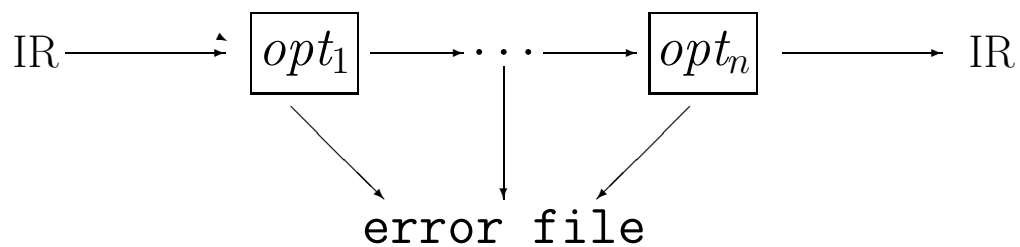


Optimizing Compilers



Many compilers include an optimizer

- often structured as a series of passes
- tries to improve code quality
- may repeat transformations several times

Code optimization

Definition

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

1. improve the running time of a program, *or*
2. decrease its space requirements, *or*
3. decrease the peak power and overall energy, *or*
4. some other metric (e.g.: heat dissipation)

Examples:

- *Classical* optimizations typically target speed. They make the number of operations actually executed to be less than or equal to the number expected by a naive programmer.
- Other optimizations improve the ordering of instructions or the locality of data and instructions (*reshaping optimizations*).

Code optimization

The point

- produce “improved” code, not “optimal” code
- can sometimes produce worse code
- range of improvement may vary (e.g.: speedup might be from 1.01 to 4 (*or more*))

How can optimizations improve code quality?

Machine independent transformations

1. replace a redundant computation with a reference
2. move evaluation to a less frequently executed place
3. specialize some general purpose code
4. find useless code and remove it
5. expose an opportunity for another optimization

Machine dependent transformations

1. replace a costly operation with a cheaper one
2. hide latency
3. replace a sequence of instructions with a more powerful one

Code optimization

From now on: mostly *performance (speed) optimizations*.

Examples:

- common subexpressions elimination
- dead code elimination
- invariant code motion
- constant propagation
- blocking for cache (e.g.: unroll and jam)
- vectorization
- parallelization

Code optimization

Three considerations arise in applying a transformation.

- *safety*
- *profitability*
- *opportunity*

Need a clear understanding of these issues.

- the literature often hides these issues
- every discussion *should* list them clearly

Safety

Fundamental question

Does applying the transformation change the results of executing the code?

yes \Rightarrow don't do it!

no \Rightarrow 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, dependence analysis,*)

Profitability

Fundamental question

Is there a reasonable expectation that applying the transformation will improve the code?

yes \Rightarrow do it!

no \Rightarrow don't do it

Compile-time estimation

- always profitable
- compute benefit *(may be hard)*

Opportunity

Fundamental question

Can we locate application sites efficiently?

yes \Rightarrow compilation time won't suffer

no \Rightarrow improvement had better be big

Issues

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

Code 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* \times *executions*
 - \Rightarrow *focus on loops* or other heavily executed code fragments (hot paths)
- want to minimize “bad” side effects such as 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
- + more opportunities for scheduling
- + more opportunities for register allocation
- instruction cache blowout
- register pressure

Opportunity: loops

Unrolling is easy to understand and perform.

Example

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)
```

Assumptions

- compiler performs only **local** optimizations
- row-major order storage scheme for arrays

Performance

- $2n^3$ *flops*, n^3 loop increments and branches
- each innermost iteration does 3 *loads*, 1 *store*, and 2 *flops*

This is the most overstudied example in the literature

Example

Matrix-matrix multiply

(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)
```

- unroll k loop by 4
- $2n^3$ flops, $\frac{n^3}{4}$ loop increments and branches
- each iteration does 9 loads, 1 store, and 8 flops
- memory traffic is better
 - $c(i,j)$ is reused (*put it in a register*)
 - $a(i,k)$ references are from cache
 - $b(k,j)$ is problematic

Example

Matrix-matrix multiply

(to improve traffic on b)

```
do j ← 1, n, 1
  do i ← 1, n, 4
    c(i,j) ← 0
    ...
    c(i+3,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)

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

Example

- interchanged i and j loops, unroll i loop, and fuse inner loops (unroll and jam)
- $2n^3$ flops, $\frac{n^3}{16}$ loop increments and branches
- first four assignments do 9 loads, 1 store, and 8 flops (total)
- remaining twelve assignments do 15 loads, 3 stores, and 24 flops (total)
- memory traffic is better
 - $c(i, j)$ is reused (register)
 - $a(i, k)$ references are from cache
 - $b(k, j)$ is reused (register)

Summary for 16 iterations:

	loads	stores	flops
original	48	8	32
unroll by 4	36	4	32
unroll and jam	24	4	32

Example

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 *(3× on SPARC)*
- performance model is tough
- check for safety of loop interchange and fusion is non-trivial (\Rightarrow *dependence analysis*)
- resulting code is *ugly* (pre-loops and post-loops needed for correctness)