CS6013 - Modern Compilers: Theory and Practise

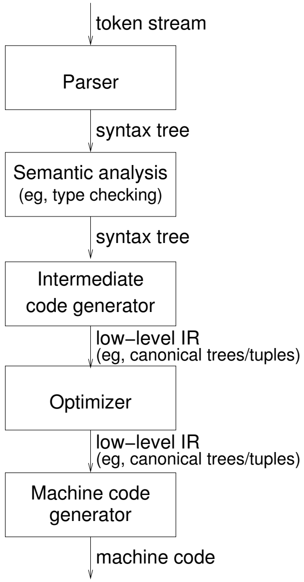
Overview of different optimizations

V. Krishna Nandivada

IIT Madras

Optimizing compilers

Compiler structure



Potential optimizations:

Source-language (AST):
- constant bounds in loops/arrays
- loop unrolling
- suppressing run-time checks
- enable later optimisations

IR: local and global
- CSE elimination
- live variable analysis
- code hoisting
- enable later optimisations

Code-generation (machine code):
- register allocation
- instruction scheduling
- peephole optimization

Optimization

Goal: produce fast code
- What is optimality?
- Problems are often hard
- Many are intractable or even undecideable
- Many are NP-complete
- Which optimizations should be used?
- Many optimizations overlap or interact

Optimization

Definition: An optimization is a transformation that is expected to:

improve the running time of a program
or decrease its space requirements

The point:

- "improved" code, not "optimal" code (forget "optimum")
- sometimes produces worse code
- range of speedup might be from 1.000001 to xxx

Machine-independent transformations

- applicable across broad range of machines
- remove redundant computations
- move evaluation to a less frequently executed place
- specialize some general-purpose code
- find useless code and remove it
- expose opportunities for other optimizations

Machine-dependent transformations

- capitalize on machine-specific properties
- improve mapping from IR onto machine
- replace a costly operation with a cheaper one
- hide latency
- replace sequence of instructions with more powerful one (use "exotic" instructions)

A classical distinction

The distinction is not always clear: replace `multiply` with `shift`s and `add`s

Optimization

Desirable properties of an optimizing compiler

- code at least as good as an assembler programmer
- stable, robust performance (predictability)
- architectural strengths fully exploited
- architectural weaknesses fully hidden
- broad, efficient support for language features
- instantaneous compiles

Unfortunately, modern compilers often drop the ball

Optimization

Good compilers are crafted, not assembled

- consistent philosophy
- careful selection of transformations
- thorough application
- coordinate transformations and data structures
- attention to results (code, time, space)

Compilers are engineered objects

- minimize running time of compiled code
- minimize compile time
- use reasonable compile-time space (serious problem)

Thus, results are sometimes unexpected

Scope of optimization

Local (single block)

- confined to straight-line code
- simplest to analyse
- time frame: '60s to present, particularly now

Intraprocedural (global)

- consider the whole procedure
- What do we need to optimize an entire procedure?
- classical data-flow analysis, dependence analysis
- time frame: '70s to present

Interprocedural (whole program)

- analyse whole programs
- What do we need to optimize and entire program?
- less information is discernible
- time frame: late '70s to present, particularly now

Optimization

Three considerations arise in applying a transformation:

- safety
- profitability
- opportunity

We need a clear understanding of these issues

- the literature often hides them
- every discussion should list them clearly

Safety

Fundamental question Does the transformation change the **results** of executing the code?

yes ⇒ don't do it!
no ⇒ it is safe

Compile-time analysis

- may be safe in all cases (loop unrolling)
- analysis may be simple (DAGs and CSEs)
- may require complex reasoning (data-flow analysis)

Profitability

Fundamental question Is there a reasonable expectation that the transformation will be an improvement?

yes ⇒ do it!
no ⇒ don't do it

Compile-time estimation

- always profitable
- heuristic rules
- compute benefit (rare)

Opportunity

Fundamental question Can we efficiently locate sites for applying the transformation?

yes ⇒ compilation time won't suffer
no ⇒ better be highly profitable

Issues

- provides a framework for applying transformation
- systematically find all sites
- update safety information to reflect previous changes
- order of application (hard)

