


# CSE 582 – Compilers

---

## Optimizing Transformations

Hal Perkins  
Autumn 2002


12/5/2002 © 2002 Hal Perkins & UW CSE 5-1



## Agenda

- n A short catalog of typical optimizing transformations


12/5/2002 © 2002 Hal Perkins & UW CSE 5-2



## Role of Transformations

- n Data-flow analysis discovers opportunities for code improvement
- n Compiler must rewrite the code (IR) to realize these improvements
  - n A transformation may reveal additional opportunities for further analysis & transformation
  - n May also block opportunities by obscuring information


12/5/2002 © 2002 Hal Perkins & UW CSE 5-3



## Organizing Transformations in a Compiler

- n Typically middle end consists of many individual transformations that filter the IR and produce rewritten IR
- n No systematic theory for the order to apply them
  - n Sometimes want to apply a single transformation repeatedly


12/5/2002 © 2002 Hal Perkins & UW CSE 5-4



## A Taxonomy

- n Machine Independent Transformations
  - n Profitability may actually depend on machine architecture, but are typically implemented without considering this
- n Machine Dependent Transformations
  - n Most of the machine dependent code is in instruction selection & scheduling and register allocation
  - n Some machine dependent code belongs in the optimizer

12/5/2002 © 2002 Hal Perkins & UW CSE 5-5



## Machine Independent Transformations

- n Dead code elimination
- n Code motion
- n Specialization
- n Strength reduction
- n Enable other transformations
- n Eliminate redundant computations
  - n Value numbering, GCSE

12/5/2002 © 2002 Hal Perkins & UW CSE 5-6

## Machine Dependent Transformations

- Take advantage of special hardware
  - Expose instruction-level parallelism, for example
- Manage or hide latencies
  - Improve cache behavior
- Deal with finite resources

12/5/2002

© 2002 Hal Perkins & UW CSE

S-7

## Dead Code Elimination

- If a compiler can prove that a computation has no external effect, it can be removed
  - Useless operations
  - Unreachable operations
- Dead code often results from other transformations
  - Often want to do DCE several times

12/5/2002

© 2002 Hal Perkins & UW CSE

S-8

## Dead Code Elimination

- Classic algorithm is similar to garbage collection
  - Pass I – Mark all useful operations
    - Start with critical operations – output, entry/exit blocks, calls to other procedures, etc.
    - Mark all operations that are needed for critical operations; repeat until convergence
  - Pass II – delete all unmarked operations
  - Note: need to treat jumps carefully

12/5/2002

© 2002 Hal Perkins & UW CSE

S-9

## Code Motion

- Idea: move an operation to a location where it is executed less frequently
  - Classic situation: move loop-invariant code out of a loop and execute it once, not once per iteration
- Lazy code motion: code motion plus elimination of redundant and partially redundant computations

12/5/2002

© 2002 Hal Perkins & UW CSE

S-10

## Specialization

- Idea: Analysis phase may reveal information that allows a general operation in the IR to be replaced by a more specific one
  - Constant folding
  - Replacing multiplications and division by constants with shifts
  - Peephole optimizations
  - Tail recursion elimination

12/5/2002

© 2002 Hal Perkins & UW CSE

S-11

## Strength Reduction

- Classic example: Array references in a loop
  - for ( $k = 0$ ;  $k < n$ ;  $k++$ )  $a[k] = 0$ ;
- Simple code generation would usually produce address arithmetic including a multiplication ( $k * \text{elementsize}$ ) and addition

12/5/2002

© 2002 Hal Perkins & UW CSE

S-12

## Implementing Strength Reduction

- Idea: look for operations in a loop involving:
  - A value that does not change in the loop, the *region constant*, and
  - A value that varies systematically from iteration to iteration, the *induction variable*
- Create a new induction variable that directly computes the sequence of values produced by the original one; use an addition in each iteration to update the value

12/5/2002

© 2002 Hal Perkins & UW CSE

S-13

## Enabling Transformations

- Already discussed
  - Inline substitution (procedure bodies)
  - Block cloning
- Some others
  - Loop Unrolling
  - Loop Unswitching

12/5/2002

© 2002 Hal Perkins & UW CSE

S-14

## Loop Unrolling

- Idea: Replicate the loop body to expose inter-iteration optimization possibilities
  - Increases chances for good schedules and instruction level parallelism
  - Reduces loop overhead
- Catch – need to handle dependencies between iterations carefully

12/5/2002

© 2002 Hal Perkins & UW CSE

S-15

## Loop Unrolling Example

- Original

```
for (i=1, i<=n, i++)
  a[i] = b[i];
```
- Unrolled by 4

```
i=1;
while (i+3 <= n) {
  a[i]   = a[i]+b[i];
  a[i+1] = a[i+1]+b[i+1];
  a[i+2] = a[i+2]+b[i+2];
  a[i+3] = a[i+3]+b[i+3];
  a+=4;
}
while (i <= n) {
  a[i] = a[i]+b[i];
  i++;
}
```

12/5/2002

© 2002 Hal Perkins & UW CSE

S-16

## Loop Unswitching

- Idea: if the condition in an if-then-else is loop invariant, rewrite the loop by pulling the if-then-else out of the loop and generating a tailored copy of the loop for each half of the new if
  - After this transformation, both loops have simpler control flow – more chances for rest of compiler to do better

12/5/2002

© 2002 Hal Perkins & UW CSE

S-17

## Summary

- This is just a sampler
  - Hundreds of transformations in the literature
- Big part of engineering a compiler is to decide which transformations to use, in what order, and when to repeat them
  - Mostly based on tradition and best guess
  - Current research: using adaptive methods based on performance of specific programs to automate selection and sequencing of transformations

12/5/2002

© 2002 Hal Perkins & UW CSE

S-18