CSE443
Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall

http://www.cse.buffalo.edu/faculty/alphonce/SP17/CSE443/index.php
https://piazza.com/class/iybn4ndqa1s3ei
Announcements

- Grading survey - link posted in Piazza. Please respond by Sunday night.

- PRO5 will be posted over the weekend. Due Friday 5/12.

- HW5 will be posted over the weekend. Due Monday 5/1.
Phases of a compiler

Target machine code generation

Figure 1.6, page 5 of text
Code Transformations on basic blocks

- Local optimizations can be performed on code inside basic blocks.
- Represent code inside a basic block as a DAG.
Example

Figure 8.7 [p. 527]

1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
Identifying leaders

1) $i = 1$
2) $j = 1$
3) $t_1 = 10 \times i$
4) $t_2 = t_1 + j$
5) $t_3 = 8 \times t_2$
6) $t_4 = t_3 - 88$
7) $a[t_4] = 0.0$
8) $j = j + 1$
9) if $j \leq 10$ goto (3)
10) $i = i + 1$
11) if $i \leq 10$ goto (2)
12) $i = 1$
13) $t_5 = i - 1$
14) $t_6 = 88 \times t_5$
15) $a[t_6] = 1.0$
16) $i = i + 1$
17) if $i \leq 10$ goto (13)

Example from last class
\begin{verbatim}
\textbf{Figure 8.9} [p. 530]

\begin{align*}
\text{i} & = 1 \\
\text{j} & = 1 \\
\text{t1} & = 10 \times i \\
\text{t2} & = \text{t1} + \text{j} \\
\text{t3} & = 8 \times \text{t2} \\
\text{t4} & = \text{t3} - 88 \\
\text{a}[\text{t4}] & = 0.0 \\
\text{j} & = \text{j} + 1 \\
\text{if} & \ \text{j} \leq 10 \ \text{goto} \ \text{B3} \\
\text{i} & = \text{i} + 1 \\
\\text{if} & \ \text{i} \leq 10 \ \text{goto} \ \text{B2} \\
\text{i} & = 1 \\
\text{t5} & = \text{i} - 1 \\
\text{t6} & = 88 \times \text{t5} \\
\text{a}[\text{t6}] & = 1.0 \\
\text{i} & = \text{i} + 1 \\
\text{if} & \ \text{i} \leq 10 \ \text{goto} \ \text{B6}
\end{align*}
\end{verbatim}
Constructing DAG for basic blocks

1. For each variable in the block, create a node representing the variable's initial value.

2. For each statement in the block, create a node.
Constructing DAG for basic blocks

3. For each node representing a statement, label it with the operator applied.

4. For each node representing a statement, attach a list of the variables for which it is the last definition within the block.
Constructing DAG for basic blocks

5. For each node representing a statement, its children are the nodes that are the last definitions of the operands used in the statement.

6. Identify as output nodes those whose variables are live on exit from the block.
Example 8.10 [p. 534]

1) $a = b + c$
2) $b = a - d$
3) $c = b + c$
4) $d = a - d$
Example 8.10 [p. 534]

Apply the "value-number" method from section 6.1.1

1) \( a = b + c \)
2) \( b = a - d \)
3) \( c = b + c \)
4) \( d = a - d \)
Example 8.10 [p. 534]

Apply the "value-number" method from section 6.1.1

1) \( a = b + c \)
2) \( b = a - d \)
3) \( c = b + c \)
4) \( d = a - d \)
Example 8.10 [p. 534]

Apply the "value-number" method from section 6.1.1

1) \( a = b + c \)
2) \( b = a - d \)
3) \( c = b + c \)
4) \( d = a - d \)
Example 8.10 [p. 534]

Apply the "value-number" method from section 6.1.1

1) \( a = b + c \)
2) \( b = a - d \)
3) \( c = b + c \)
4) \( d = a - d \)
Example 8.10 [p. 534]

Apply the "value-number" method from section 6.1.1

1) \( a = b + c \)
2) \( b = a - d \)
3) \( c = b + c \)
4) \( d = a - d \)
Example 8.10 [p. 534]

If \( b \) is live on exit:

1) \( a = b + c \)
2) \( b = a - d \)
3) \( c = b + c \)
4) \( d = b \)
Example 8.10 [p. 534]

If b is not live on exit:

1) \( a = b + c \)
2) \( d = a - d \)
3) \( c = d + c \)
8.5.3 Dead Code Elimination

"Delete from a DAG any root [...] that has no live variables attached." [p. 535]
8.5.3 Dead Code Elimination

"Delete from a DAG and root [...] that has no live variables attached." [p. 535]

If c and e are NOT live on exit:

1) \( a = b + c \)
2) \( b = b - d \)
3) \( c = c + d \)
4) \( e = b + c \)
8.5.3 Dead Code Elimination

"Delete from a DAG and root [...] that has no live variables attached." [p. 535]

