Optimization

On **gcc**, the difference between optimized and non-optimized compilation is usually a factor 2.

Optimization is less sophisticated than most people think: It is mostly about the removal of obvious inefficiencies that are introduced by the translation algorithm.

Although it may be possible in principle to use computer algebra or automated deduction to improve programs, I am not aware of any working attempts.

There exists a lot of literature, but not much concrete.

Optimization Algorithms

1. Remove computations of values that are not used. (Dead code elimination).
2. Remove unreachable code.
3. Detect constant expressions.
4. Detect recomputed expressions.
5. Decide which values, stored in memory, could be stored in a local variable.

The different optimization tasks are related to each other in very complicated ways.

For example, code may become unreachable due to the fact that the condition of an **if** statement is constant.

Task 5, deciding which locations can be stored in local variables, is very hard.

Note that we are not deciding about machine registers yet. The reason that we need local variables is because of symbolic evaluation.

Example

Consider the following code fragment:

```
strcopy( const char* p, char* q )
{
   unsigned int i = 0;
   while( p[i] != 0 )
   {
      q[i] = p[i];
      i ++ ;
   }
   q [i] = 0;
}
```

Example (2)

The translation algorithm could produce the following intermediate code:

```
strcpy:
func( void; pointer( const( char )), pointer( char ) )

// Recover positions of local variables in memory:

$P = $SP + sizeof( returnaddress );
$Q = $P + sizeof( pointer( const( char )) );

$I = $SP;
$SP = $SP - sizeof( int );
   // Create local variable i in memory.
```

Example (3)

```
L1:
   // Evaluate p[i] != 0?

   @A = [ $I ];  // Inline CC.
   @B = @A * sizeof( char );
   $C = [ $P ];  // Inline CC.
   $D = $C + @B;
   @E = [ $D ];  // Finally, the character! (using ICC)
   @B = ( @E != 0 );
   jumpfalse L2;
```

Example (4)

```
// Evaluate q[i] as reference:
@A1 = [ $I ];
@B1 = @A1 * sizeof( char );
$C1 = [ $Q ];
$D1 = $C1 + @B1;  // now D1 contains q[i].

// Evaluate p[i] as object:
@A2 = [ $I ];
@B2 = @A2 * sizeof( char );
$C2 = [ $P ];
$D2 = $C2 + @B2;
@E2 = [ $D2 ];

[ $D1 ] = @E2;  // Character is copied!
```

Example (5)

```
// Do i ++ :

@R = [ $I ];
@S = @R;     // Copy result (will not be used)
@T = @S + 1;
[ $I ] = @T; // Write result back.

goto L1;
```

Example (6)

```
L2:
   // Evaluate q[i] as reference:

   @A3 = [ $I ];
   @B3 = @A3 * sizeof( char );
   $C3 = [ $Q ];
   $D3 = $C3 + @B3;

   @U = 0;
   [ $D3 ] = @U;

   $SP = $SP + sizeof( int ); // Remove variable i.
   return;
```

Possible Optimizations

- **char** probably has size 1. Multiplication by 1 can be removed.
- $p[i]$ is calculated (and looked up) twice in the loop.
- If we would not be copying characters, but something with size $\neq 1$, we could reuse the multiplication.
- Variable i could be put in a register.

Cost

Optimization means that we want to reduce the cost of executing the program. There are several meaningful notions of cost:

1. Execution time of the program.
2. Size of the program.
3. Energy use of the program.

Nearly always, execution time is the most important goal.
(inlining, loop unfolding trade space for time.)

Redundant Expressions

Redundant expressions occur when expressions are recomputed.

Definition An expression E is redundant if it has already been computed on every path that leads to E.

Redundant Expressions (2)

$m = 2 + y;$

$n = y;$

$k = 2 + n;$

can be replaced by

$m = 2 + y;$

$k = m;$

Redundant Expressions (3)

In the example

```
   unsigned int i = 0;
loop:
   if( *(p+i) == 0 ) goto end;

   *(q+i) = *(p+i);
   i = i + 1;
   goto loop;
end:
   return;
```

the second `*(p+i)` is redundant.

Redundant Expressions (4)

When is an expression redundant?

