A highly optimised introduction to compiler optimisations DD2488

Torbjörn Granlund, tg@gmplib.org

2014-04-27, 16:01

Q: What is ”optimisation”?
A: Semantics-preserving rewrites which we expect to improve things.

Q: What does ”optimisation” improve?
A: It improves execution performance or decreases code size. This note focuses on *speed* optimisation, although e.g., register allocation as a double effect.

Limitations:
We cannot expect optimisation to transform bubble-sort to quick-sort…

Ideal:
To optimise for speed, minimise

$$\sum w_i c_i$$

where w_i is the average repetition count of insn i and c_i is the cost of executing insn i, for a representable execution.

How does a CPU execute insns?

A modern CPU pipeline have these characteristics:

- Fetches **several instructions** per cycle
- Executes **several instructions** per cycle
- Does not execute things in strict assembly program order, but instead executes instructions as data becomes available (within a window of a few dozen instructions)
- Tries to guess branch outcome, using large internal tables of statistics, then executes along predicted path. Incorrect guesses result in ”undo” operations of incorrectly computed results.

CPU Pipeline characteristics vary between CPUs! Optimisations should take that into account.

This note is very terse. You are expected to get a fundamental understanding of each optimisation trick described herein. If this text is too terse, the key terms will readily lead to additional information, e.g., on Wikipedia.

1 Register allocation

What: Minimise memory (cache) loads and stores by allocating common quantities to CPU registers.

Register allocation is the process of assigning compiler internal scalars to physical CPU registers. This is usually considered as an optimisation, but it should be noted that some level of register allocation is necessary in a compiler, since many instructions require all or some operands to be in registers.

Two exits possible:

- All internal scalars (”TEMPs”) have gotten a real register – SUCCESS
- We could not assign all TEMP nodes – partial FAIL

In practice, partial FAIL is common. Compilers then allocate a stack frame slot for such scalars, and modify the code to access the stack slot. This process is known as *register spilling*.

To make the code valid, one may need to generate several additional instructions, for reading the stack slot into a physical register and/or to store it back again.

Register allocation requires control flow analysis and data flow analysis, in order to work out where each scalar is *live* and *dead*. From that information, an interference graph is constructed, each scalars is represented by a node, and two nodes are connected with an edge if they are *live at any point* simultaneously.

Register allocation with the interference graph $\equiv$ k-colouring.

1.1 Copy propagation

What: Use right-hand-side of copy operations hoping to kill copy.

Example: Transform

```
x = y
a = f(x)
```

into

```
x = y
a = f(y)
```

where f is some expression depending on y. This is done in the hope that x will then not be used, and thereby the copying x = y could be removed.

The above transformation is typically invalid if y has side-effects (although if one can preserve the side-effect order and number of y occurrences, it might still be possible).

1.2 Canonicalisation

What: Try to write a stmt/expression is some standard way

We often need to determine equivalence of (sub)expressions. In order to see that $a + b \equiv b + a$ we need some canonicalisation.

We might not want to permanently canonicalise output program (i.e., not transform $a + b$ to $b + a$), only canonicalise while examining an expression.

This is not an optimisation in itself, but it will assist in optimisation.

1.3 Dead code elimination

What: Remove unreachable code
What: Remove code producing unused value

Example 1: Transform

```
if (2 < 1)
  x = f(y)
```

into

```
[nothing]
```

We can perform this optimisation even if the dead code has side-effects, since it is never going to be executed.

Example 2: Transform

```
x = f(y)
```

into

```
[nothing]
```

if data flow analysis has determined that x dies in that statement. If f(y) has side-effects, one needs to preserve those, i.e., execute the side-effects but throw away the result of f.

1.4 Common subexpression elimination (CSE)

What: Locate (sub)expressions that are algebraically equivalent.

Almost all programs have lots of **common subexpressions**. Most are implicit, some are explicit.

Example: It might make sense to write

```
a = c + d + e
b = c + d + f
```

for clarity. Here c + d is a **common subexpression**.