Optimization

Successful optimization requires

- test for safety
- profit is *local improvement* × *executions*
  ⇒ focus on loops:
  - loop unrolling
  - factoring loop invariants
  - strength reduction
- want to minimize side-effects like code growth

Example: loop unrolling

Idea: reduce loop overhead by creating multiple successive copies of the loop's body and increasing the increment appropriately

Safety: always safe

Profitability: reduces overhead
(instruction cache blowout)
(subtle secondary effects)

Opportunity: loops

Unrolling is easy to understand and perform

Example: loop unrolling

Matrix-matrix multiply

```
do i ← 1, n, 1
   do j ← 1, n, 1
      c(i,j) ← 0
      do k ← 1, n, 1
         c(i,j) ← c(i,j) + a(i,k) * b(k,j)
```

- $2n^3$ flops, n^3 loop increments and branches
- each iteration does 2 loads and 2 flops

This is the most overstudied example in the literature

Example: loop unrolling

Matrix-matrix multiply (assume 4-word cache line)

```
do i ← 1, n, 1
   do j ← 1, n, 1
      c(i,j) ← 0
      do k ← 1, n, 4
         c(i,j) ← c(i,j) + a(i,k) * b(k,j)
         c(i,j) ← c(i,j) + a(i,k+1) * b(k+1,j)
         c(i,j) ← c(i,j) + a(i,k+2) * b(k+2,j)
         c(i,j) ← c(i,j) + a(i,k+3) * b(k+3,j)
```

- $2n^3$ flops, $\frac{n^3}{4}$ loop increments and branches
- each iteration does 8 loads and 8 flops
- memory traffic is better
  - `c(i,j)` is reused (put it in a register)
  - `a(i,k)` reference are from cache
  - `b(k,j)` is problematic

Example: loop unrolling

Matrix-matrix multiply (to improve traffic on `b`)

```
do j ← 1, n, 1
   do i ← 1, n, 4
      c(i,j) ← 0
      do k ← 1, n, 4
         c(i,j) ← c(i,j) + a(i,k) * b(k,j)
            + a(i,k+1) * b(k+1,j) + a(i,k+2) * b(k+2,j)
            + a(i,k+3) * b(k+3,j)
         c(i+1,j) ← c(i+1,j) + a(i+1,k) * b(k,j)
            + a(i+1,k+1) * b(k+1,j)
            + a(i+1,k+2) * b(k+2,j)
            + a(i+1,k+3) * b(k+3,j)
         c(i+2,j) ← c(i+2,j) + a(i+2,k) * b(k,j)
            + a(i+2,k+1) * b(k+1,j)
            + a(i+2,k+2) * b(k+2,j)
            + a(i+2,k+3) * b(k+3,j)
         c(i+3,j) ← c(i+3,j) + a(i+3,k) * b(k,j)
            + a(i+3,k+1) * b(k+1,j)
            + a(i+3,k+2) * b(k+2,j)
            + a(i+3,k+3) * b(k+3,j)
```

Example: loop unrolling

What happened?

- interchanged `i` and `j` loops
- unrolled `i` loop
- fused inner loops

- $2n^3$ flops, $\frac{n^3}{16}$ loop increments and branches
- first assignment does 8 loads and 8 flops
- 2nd through 4th do 4 loads and 8 flops

- memory traffic is better
  - `c(i,j)` is reused (register)
  - `a(i,k)` references are from cache
  - `b(k,j)` is reused (register)

Example: loop unrolling

It is not as easy as it looks:

Safety: loop interchange? loop unrolling? loop fusion?