$a := *(p + i);$
$i := i + 1;$
$b := *(p + i)$ (obviously not)

$a := *(p + i);$
$*(p + i) := 44;$
$b := *(p + i);$ (obviously not, but one could reuse 44)

Redundant Expressions (5)

There exists a quite sophisticated field of automated theorem proving, but practical code is so big that only efficient (close to linear) algorithms have been used in practice: (but maybe this will change)

Instead, one builds a container of normalized available expressions. (Usually a hash map.)

Normalization

We will create a set of local variables $\mathcal{X}$ and a set of rewrite rules $\mathcal{R}$, which maps expressions to local variables. Initialize $\mathcal{X} := \{\ \}$.

First we give a function $\text{NORM}(E, \mathcal{X}, \mathcal{R})$ for normalizing expressions. If necessary, the function extends the parameters $\mathcal{X}$ and $\mathcal{R}$. The result of $\text{NORM}(E, \mathcal{X}, \mathcal{R})$ is always an input variable, a variable in $\mathcal{X}$, or a constant.

- For an input variable $x \in \mathcal{X}$, $\quad \text{NORM}(x, \mathcal{X}, \mathcal{R}) = x$.
- For a non-input variable v, find a rule $v \Rightarrow x$ in $\mathcal{R}$. If no such rule exists, then the variable is unitialized. Otherwise $\text{NORM}(v, \mathcal{X}, \mathcal{R}) = x$.
- For a constant, $\text{NORM}(c, \mathcal{X}, \mathcal{R}) = c$.

Normalization (2)

- For an expresssion $f(t_1, \ldots, t_n)$, first recursively compute $x_1 := \text{NORM}(t_1, \mathcal{X}, \mathcal{R}), \ldots, x_n := \text{NORM}(t_n, \mathcal{X}, \mathcal{R})$.

  If there is a rule $f(x_1, \ldots, x_n) \Rightarrow x$ in $\mathcal{R}$, then $\text{NORM}(\ f(t_1, \ldots, t_n),\ \mathcal{X}, \mathcal{R}) = x$.

  Otherwise, create a new variable x, add it to $\mathcal{X}$ and add the rule $f(x_1, \ldots, x_n) \Rightarrow x$ to $\mathcal{R}$. Now $\text{NORM}(\ f(t_1, \ldots, t_n),\ \mathcal{X}, \mathcal{R}) = x$.

$\mathcal{R}$ can be implemented very efficient with hashing or some other form of indexing.

Normalization (3)

Using NORM, the normalization procedures processes the assignments. For each assignment $v := E$, do the following:

- Compute $x = \text{NORM}(E, \mathcal{X}, \mathcal{R})$. Add a rule $v \Rightarrow x$ to $\mathcal{R}$.

Normalization (4)

When the algorithm has processed all assignments, one can reconstruct the expressions for the output variables of the block. (These are the variables that are looked at later on a path that originates from the block)

The $x \in \mathcal{X}$ will become local variables.

Normalization (5)

In practice, one should attempt to normalize expressions before analyzing:

- Replace $X + 0 \Rightarrow X, \quad 0 + X \Rightarrow X$.
- Replace $X \times 1 \Rightarrow X, \quad X \times 0 \Rightarrow 0$, etc.
- Sort long multiplications and additions. (For example, first numbers, next by index.)

Normalization (6)

It remains to generate the simplification of the block. The simplification is a sequence of assignments, but without recomputations.

Let $v_1, \ldots, v_n$ be the output variables of the block. (The variables that are used on a path originating from the block.)

Replace each v_i by $\mathrm{NORM}(v_i, \mathcal{X}, \mathcal{R})$ on every path that originates from the block.

Normalization (7)

Put

$$\text{SIMP} := (\ ).$$

(the result of simplifying the block.)

$$X_d := \{\ \}.$$

(the intermediate variables that have an assignment in SIMP)

$$X_n := \{\text{NORM}(v_i, \mathcal{X}, \mathcal{R}) \mid \text{NORM}(v_i, \mathcal{X}, \mathcal{R}) \text{ is not a constant or input variable of the block } \},$$

(the intermediate variables that need to be defined.)

Normalization (8)

While $X_n \backslash X_d$ is not empty, select an x (with maximal weight) from $X_n \backslash X_d$, and call ASSIGN(x).

