**Advanced Compilers**

**CMPSCI 710**

**Spring 2003**

**Basic Loop Optimizations**

Emery Berger

University of Massachusetts, Amherst

---

**Topics**

- **Last time**
  - Optimizations using SSA form
    - Constant propagation & dead code elimination
  - Loop invariant code motion
- **This time**
  - Loop optimizations
    - Induction variable
    - Linear test replacement
    - Loop unrolling
    - Scalar replacement

---

**Easy Detection of Loop Induction Variables**

- **Pattern match & check:**
  - Search for “i = i + b” in loop
  - i is induction variable if no other assignment to i in loop

- **Pros & Cons:**
  - Easy!
  - Does not catch all loop induction variables
    - e.g., “i = a * c + 2”

---

**Taxonomy of Induction Variables**

- **basic induction variable:**
  - only definition in loop is assignment
    - j = j + c, where c is loop invariant
- **mutual induction variable:**
  - definition is linear function of other induction variable i:
    - i = c1 * j + c2
    - i = j / c1 ± c2
- **family of basic induction variable j:**
  - set of induction variables i such that i is always assigned linear function of j

---

**Strength Reduction**

- Replace “expensive” op by “cheaper” one
  - E.g., replace multiply by addition
- Apply to induction variable families
  - Especially: array indexing

---

**Strength Reduction Algorithm**

- Let i be induction variable in the family of basic induction variable j:
  - i = c1 * j + c2
- Create new variable i’
  - Initialize in pre-header: i’ = c1 * j + c2
- Track value of j after j = j + c3, add i’ = i’ + (c1 * c3)
Strength Reduction Example

```
\[ i \leftarrow 0 \%
\]
```

Candidates for Strength Reduction

- Induction variable IV multiplied by invariant
  
  \[
  i \leftarrow 0
  \]
  \[
  \text{IV} \times \text{IV} + \text{IV}
  \]

- Recursively:
  - \( \text{IV} \times \text{IV}, \text{IV} \text{ mod constant}, \text{IV} + \text{IV} \)

Strength Reduction Algorithm

```
\[
\text{if } \text{IV} \text{ is linear function of other induction variables,}\n\text{then}\n\text{IV becomes new constant,}\n\text{otherwise}\n\text{IV is unchanged.}\n\]
```

Linear Test Replacement

- Eliminates induction variable!
  - After strength reduction, loop test is often last use of induction variable
  - Algorithm:
    - If only use of IV is loop test and its own increment, and test is always computed
      - i.e., only one exit from loop
    - Replace test with equivalent one:
      - E.g., \( i \text{ comp } k = \text{?”} \text{.50 comp } k \text{”} \)

Linear Test Replacement Example

```
\[
\begin{align*}
\text{IV} &= \text{iv} \\
\text{IV} &= \text{iv} + \text{iv} \\
\end{align*}
\]
```

```
\[
\begin{align*}
\text{IV} &= \text{iv} \\
\text{IV} &= \text{iv} + \text{iv} \\
\end{align*}
\]
```
**Loop Unrolling**

- To reduce loop overhead, we can *unroll* loops

```c
for (int i = 0; i < n; i++)
    a[i] = a[i-1] + b[i+1];
```

- Advantages:
  - Execute fewer total instructions
  - More fodder for common subexpression elimination, strength reduction, etc.
  - Move consecutive access closer together

- Disadvantages:
  - Still updating
  - Code bloat

**Scalar Replacement**

- Problem: register allocators never keep a[i] in register

- Idea: trick allocator
  - Locate patterns of consistent reuse
  - Replace load with copy into temporary
  - Replace store with copy from temporary
  - May need copies at end of loop
  - E.g., when reuse spans > 1 iteration

- Advantages:
  - Decreases number of loads and stores
  - Keeps reused values in registers
  - Big performance impact (2x, 3x!)

**Scalar Replacement Example**

```c
for (int i = 0; i < n; i++)
    a[i] = a[i-1] + b[i+1];
```

- Scalar replacement exposes the reuse of a[i]
  - Traditional scalar analysis – inadequate
  - Use dependence analysis to understand array references (later)

**Next Time**

- Common Subexpression Elimination
- Read ACDI:
  - Ch. 12, pp. 343-355
  - Ch. 13, pp. 378-396