CSE P 501 – Compilers

Optimizing Transformations

Hal Perkins

Autumn 2011

© 2002-11 Hal Perkins & UW CSE

Agenda

- A sampler of typical optimizing transformations
  - Mostly a teaser for later, particularly once we've looked at analyzing loops

Role of Transformations

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

© 2002-11 Hal Perkins & UW CSE

Organizing Transformations in a Compiler

- Typically middle end consists of many individual transformations that filter the IR and produce rewritten IR
- No formal theory for order to apply them
  - Some rules of thumb and best practices
  - Some transformations can be profitably applied repeatedly, particularly if others transformations expose more opportunities

© 2002-11 Hal Perkins & UW CSE

A Taxonomy

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

© 2002-11 Hal Perkins & UW CSE

Machine Independent Transformations

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

© 2002-11 Hal Perkins & UW CSE

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

© 2002-11 Hal Perkins & UW CSE

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

© 2002-11 Hal Perkins & UW CSE

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
  - Need to treat jumps carefully

© 2002-11 Hal Perkins & UW CSE

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

© 2002-11 Hal Perkins & UW CSE

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

© 2002-11 Hal Perkins & UW CSE

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**elementsize*) and addition
- Optimization can produce `*p++ = 0;`

© 2002-11 Hal Perkins & UW CSE

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

© 2002-11 Hal Perkins & UW CSE

Some Enabling Transformations

- Inline substitution (procedure bodies)
- Block cloning
- Loop Unrolling
- Loop Unswitching

© 2002-11 Hal Perkins & UW CSE

Inline Substitution

- Idea: Replace method calls with a copy of the method body. Instead of

```
x = foo.getY();
```

use

```
x = foo.y
```

  - Eliminates call overhead
  - Opens possibilities for other optimizations
- But: Possible code bloat, need to catch changes to inlined code
- Still, huge win for much object-oriented code

© 2002-11 Hal Perkins & UW CSE

Code Replication

- Idea: duplicate code to increase chances for optimizations, better code generation
- Tradeoff: larger code size, potential interactions with caches, registers

© 2002-11 Hal Perkins & UW CSE

Code Replication Example

- Original

```
if (x < y) {
   p = x+y;
} else {
   p = z + 1;
}
q = p*3;
w = p + q;
```

- Duplicating code; larger basic blocks to optimize

```
if (x < y) {
   p = x+y;
   q = p*3;
   w = p + q;
} else {
   p = z + 1;
   q = p*3;
   w = p + q;
}
```


© 2002-11 Hal Perkins & UW CSE

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

© 2002-11 Hal Perkins & UW CSE

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++;
}
```


© 2002-11 Hal Perkins & UW CSE

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

© 2002-11 Hal Perkins & UW CSE

Loop UnswitchingExample

- Original

```
for (i=1, i<=n, i++)
    if (x > y)
      a[i] = b[i]*x;
    else
      a[i] = b[i]*y
```

- Unswitched

```
if (x > y)
  for (i = 1; i < n; i++)
    a[i] = b[i]*x;
  else
    a[i] = b[i]*y;
```


© 2002-11 Hal Perkins & UW CSE

Summary

- This is just a sampler
  - Hundreds of transformations in the literature
  - We will look at several in more detail, particularly involving loops
- Big part of engineering a compiler is to decide which transformations to use, in what order, and when to repeat them
  - Different tradeoffs depending on compiler goals

© 2002-11 Hal Perkins & UW CSE