Reminders

• Project Part 4 due on Friday, March 15.
• There will be a short project report due on Sunday, March 17 – at most one late day may be used for the report (if you have any left).
  – One-two pages
  – Describe what you did, what works and doesn’t work, how you tested, what you would have done the same/different, etc...
  – More details on the assignment page (out soon).
  – Technical writing is an important skill for engineers – don’t blow this off. “Concise but precise, and clear enough that even a manager can understand it ...”
• Laure out of town – no office hours today.
Today’s Agenda

• Finish our optimization overview from Friday.
• Begin discussing Dataflow Analysis, with specific examples of how it is used (e.g., Common Subexpression Elimination a.k.a. CSE).
  – (No, this is not the UW Department of Common Subexpression Elimination...)

Review: Intraprocedural Constant Propagation & Folding

• Create tables mapping each variable in scope to one of:
  – A particular constant
  – NonConstant
  – Undefined
• Propagate current table along control flow edges in the CFG
• Transformation at each instruction in a basic block (straightline code):
  – If instruction is an assignment of a constant to a variable, set variable as constant in table
  – If we reference a variable that the table maps to a constant, then replace it with the constant (constant propagation)
  – If an expression involves only constants, and has no side-effects, then perform operation at compile-time and replace with constant result (constant folding)
Merging data flow analysis info

- To propagate between blocks, we must account for merges (multiple incoming control flow edges).
- Constraint: merge results must be sound/conservative
  - If something is believed true after the merge, then it must be true no matter which path we took into the merge
  - I.e., only things true for all predecessors are true after merge
- To merge two maps of constant information, build map by merging corresponding variable information (merge x’s, merge y’s, etc.)
- To merge information about a variable from two paths:
  - If Undefined in one path, keep the status from the other (uninitialized variables are allowed to have any value)
  - If both paths have the same constant, keep that constant
  - Otherwise, degenerate to NonConstant

Example Merges

```c
// Block A
int x;
x = 5;
if (foo) {
  // Block B
  z++;
} else {
  // Block C
  z--;
}
// Block D
```

A {x:5}

B {x:5}  C {x:5}

D {x:5}
Example Merges

// Block A
int x;
if (foo) {
  // Block B
  z++;
  x = 5;
} else {
  // Block C
  z--;
  x = 5;
}
// Block D
...

// Block A
int x;
if (foo) {
  // Block B
  z++;
  x = 5;
} else {
  // Block C
  z--;
  x = 4;
}
// Block D
...
Example Merges

A {x:Undefined}
B [x:Undefined]
C {x:4}
D {x:4}

// Block A
int x;
if (foo) {
  // Block B
  z++;
} else {
  // Block C
  z--; 
  x = 4;
}
// Block D


How to analyze loops

i = 0;
x = 10;
y = 20;
while (...) {
  // what’s true here?
  ...
  i = i + 1;
y = 30;
} 
// what’s true here?
... x ... i ... y ...

• What do we do about backwards edges (aka, loops)?
• Safe but imprecise: forget everything when we enter or exit a loop
• Precise but unsafe: keep everything when we enter or exit a loop
• Can we do better?
Optimistic Iterative Analysis

- Assuming information at loop head is same as information at loop entry
- Then analyze loop body (using this head assumption), and compute information known at back edge
- Merge information at loop back edge with current loop head information
- Test if merged information is same as original assumption
  - If so, then we’re done
  - If not, then replace previous assumption with merged information,
  - and repeat analysis of loop body
Example

i = 0;
x = 10;
y = 20;
while (...) {
    // what’s true here?
    ...
    i = i + 1;
y = 30;
} // what’s true here?
... x ... i ... y ...

Example

i = 0; x = 10; y = 20
while (...) {
    // what’s true here?
    ...
    i = i + 1;
y = 30;
} // what’s true here?
... x ... i ... y ...
Example

i = 0;
x = 10;
y = 20;
while (...) {
    // what’s true here?
    ...
    i = i + 1;
y = 30;
} // what’s true here?
... x ... i ... y ...