The procedure ASSIGN(x) recursively assigns the variables that are needed to obtain a definition of x. (It is assumed that x is has no assignment when ASSIGN(x) is called.)

- Lookup the rule of form $(\ \ f(x_1, \ldots, x_n) \Rightarrow x\ \ ) \in \mathcal{R}$ that defines x.
- As long as one of the $x_1, \ldots, x_n$, that is not a constant nor an input variable, does not occur in X_d, select the x_i with greatest weight among those. Call ASSIGN(x_i).
- Append the assignment $x := f(x_1, \ldots, x_n)$; to SIMP.

  Put $X_d := X_d \cup \{x\}$.

Example

(x, y are input variables.)

$a := x + y;$

$b := x + 1 + y;$

$c := 17;$

$d := x + y + c;$

$e := x + z;$

(Later, a, b, d are used)

Static Single Assignment Form

- The normalization algorithm has problems when variables are reused:

  $a := (x + y + z);$

  $a := a + a;$

  $b := (x + y + z);$

  $\Rightarrow$ Rename variables in advance.

- It renames its output variables.

Renaming is problematic when paths merge.

Static Single Assignment Form

Definition: Let $\mathcal{G}$ be the flow graph of a procedure. (We assume that this is the basic block of analysis.)

We call P in static single assignment form if each variable that occurs in P is either an input variable, or has exactly one point of assignment. (which then is an initialization.)

In order to reunate variables in different branches, a special function is used, which is called the ϕ function. (It seems to mean 'phoney')

It is difficult to give an intuitive meaning to the ϕ function, but one could define $\phi(x_1, \ldots, x_n)$ as: From those x_i that have a value, select the value of the variable that was assigned most recently.

SSA

Consider the procedure:

```
   int fact( N ):

   R = 1;
loop:
   if( N == 0 ) goto end;
   R = R * N;
   N = N - 1;
   goto loop;

end:
   return R;
```

SSA

The SSA is:

```
   int fact( N1 )   // Treated as assignment to N.
   R1 = 1;
loop:
   // This is a merging point:

   R2 = Phi( R1, R3 );
   N2 = Phi( N1, N3 );
   if( N2 == 0 ) goto end;
   R3 = R2 * N2;
   N3 = N2 - 1;
   goto loop;

end:
   return R2;
```

Computing SSA

There are in principle two strategies for placing ϕ functions, but the first one makes it is difficult to remove the ϕ functions again, so we use the second.

1. Place a ϕ just before every point where a variable is used.
2. Place a ϕ at each merging point, for each variable that is 'alive' at this point. (has a path towards a point where it it used.)

The algorithm for constructing SSA form consists of two stages:

1. Insert ϕ functions of form $v = \phi(\emptyset)$, where necessary.
2. Rename different versions of variable v by v_i, using different i and update the ϕ functions.

Inserting ϕ functions

If there are two assignments $v = t_1$ and $v = t_2$ in different nodes of the flow graph, and there exist two paths without repeated nodes towards a common node in which v is used, and the first node in which these paths meet is N, and N does not contain a ϕ function for v yet, then add $v = \phi(\emptyset)$ to node N, before any other statements in N.

(One can also create a new node N' in front of N, and put the ϕ assignment there.)

Renaming the Variables

Variable renaming is done by a recursive algorithm. When a node is visited, it is marked, so that it will be not visited again. Initially, all nodes are unmarked.

The marking algorithm uses a matching Θ which matches each variable $\Theta(v)$ to a unique version v_i. Initially, we have $\Theta(v) = v_1$ for all input variables v, and $\Theta(w) = \perp$ for all other variables. ($\perp$ is a special value denoting that the variable is not initialized.)

We assume that each node in the flow graph $\mathcal{G}$ contains at most one statement. (Otherwise, the node can be split.)

We start by calling **rename**(Θ, N) for the starting node N of the flow graph $\mathcal{G}$.

rename(Θ, N).

- If node N contains a ϕ assignment of form $v = \phi(V)$, then add $\Theta(v)$ to V.
- If node N is marked, then we are done at this point. If node N was not marked, then we mark it now, and continue.
- If variable v is used in node N, but not in a ϕ function, then we replace the occurrence by $\Theta(v)$.
- If node N contains an assignment to some variable v (with a ϕ function or some other function), then let $w = \Theta(v)$, and assign $\Theta(v) = v_i$, using a new version number i for v. Replace the assigned variable by v_i.
- Recursively call **rename**(Θ, N_j) for all nodes N_j that are reachable from N in one step.
- If Θ was changed two steps back, then restore $\Theta(v) = w$.