1) \(a = b + c\)
2) \(b = b - d\)
3) \(c = c + d\)
4) \(e = b + c\)

Delete + node with \(e\) attached
8.5.3 Dead Code Elimination

"Delete from a DAG and root [...] that has no live variables attached." [p. 535]
8.5.3 Dead Code Elimination

"Delete from a DAG and root [...] that has no live variables attached." [p. 535]
8.5.4 Algebraic Identities

"...apply arithmetic identities...to eliminate computations from a basic block" [p. 536]

\[\begin{align*}
x + 0 &= 0 + x = x \\
x \times 1 &= 1 \times x = x \\
x \times 0 &= 0 \times x = 0 \\
x - 0 &= x \\
x / 1 &= x
\end{align*}\]
8.5.4 Algebraic Identities

"Another class of algebraic optimizations includes local reduction in strength...replacing a more expensive operator by a cheaper one..." [p. 536]

\[ x^2 = x \times x \]
\[ 2 \times x = x + x \text{ (or shift L for int)} \]
\[ x / 2 = x \times 0.5 \text{ (or shift R for int)} \]
8.5.4 Algebraic Identities

"A third class ... is constant folding, ... evaluate constant expressions at compile time..." [p. 536]

\[ 2 \times 3.14 = 6.28 \]

In footnote: "Arithmetic expressions should be evaluated the same way at compile time as they are at run time. [...] compile the constant expression, execute the target code on the spot, and replace the expression with the result." [p. 536]. Consider the problem of cross-compilation.
8.5.4 Algebraic Identities

"The DAG-construction process can help us apply these and other more general algebraic transformations such as commutativity and associativity." [p. 536]

"Before we create a new node labeled * with left child M and right child N, we always check whether such a node already exists. However, because * is commutative, we should then check for a node having operator *, left child N, and right child M." [p. 536]
8.5.4 Algebraic Identities

$x < y$ may be tested by computing $(y-x)$ and looking at the resulting condition codes.

If the code already computes $(y-x)$ it may not be necessary to compute this result twice.
8.5.4 Algebraic Identities

Consider:

\[ a = b + c \]
\[ e = c + d + b \]
8.5.4 Algebraic Identities

Consider:

\[ a = b + c \]
\[ e = c + d + b \]

Note that the sum \( b + c \) is computed twice.
8.5.4 Algebraic Identities

Consider:

\[ a = b + c \]
\[ e = c + d + b \]

Note that the sum \( b + c \) is computed twice.

Using both the associativity and commutativity of \( + \) we can rearrange:

\[ a = b + c \]
\[ e = b + c + d \]
Consider:
\[ a = b + c \]
\[ e = c + d + b \]

Note that the sum \( b + c \) is computed twice.

Using both the associativity and commutativity of + we can rearrange:
\[ a = b + c \]
\[ e = b + c + d \]

and then simplify to:
\[ a = b + c \]
\[ e = a + d \]
Array indexing must be handled with care.

"An assignment from an array, like $x = a[i]$, is represented by creating a node with operator $=\[]$ and two children representing the initial value of the array, $a_0$ in this case, and the index $i$. Variable $x$ becomes a label of this new node." [p. 537]
\[ x = a[i] \]
"An assignment to an array, like $a[j] = y$, is represented by creating a node with operator $[]=$ and three children representing $a_0$, $j$ and $y$. There is no variable labeling this node. What is different is that the creation of this node kills all currently constructed nodes whose value depends on $a_0$. A node that has been killed cannot receive any more labels; that is, it cannot become a common subexpression." [p. 537]
${a[j]} = y$
Ex. 8.13 [p. 538]

1) \( x = a[i] \)
2) \( a[j] = y \)
3) \( z = a[i] \)

Issue: we cannot assume that \( a[i] \) in 3rd statement is the same as \( a[i] \) in the first, as it may be that case that \( i = j \)
Ex. 8.13 [p. 538]

1) \( x = a[i] \)
2) \( a[j] = y \)
3) \( z = a[i] \)
Ex. 8.13 [p. 538]

1) \( x = a[i] \)
2) \( a[j] = y \)
3) \( z = a[i] \)

Because the first node we built depends on \( a_0 \), it is killed, meaning that no more variables can be added to its label from this point on.
The effect of this is that even though the third statement involves $a[i]$, we cannot structure share by adding $z$ as a label next to $x$. Instead a new node must be constructed, forcing recomputation of this value.
Next week

- Monday: guest lecture by Kris

- W/F/M:
  - 8.5 A few remaining details
  - 8.6 A simple code generator
  - 8.7 Peephole optimization
  - 8.8 Register allocation & assignment
  - 8.9 Instruction selection by tree rewriting
  - 8.10 Optimal code generation for expressions
  - 8.11 Dynamic programming code generation
Further ahead

9. Machine-Independent optimizations
10. Instruction-level Parallelism
11. Optimizing for Parallelism and Locality
12. Interprocedural Analysis