Winter 2013        UW CSE 401 (Michael Ringenburg)
Example

```
i = 0;
x = 10;
y = 20;
while (...) {
    // what’s true here?
    ...
    i = i + 1;
    y = 30;
} // what’s true here?
... x ... i ... y ...
```

Example

```
i = 0;
x = 10;
y = 20;
while (...) {
    // what’s true here?
    ...
    i = i + 1;
    y = 30;
} // what’s true here?
... x ... i ... y ...
```
Example

```c
i = 0;
x = 10;
y = 20;
while (...) {
    // what’s true here?
    ...
    i = i + 1;
y = 30; }
// what’s true here?
... x ... i ... y ...
```

```
i = NC, x = 10, y = NC
i = NC, x = 10, y = 30
```

```
i = NC, x = 10, y = NC
```

```
i = NC, x = 10, y = NC
```
Why does this work?

- Why are the results always conservative?
- Because if the algorithm stops, then
  - the loop head info is at least as conservative as both the loop entry info and the loop back edge info
  - the analysis within the loop body is conservative, given the assumption that the loop head info is conservative

More analyses

- Alias analysis
  - Detect when different references may or must refer to the same memory locations
- Escape analysis
  - Pointers that are live on exit from procedures
  - Pointed to data may “escape” to other procedures or threads
- Dependence analysis
  - Determining which references depend on other references
  - May analyze array subscripts that depend on loop induction variables, to determine which loop iterations depend on each other.
    - Important for loop parallelization/vectorization
Optimization Summary

- Optimizations organized as collections of passes, each rewriting IL in place into (hopefully) better version
- Each pass does analysis to determine what is possible, followed by (or concurrent with) transformations that (hopefully) improve the program
  - Sometimes have “analysis-only” passes – produce info used by later passes

Next topic: Dataflow Analysis

- A framework and algorithm for many common compiler analyses
- Initial example: dataflow analysis for common subexpression elimination
- Other analysis problems that work in the same framework
- We’ll be discussing some of the same optimizations we saw in the optimization overview, but with more formalism and details.
Motivating Example: Common Subexpression Elimination (CSE)

• Goal: Find common subexpressions, replace with temporaries
• Idea: calculate available expressions at beginning of each basic block
• Avoid re-evaluation of an available expression – copy a temp instead
  – Simple inside a single block; more complex dataflow analysis used across blocks

“Available” and Other Terms

• An expression e is defined at point p in the CFG (control flow graph) if its value is computed at p
  – Sometimes called definition site
• An expression e is killed at point p if one of its operands (components) is redefined at p
  – Sometimes called kill site
• An expression e is available at point p if every path leading to p contains a prior definition of e and e is not killed between that definition and p
Available Expression Sets

- To compute available expressions, for each block $b$, define
  - $\text{AVAIL}(b)$ – the set of expressions available on entry to $b$
  - $\text{NKILL}(b)$ – the set of expressions not killed in $b$
  - $\text{DEF}(b)$ – the set of expressions defined in $b$ and not subsequently killed in $b$

\[
\text{AVAIL}(b) = \bigcap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x)))
\]

- $\text{preds}(b)$ is the set of $b$’s predecessors in the CFG
- In “english”, the expressions available on entry to $b$ are the expressions that were available at the end of every preceeding basic block $x$. (This is the $\bigcap_{x \in \text{preds}(b)}$)
- The expressions available at the end of block $x$ are exactly those that were defined in $x$ (and not killed), and those that were available at the beginning of $x$ and not killed in $x$.

- Applying to every block gives a system of simultaneous equations – a dataflow problem
Computing Available Expressions

- Big Picture
  - Build control-flow graph
  - Calculate initial local data – DEF(b) and NKILL(b)
    - This only needs to be done once
  - Iteratively calculate AVAIL(b) by repeatedly evaluating equations until nothing changes
    - Another fixed-point algorithm

Computing DEF and NKILL (1)

- For each block $b$ with operations $o_1, o_2, ..., o_k$

  $KILLED = \emptyset$  // Killed variables (not expressions)
  $DEF(b) = \emptyset$
  for $i = k$ to 1  // Note we are working backwards - important
    assume $o_i$ is “$x = y + z$”
    if ($y \notin KILLED$ and $z \notin KILLED$)  // Expression in DEF only if
      add “$y + z$” to $DEF(b)$      // they aren’t later killed
    add $x$ to $KILLED$

...
Example: Computing DEF and KILL

\[ x = a + b; \quad \text{DEF} = \{ \} \]
\[ b = c + d; \quad \text{KILL} = \{ \} \]
\[ m = 5*n; \]