The compiler should then transform the above code to

```
t = c + d
a = t + e
b = t + f
```

where t is a new temporary.

In this example,

```
a = b + c
b = a - d
c = b + c
d = a - d
```

it seems that b + c is a common subexpression. But since b was modified in between, it is not. The compiler needs to not just look for expression equivalence, but must make sure no (relevant) part of the subexpression had its variables modified.

In this example, the common subexpression might be less obvious.

```
for i = 0 ... n
  x[i] = 4 * x[i]
```

Assume x is an array of a 4-byte type. The expression x[i] then include an address expression x+i*4. Since we have two x[i] expressions in the statement, x+i*4 is a common subexpression. (Note that this subexpression might be implicit in source code as above, but will probably be more explicitly represented in IR code.)

1.5 Strength reduction, Algebraic rewrites

What: Replace expensive operations by cheaper operations

Examples:

- $(a)(b) + -(a)(c) \rightarrow (a)((b) + -(c))$
- $(a) - (a) \rightarrow 0$
- $0 \times (a) \rightarrow 0$
- $(a) \times 1 \rightarrow a$
- $4 \times (a) \rightarrow (a) << 2$
- $17 \times (a) \rightarrow t = ((a) << 4) + (a)$
- $(a)/3 \rightarrow \lfloor \lceil 2^{33}/3 \rceil \times (a)/2^{33} \rfloor$ (assuming 32-bit unsigned type)

1.6 Constant folding

What: Perform constant arithmetic at compile time.

To allow for more folding, try to propagate constants ”out” of expressions, where hopefully many constants are collected.

Turn
(a + 1) - b + c + (d + 1)
into
a - b + c + d + 2
(assuming language permits that for computation type).

We can sometimes do folding only after some other optimisations. Consider:

```
c = 17
...
if (c < 4711)
  some code
```

Assuming `c` does not change at ... we constant propagation turns the example into:

```
...
if (17 < 4711)
  some code
```

And finally we can fold:

```
...
some code
```

1.7 Motion of invariant code

What: Move **invariant code** out-of-loops

```
while (...)
  x = a * b - c / d
  y = ...
```

If a and b, or c and d are *invariant* in the loop, compute a * b or c / d before loop. Watch for side effects!

1.8 Motion of conditionally used code

What: Move conditionally used code to where it is used

Rewrite

```
x = expr
if (...)
  use x
else
  do not use x
```

to

```
if (...)
  x = expr
  use x
else
  do not use x
```

What about if expr has side effects?

If expr has side-effects, compute side-effects before ’if’ stmt, any other part of expr in ’if’.

1.9 Induction variable elimination

What: Loops may have (or get as result of compiler rewrites) multiple induction variables. Try to rewrite loop control to eliminate some of them.

1.10 Instruction scheduling

What: (1) Move producer insn away from consumer insn (”latency scheduling”)
What: (2) Adapt insn order to any CPU quirk (say, avoid putting multiply insns too close, if target CPU has limited multiply throughput.

Out-of-order CPUs (common!) need less scheduling then in-order CPUs (less common). See ”How does a CPU executes insns” above.

1.11 Caching into registers

What: Cache inherently memory-bound values into registers

Sometimes a field of a composite type (struct field, class variable) is read/written many times in a overviewable piece of code. Then the compiler can keep it in a register, with due regard to exceptions, side effect, etc.

1.12 Loop unrolling

What: Repeat a loop body and cut trip count

When a loop body is small, the bookkeeping overhead (increment induction variable, test termination condition, conditionally branch to loop start) can be significant.

The work/bookkeeping quotient can be increased by repeating the loop body k times, doing the bookkeeping only every k iteration.

Loop unrolling is usually only beneficial for plain `for` loops where the termination condition is not data-dependent.

1.13 Software pipelining

What: Extend *Instruction scheduling* over a loop branch.

Some loops have lots of data dependency, making (micro-) parallel execution difficult. Plain Instruction scheduling might not be able to move a single producer/consumer pair apart. Software pipelining schedules instructions across the branch point, making each apparent iteration work on two or mode subsequent iterations