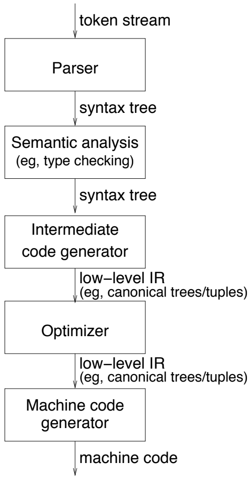
Optimizing compilers

Compiler structure


token stream
Parser
syntax tree
Semantic analysis
(eg, type checking)
syntax tree
Intermediate
code generator
low−level IR
(eg, canonical trees/tuples)
Optimizer
low−level IR
(eg, canonical trees/tuples)
Machine code
generator
machine code

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
- sometimes produces worse code
- range of speedup might be from 1.000001 to 4 (*or more*)

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 `shifts` and `adds`

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 (DAG*s and* CSE*s*)
- 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, profitability should be
  O(1) per transformation
  or **O**(n) for whole routine (*maybe* $n \log n$)
- 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
- 2^{nd} through 4^{th} 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

```
for $i := 1$ to $100$ do $(* \mathit{LoopDef} = \{i, j, k, A\} *)$
    for $j := 1$ to $100$ do $(* \mathit{LoopDef} = \{j, k, A\} *)$
        for $k := 1$ to $100$ do $(* \mathit{LoopDef} = \{k, A\} *)$
            $A[i, j, k] := i * j * k;$
```

- 3 million index operations
- 2 million multiplications

Example: factoring loop invariants (cont.)

Factoring the inner loop:

```
for $i := 1$ to $100$ do
    for $j := 1$ to $100$ do
        $t_1 := \mathit{ADR}(A[i][j]);$
        $t_2 := i * j;$
        for $k := 1$ to $100$ do
            $t1[k] := t_2 * k;$
```

And the second loop:

```
for $i := 1$ to $100$ do
    $t_3 := \mathit{ADR}(A[i]);$
    for $j := 1$ to $100$ do
        $t_1 := \mathit{ADR}(t_3[j]);$
        $t_2 := i * j;$
        for $k := 1$ to $100$ do
            $t1[k] := t_2 * k;$
```

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:

```
for $i := 1$ to 100 do
    $t_3 := ADR(A[i]);$
    $t_4 := i;$  $(* \ i * j_0 = i \ *)$
    for $j := 1$ to 100 do
        $t_1 := ADR(t_3[j]);$
        $t_2 := t_4;$  $(* \ t_4 = i * j \ *)$
        $t_5 := t_2;$  $(* \ t_2 * k_0 = t_2 \ *)$
        for $k := 1$ to 100 do
            $t1[k] := t_5;$  $(* \ t_5 = t_2 * k \ *)$
            $t_5 := t_5 + t_2;$
        end;
        $t_4 := t_4 + i;$
    end;
end;
```

Example: strength reduction in loops

After *copy propagation* and exposing indexing:

```
for $i := 1$ to 100 do
    $t_3 := A_0 + (10000 * i) - 10000;$
    $t_4 := i;$
    for $j := 1$ to 100 do
        $t_1 := t_3 + (100 * j) - 100;$
        $t_5 := t_4;$
        for $k := 1$ to 100 do
            $(t1 + k - 1) \uparrow := t_5;$
            $t_5 := t_5 + t_4;$
        end;
        $t_4 := t_4 + i;$
    end;
end;
```

Example: strength reduction in loops

Applying strength reduction to exposed index expressions:

```
$t_6 := A_0;$
for $i := 1$ to 100 do
    $t_3 := t_6;$ $t_4 := i;$ $t_7 := t_3;$
    for $j := 1$ to 100 do
        $t_1 := t_7;$ $t_5 := t_4;$ $t_8 := t_1;$
        for $k := 1$ to 100 do
            $t_8 \uparrow := t_5;$
            $t_5 := t_5 + t_4;$
            $t_8 := t_8 + 1;$
        end;
        $t_4 := t_4 + i;$
        $t_7 := t_7 + 100;$
    end;
    $t_6 := t_6 + 10000;$
end;
```

Again, copy propagation further improves the code.

Intraprocedural analysis and transformation

In order to *move*, *remove* or *rearrange* instructions, must

- understand the control flow of the procedure
- find and connect definitions and uses of variables
  ⇒ *data-flow analysis*

The control flow graph (CFG):

- nodes are basic blocks
- edges represent potential flow of control

Any-path flow analysis: property *may* be true if it holds along any path to node

All-paths flow analysis: property *must* be true if it holds along all paths to node

Live variables: any-path backward flow

Define:

$\mathit{In}(b)$: variables live on entry to block b
$\mathit{Out}(b)$: variables live on exit from block b

Let $S(b)$ be the set of all immediate *successors* of b. Then:

$$\begin{aligned} \mathit{Out}(b) &= \cup_{i\in S(b)}\mathit{In}(i) \\ S(b)=\phi \;\Rightarrow\;& \mathit{Out}(b)=\phi \end{aligned}$$

Now, define:

$\mathit{Use}(b)$: variables used in b before/without definition in b
$\mathit{In}(b)\supseteq \mathit{Use}(b)$
$\mathit{Def}(b)$: variables defined in b
$\mathit{In}(b)\supseteq \mathit{Out}(b)-\mathit{Def}(b)$

$\mathit{Use}(b)$ and $\mathit{Def}(b)$ are constant (independent of control flow)

Now, $v\in \mathit{In}(b)$ iff. $v\in \mathit{Use}(b)$ or $v\in \mathit{Out}(b)-\mathit{Def}(b)$
i.e., $\mathit{In}(b)=\mathit{Use}(b)\cup(\mathit{Out}(b)-\mathit{Def}(b))$