Example: Computing DEF and KILL

\[ x = a + b; \quad \text{DEF} = \{ 5*n \} \]
\[ b = c + d; \quad \text{KILL} = \{ m \} \]
\[ m = 5*n; \]
Example: Computing DEF and KILL

\[
x = a + b; \\
b = c + d; \\
m = 5*n;
\]

DEF = \{ 5*n, \( c+d \) \}

KILL = \{ m, b \}

Winter 2013
UW CSE 401 (Michael Ringenburg)
Computing DEF and NKILL (2)

- After computing DEF and KILLED for a block b,

```
// NKILL is expressions not killed.
NKILL(b) = {all expressions} // Start with all
for each expression e // Remove any killed
  for each variable v ∈ e
    if v ∈ KILLED then
      NKILL(b) = NKILL(b) - e
```

Example: Computing DEF and NKILL

```
\[
x = a + b;
b = c + d;
m = 5*n;
\]
```

DEF = \{ 5*n, c+d \}
KILL = \{ m, b, x \}
NKILL = all expressions that don’t use m, b, or x
Computing Available Expressions

- Once DEF(b) and NKILL(b) are computed for all blocks b, compute AVAIL for all blocks by repeatedly applying the previous formula in a fixed-point algorithm:

\[
\text{AVAIL}(b) = \bigcap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x)))
\]

Example: Computing DEF and NKILL

\[
\begin{align*}
\text{DEF} &= \{ 5*n, c+d \} \\
\text{NKILL} &= \text{exprs w/o } m, b, \text{ or } x
\end{align*}
\]

\[
\begin{align*}
\text{DEF} &= \{ 2*a, 2*b \} \\
\text{NKILL} &= \text{exprs w/o } j \text{ or } k
\end{align*}
\]

\[
\begin{align*}
\text{DEF} &= \{ 5*n \} \\
\text{NKILL} &= \text{exprs w/o } c
\end{align*}
\]

\[
\begin{align*}
\text{DEF} &= \{ 2*a \} \\
\text{NKILL} &= \text{exprs w/o } h
\end{align*}
\]
Example: Computing DEF and NKILL

$$\text{AVAIL}(b) = \bigcap_{x \in \text{pred}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x)))$$

\[
\begin{align*}
\text{DEF} &= \{ 5*n, c+d \} \\
\text{NKILL} &= \text{exprs w/o m, b, or x}
\end{align*}
\]

\[
\begin{align*}
\text{AVAIL} &= \{ \} \\
\text{DEF} &= \{ 2*a, 2*b \} \\
\text{NKILL} &= \text{exprs w/o j or k}
\end{align*}
\]

\[
\begin{align*}
\text{DEF} &= \{ 5*n \} \\
\text{NKILL} &= \text{exprs w/o c}
\end{align*}
\]

\[
\begin{align*}
\text{DEF} &= \{ 2*a \} \\
\text{NKILL} &= \text{exprs w/o h}
\end{align*}
\]
Example: Computing DEF and NKILL

\[ \text{AVAIL}(b) = \bigcap_{x \in \text{pred}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x))) \]

\[ j = 2a \]
\[ k = 2b \]
\[ x = a + b; \]
\[ b = c + d; \]
\[ m = 5n; \]
\[ c = 5n \]
\[ h = 2a \]

AVAIL = \{2*a, 2*b\}
DEF = \{5*n, c+d\}
NKILL = exprs w/o m, b, or x

AVAIL = \{5*n\}
DEF = \{2*a\}
NKILL = exprs w/o c

AVAIL = \{5*n\}
DEF = \{2*a\}
NKILL = exprs w/o h

= in Worklist

= Processing
Example: Computing DEF and NKILL

AVAIL(b) = ∩_{x∈pred(b)} (DEF(x) ∪ (AVAIL(x) ∩ NKILL(x)))

j= 2*a  
k = 2*b

AVAIL = { }  
DEF = { 2*a, 2*b }  
NKILL = exprs w/o j or k

AVAIL = {2*a, 2*b}  
DEF = { 5*n, c+d }  
NKILL = exprs w/o m, b, or x

x = a + b;  
b = c + d;  
m = 5*n;

