compute \( \text{use}[I] \) and \( \text{def}[I] \) for load and store statements:

\[
\begin{align*}
x &= [y] \quad \text{use}[I] = \{ y \} \cup \text{Pt}(y) \quad \text{def}[I] = \{ x \} \\
[x] &= y \quad \text{use}[I] = \{ x, y \} \quad \text{def}[I] = \text{Pt}(x)
\end{align*}
\]