Uninitialized variables: any-path forward flow

Define:

$\mathit{In}(b)$: possibly uninitialized vars. on entry to b
$\mathit{Out}(b)$: possibly uninitialized vars. on exit from b

Let $P(b)$ be the set of all immediate *predecessors* of b. Then:

$$\begin{aligned} \mathit{In}(b) &= \cup_{i\in P(b)}\mathit{Out}(i) \\ P(b)=\phi \;\Rightarrow\;& \mathit{In}(b)=\mathcal{U} \text{ (the set of all variables)} \end{aligned}$$

Now, define:

$\mathit{Init}(b)$: vars known to be initialized on exit from b
(e.g., vars assigned legal values or tested before use)
$\mathit{Out}(b)\supseteq \mathit{In}(b)-\mathit{Init}(b)$
$\mathit{Uninit}(b)$: vars that become uninitialized in b,
not subsequently reassigned or tested
(e.g., assigned **null**, **free**d ptrs, nested variables)
$\mathit{Out}(b)\supseteq \mathit{Uninit}(b)$

Now, $\mathit{Out}(b)=\mathit{Uninit}(b)\cup(\mathit{In}(b)-\mathit{Init}(b))$

Available expressions: all-paths forward flow

An expression is available if recomputing it is redundant

Define:

$\mathit{RelVar}(T)$: all vars/temps used to calculate T
T is *available on exit* from b iff. $\forall X\in \mathit{RelVar}(T)$,
T is more recently defined in b than X
$\mathit{Out}(b)$: (value-numbered) temps available on exit from b
$\mathit{In}(b)$: temporaries available on entry to b

$$\begin{aligned} \mathit{In}(b) &= \cap_{i\in P(b)}\mathit{Out}(i) \\ P(b)=\phi \;\Rightarrow\;& \mathit{In}(b)=\phi \end{aligned}$$

Now, define:

$\mathit{Computed}(b)$: exprs computed in b, not subsequently killed
$\mathit{Kill}(b)$: exprs killed in b
(because one of its rel. vars. is defined in b)

Now, $\mathit{Out}(b)=\mathit{Computed}(b)\cup(\mathit{In}(b)-\mathit{Kill}(b))$

Very busy expressions: all-paths backward flow

An expression is very busy if its value is used along all paths before the expression is killed (reg. alloc., loop invariants)

Define: *Out*(b): expressions very busy on exit from b
In(b): expressions very busy on entry to b

Then:

$$\begin{aligned} \mathit{Out}(b) &= \cap_{i \in S(b)} \mathit{In}(i) \\ S(b) = \phi &\Rightarrow \mathit{Out}(b) = \phi \end{aligned}$$

Now, define: *Used*(b): all expressions used before they are killed in b
Kill(b): all expressions killed in b before they are used

Then, $\mathit{In}(b) = \mathit{Used}(b) \cup (\mathit{Out}(b) - \mathit{Kill}(b))$

A taxonomy of data-flow problems

	Forward	Backward
Any	$\mathit{Out}(b) = \mathit{Gen}(b) \cup (\mathit{In}(b) - \mathit{Kill}(b)$ $\mathit{In}(b) = \bigcup_{i \in P(b)} \mathit{Out}(i)$	$\mathit{In}(b) = \mathit{Gen}(b) \cup (\mathit{Out}(b) - \mathit{Kill}(b)$ $\mathit{Out}(b) = \bigcup_{i \in S(b)} \mathit{In}(i)$
All	$\mathit{Out}(b) = \mathit{Gen}(b) \cup (\mathit{In}(b) - \mathit{Kill}(b)$ $\mathit{In}(b) = \bigcap_{i \in P(b)} \mathit{Out}(i)$	$\mathit{In}(b) = \mathit{Gen}(b) \cup (\mathit{Out}(b) - \mathit{Kill}(b)$ $\mathit{Out}(b) = \bigcap_{i \in S(b)} \mathit{In}(i)$

Other data-flow problems

Reaching definitions: a definition of v *reaches* a use of v if $\exists$ any path from the definition to the use, no intervening definitions
(register targeting, constant propagation)

- *In*(b): defs that reach entry to b
- *Out*(b): defs that reach exit from b
- *Gen*(b): defs in b that reach exit from b
- *Kill*(b): defs killed by b (i.e., not in *Gen*(b))

ud-chains: reaching defs associated with each use (forward)

Other data-flow problems (cont.)

du-chains: uses associated with each definition (backward)

- *In*(b): possible uses of a var defined on entry to b
- *Out*(b): possible uses of a var defined on exit from b
- *Gen*(b): uses in b of a var not yet defined in b
- *Kill*(b): uses in b of vars defined in b

Copy propagation: given `A:=B`, replace use of `A` with use of `B`

- *In*(b): copy statements that reach entry to b
- *Out*(b): copy statements that reach exit from b
- *Gen*(b): copy statements in b that reach end of b
- *Kill*(b): copy statements that reach entry to, but not exit from, b

Optimizations based on global data-flow

Very busy expressions:

- move invariants outside loops (used on every iter^{n})
- *code hoisting*: move from successors common predecessor (*dominator*)

Global CSE:

- remove redundant first calculations of expressions
- local CSE handles later redundant calculations

Live variable analysis:

- only live variables (incl. live global CSEs) are stored on block exit

Uninitialized variable analysis:

- compile-time warnings of possible uses
- run-time checks to detect illegal uses (e.g., null ptrs)

Constant and copy propagation: uses reaching definitions, du-chains, copy propagation analysis

- propagate constant and variable assignments and simplify resulting expressions

---

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