c = 5*n

AVAIL = {2*a, 2*b}  
DEF = { 5*n }  
NKILL = exprs w/o c

j= 2*a  
k = 2*b

AVAIL = { }  
DEF = { 2*a, 2*b }  
NKILL = exprs w/o j or k

h = 2*a

AVAIL = { 5*n, 2*a }  
DEF = { 2*a }  
NKILL = exprs w/o h

h = 2*a

AVAIL = { 5*n, 2*a }  
DEF = { 2*a }  
NKILL = exprs w/o h

h = 2*a

AVAIL = { }  
DEF = { 2*a, 2*b }  
NKILL = exprs w/o j or k

h = 2*a

AVAIL = { 5*n, 2*a }  
DEF = { 2*a }  
NKILL = exprs w/o h

h = 2*a

AVAIL = { }  
DEF = { 2*a, 2*b }  
NKILL = exprs w/o j or k

h = 2*a
Dataflow analysis

• Available expressions are an example of a dataflow analysis problem
• Many other compiler analyses can be expressed in a similar framework
• Only the first part of the story – once we’ve discovered facts, we then need to use them to improve code

Characterizing Dataflow Analysis

• All of these algorithms involve sets of facts about each basic block $b$
  – $IN(b)$ – facts true on entry to $b$
  – $OUT(b)$ – facts true on exit from $b$
  – $GEN(b)$ – facts created and not killed in $b$
  – $KILL(b)$ – facts killed in $b$
• These are related by the equation
  \[ OUT(b) = GEN(b) \cup (IN(b) - KILL(b)) \]
  – (Subtracting $KILL(b)$ is equivalent to intersecting $NKILL(b)$)
  – Solve this iteratively for all blocks
  – Sometimes information propagates forward; sometimes backward
Example: Live Variable Analysis

- A variable $v$ is *live* at point $p$ if and only if there is any path from $p$ to a use of $v$ along which $v$ is not redefined (i.e., $v$ might be used before it is redefined)
- Some uses:
  - Register allocation – only live variables need a register
  - Eliminating useless stores – if variable is not live at store, the stored value will never be used
  - Detecting uses of uninitialized variables – if live at declaration (before initialization), may be used uninitialized.
  - Improve SSA construction – only create phi functions (variable merges) for live variables - coming later ...

Liveness Analysis Sets

- For each block $b$, define
  - $\text{use}[b] = \text{variable used in } b \text{ before any } \text{def}$
  - $\text{def}[b] = \text{variable defined in } b \text{ before any use}$
  - $\text{in}[b] = \text{variables live on entry to } b$
  - $\text{out}[b] = \text{variables live on exit from } b$
Equations for Live Variables

- Given the preceding definitions, we have
  \[ \text{in}[b] = \text{use}[b] \cup (\text{out}[b] - \text{def}[b]) \]
  \[ \text{out}[b] = \bigcup_{s \in \text{succ}[b]} \text{in}[s] \]

- Algorithm
  - Set \( \text{in}[b] = \text{out}[b] = \emptyset \)
  - Update \( \text{in}, \text{out} \) until no change

Example

- Code
  
  \[
  \begin{align*}
  a &:= 0 \\
  L: & \quad b := a + 1 \\
  & \quad c := c + b \\
  & \quad a := b \times 2 \\
  & \quad \text{if } a < N \text{ goto } L \\
  & \quad \text{return } c
  \end{align*}
  \]

  1: \( a := 0 \)
  2: \( b := a + 1 \)
  3: \( c := c + b \)
  4: \( a := b \times 2 \)
  5: \( a < N \)
  6: \text{return } c
Calculation

1: a := 0
2: b := a + 1
3: c := c + b
4: a := b + 2
5: a < N
6: return c

\[
\text{in}[b] = \text{use}[b] \cup (\text{out}[b] - \text{def}[b]) \\
\text{out}[b] = \bigcup_{s \in \text{succ}(b)} \text{in}[s]
\]

Equations for Live Variables v2

- Many problems have more than one formulation. For example, Live Variables...
- Sets
  - USED(b) – variables used in b before being defined in b
  - NOTDEF(b) – variables not defined in b
  - LIVE(b) – variables live on exit from b
- Equation
  \[
  \text{LIVE}(b) = \bigcup_{s \in \text{succ}(b)} \text{USED}(s) \cup (\text{LIVE}(s) \cap \text{NOTDEF}(s))
  \]
Example: Reaching Definitions