Profitability: machine dependent (mostly)

Opportunity: find memory-bound loop nests

Summary

- chance for large improvement
- answering the fundamentals is tough
- resulting code is ugly

Matrix-matrix multiply is everyone's favorite example

Loop optimizations: factoring loop-invariants

Loop invariants: expressions constant within loop body
Relevant variables: those used to compute and expression

Opportunity:

1. identify variables defined in body of loop (*LoopDef*)
2. loop invariants have no relevant variables in *LoopDef*
3. assign each loop-invariant to temp. in loop header
4. use temporary in loop body

Safety: loop-invariant expression may throw exception early

Profitability:

- loop may execute 0 times
- loop-invariant may not be needed on every path through loop body

Example: factoring loop invariants

```
foreach i=1 .. 100 do
  // LoopDef = {i,j,k, A}
  foreach j=1 .. 100 do
    // LoopDef = {j,k, A}
    foreach k=1 .. 100 do
      // LoopDef = {k, A}
      A[i,j,k] = i * j * k;
    end
  end
end
```

- 3 million index operations
- 2 million multiplications

Example: factoring loop invariants (cont.)

Factoring the inner loop:

```
foreach i=1 .. 100 do
   // LoopDef = {i,j,k, A}
   foreach j=1 .. 100 do
      // LoopDef = {j,k, A}
      t1 = &A[i][j];
      t2 = i * j ;
      foreach k=1 .. 100 do
         // LoopDef = {k,A}
         t1[k] = t * k;
      end
   end
end
```

And the second loop:

```
foreach i=1 .. 100 do
   // LoopDef = {i,j,k, A}
   t3 = &A[i];
   foreach j=1 .. 100 do
      // LoopDef = {j,k, A}
      t1 = &t3[j];
      t2 = i * j ;
      foreach k=1 .. 100 do
         // LoopDef = {k,A}
         t1[k] = t * k;
      end
   end
end
```


Strength reduction in loops

Loop induction variable: incremented on each iteration
$i_0, i_0+1, i_0+2, \ldots$

Induction expression: $ic_1 + c_2$, where c_1, c_2 are loop invariant
$i_0c_1 + c_2, (i_0+1)c_1 + c_2, (i_0+2)c_1 + c_2, \ldots$

1. replace $ic_1 + c_2$ by t in body of loop
2. insert $t := i_0c_1 + c_2$ before loop
3. insert $t := t + c_1$ at end of loop

Example: strength reduction in loops

From previous example:

```
foreach i=1 .. 100 do
   t3 = &A[i];
   t4 = i; // i * j0 = i
   foreach j=1 .. 100 do
      t1 = &t3[j];
      t2 = t4; // t4 = i * j
      t5 = t2; // t2 * k0 = t2
      foreach k=1 .. 100 do
         t1[k] = t5; // t5 = t2 * k
         t5 = t5 + t2;
      end
      t4 = t4 + i;
   end
end
```


Example: strength reduction in loops

After copy propagation and exposing indexing:

```
foreach i=1 .. 100 do
   t3 = A + (10000 * i) - 10000;
   t4 = i;
   foreach j=1 .. 100 do
      t1 = t3 + (100 * j) - 100;
      t5 = t4;
      foreach k=1 .. 100 do
         *(t1 + k - 1) = t5;
         t5 = t5 + t4;
      end
      t4 = t4 + i;
   end
end
```

Example: strength reduction in loops

Applying strength reduction to exposed index expressions:

```
t6 = A;
foreach i=1 .. 100 do
   t3 = t6; t4 = i;
   t7 = t3;
   foreach j=1 .. 100 do
      t1 = t7; t5 = t4;
      t8 = t1;
      foreach k=1 .. 100 do
         *t8 = t5;
         t5 = t5 + t4;
         t8 = t8 + 1;
      end
      t4 = t4 + i;
      t7 = t7 + 100;
   end
   t6 = t6 + 10000;
end
```


Again, copy propagation further improves the code.

Ordering optimization phases

1. semantic analysis and intermediate code generation:
   - loop unrolling
   - inline expansion
2. intermediate code generation:
   - build basic blocks with their *Def* and *Kill* sets
3. build control flow graph:
   - perform initial data flow analyses
   - assume worst case for calls if no interproc. analysis
4. early data-flow optimizations: constant/copy propagation (may expose dead code, changing flow graph, so iterate)
5. CSE and live/dead variable analyses
6. translate basic blocks to target code: local optimizations (register allocation/assignment, code selection)
7. peephole optimization