• A definition \( d \) of some variable \( v \) reaches operation \( i \) iff \( i \) reads the value of \( v \) and there is a path from \( d \) to \( i \) that does not define \( v \) (i.e., \( i \) might use value defined at \( d \))

• Uses
  – Find all of the possible definition points for a variable in an expression

Equations for Reaching Definitions

• Sets
  – \( \text{DEFOUT}(b) \) – set of definitions in \( b \) that reach the end of \( b \) (i.e., not subsequently redefined in \( b \))
  – \( \text{SURVIVED}(b) \) – set of all definitions not obscured by a definition in \( b \)
  – \( \text{REACHES}(b) \) – set of definitions that reach \( b \)

• Equation
  \[
  \text{REACHES}(b) = \bigcup_{p \in \text{preds}(b)} \text{DEFOUT}(p) \cup (\text{REACHES}(p) \cap \text{SURVIVED}(p))
  \]
Example: Very Busy Expressions

- An expression $e$ is considered very busy at some point $p$ if $e$ is evaluated and used along every path that leaves $p$, and evaluating $e$ at $p$ would produce the same result as evaluating it at the original locations.

- Uses
  - Code hoisting – move $e$ to $p$ (reduces code size; no effect on execution time)

Equations for Very Busy Expressions

- Sets
  - USED(b) – expressions used in $b$ before they are killed
  - KILLED(b) – expressions redefined in $b$ before they are used
  - VERYBUSY(b) – expressions very busy on exit from $b$

- Equation
  $$\text{VERYBUSY}(b) = \bigcap_{s \in \text{succ}(b)} \text{USED}(s) \cup (\text{VERYBUSY}(s) - \text{KILLED}(s))$$
Using Dataflow Information

- A few examples of possible transformations...

Classic Common-Subexpression Elimination

- In a statement $s$: $t := x \text{ op } y$, if $x \text{ op } y$ is available at $s$ then it need not be recomputed
- Analysis: compute reaching expressions i.e., statements $n$: $v := x \text{ op } y$ such that the path from $n$ to $s$ does not compute $x \text{ op } y$ or define $x$ or $y$
  - As we saw in earlier example, available expressions may be available from different places in different paths (e.g., $5*n$ earlier).
**Classic CSE**

- If \( x \text{ op } y \) is defined at \( n \) and reaches \( s \)
  - Create new temporary \( w \)
  - Rewrite \( n \) as
    
    \[
    n: w := x \text{ op } y \\
    n': v := w
    \]
  - If multiple reaching definition points, rewrite all of them
  - Modify statement \( s \) to be
    
    \[
    s: t := w
    \]
  - (Rely on copy propagation to remove extra assignments if not really needed)

---

**Constant Propagation**

- Suppose we have
  - Statement \( d: t := c \), where \( c \) is constant
  - Statement \( n \) that uses \( t \)
- If \( d \) reaches \( n \) and no other definitions of \( t \) reach \( n \), then rewrite \( n \) to use \( c \) instead of \( t \)
  - Or (less common), if all reaching definitions set \( t \) to same constant \( c \).
Copy Propagation

- Similar to constant propagation
- Setup:
  - Statement d: t := z
  - Statement n uses t
- If d reaches n and no other definition of t reaches n, and there is no definition of z on any path from d to n, then rewrite n to use z instead of t
  - We saw earlier how this can help remove dead assignments

Copy Propagation Tradeoffs

- Downside is that this can increase the lifetime of variable z and increase need for registers or memory traffic
- But it can expose other optimizations, e.g.,
  
  a := y + z
  u := y
  c := u + z  // Copy propagation makes this y + z

- After copy propagation we can recognize the common subexpression
Dead Code Elimination

• If we have an instruction
  
  \[ s: a := b \text{ op } c \]
  
  and \( a \) is not live-out after \( s \), then \( s \) can be eliminated
  
  – Provided it has no implicit side effects that are visible (output, exceptions, etc.)
  
  – E.g., if \( b \) or \( c \) are a function call, they may have unknown side effects.

Dataflow...

• General framework for discovering facts about programs
  
  – Although not the only possible story

• And then: facts open opportunities for code improvement

• Next time: SSA (single static assignment) form – transform program to a new form where each variable has only a single definition.
  
  – Can make many optimizations/analyses more efficient