Removal of ϕ Functions

ϕ functions cannot be efficiently executed. One could use time stamps, but this is not practical.

One needs a method to get rid of the ϕ functions, when all optimizations are complete.

Removal of ϕ functions (2)

In many cases, ϕ functions can be eliminated by merging the variables. (For example in the factorial function given earlier.)

But this often fails in optimized code:

```
      X = 1;
   loop:
      Y = X;
      X = X + 1;
      if( something ) goto loop;

      return Y;
```

Lost Copy Problem

Optimization would remove the assignment `Y=X`, and the resulting code (in SSA) would be:

```
    X1 = 1;
loop:
    X2 = Phi(X1,X3);
    X3 = X2 + 1;
    if( something ) goto loop;
return X2;
```

If one would simply merge the variables `X1,X2,X3`, the return-statement would return the wrong copy of `X`.

Replacement of ϕ Functions by Assignments

When ϕ functions are positioned as early as possible, (our algorithm did this), it is easy to replace ϕ functions by assignments.

If node N contains a ϕ assignment $w = \phi(v_1, \ldots, v_n)$, then let $N_1, \ldots, N_n$ be the nodes from which N is reachable in one step.

In each of the branches (N_i, N), insert a new node with an assignment $w = v_i$.

Replacement of ϕ Functions by Assignments (2)

If one has a node with multiple ϕ assignments, then one must be careful for the swapping problem.

Assume that the ϕ assignment has form:

$$\begin{aligned} w_1 &= \phi(v_{1,1}, \ldots, v_{1,n}) \\ w_2 &= \phi(v_{2,1}, \ldots, v_{2,n}) \\ &\ldots \\ w_m &= \phi(v_{m,1}, \ldots, v_{m,n}). \end{aligned}$$

One must insert assignments that assign $(w_1, \ldots, w_m) = (v_{1,j}, \ldots, v_{m,j})$ between N_j and N, but be careful with overlapping variables.

Replacement of ϕ Functions by Assignments (3)

```
   a1 = 1;
   b1 = 2;
loop:
   a2 = phi( a1, b2 );
   b2 = phi( b1, a2 );
      // Block with two phi's.
   if( something ) goto loop;

end:
   return a2;
```

Variable Conflicts

We want to answer the following question: When is it possible to merge two variables v_1, v_2?

Definition: Two variables v_1 and v_2 are in conflict with each other if there exists a path of form:

$v_1 = ...$

$\ldots$

$v_2 = ...$

$... = ... \ v_1 \ ....$

In words: There exists a path through the flow graph, starting with an assignment of v_1, ending with a use of v_1, and somewhere on this path, v_2 is assigned. If such path exists, then v_1, v_2 cannot be merged, because the assignment $v_2 = ...$ would overwrite the value of v_1.

Simplification of SSA

1. As long as there exist two variables v_1, v_2 in the flow graph that are of the same type and not in conflict, substitute $v_1 := v_2$. (After this, the code it is not in SSA anymore, but we want to remove the ϕ functions anyway.)

   When more than one such pair exists, give preference to a pair v_1, v_2 that is connected by a ϕ-function. (One can also restrict simplifiation to variables that are connected by a ϕ-function.)

2. Remove repeated arguments in ϕ functions.

3. Remove ϕ functions of form $v = \phi(v)$.

(This algorithm is quadratic, but it can be made linear.)

Removal of ϕ functions (3)

This gives a final algorithm:

1. Merge as many variables as possible.
2. Replace the remaining ϕ functions by assignments.
3. Remove identity assignments of form $v = v$.

Detection of Constants

Consider

```
for( unsigned int i = 0; i < n; ++ i )
{
   for( unsigned int j = 0; j < n; ++ i )
   {
      M[i][j] = 0.0;
   }
}
```

The address calculation $M + i$ can be reused in the inner loop.

Detection of Constants (2)

As with redundancy elimination, the algorithm works by symbolic evaluation of the flowgraph. One can apply the algorithm on the complete prodecure, or separately on each strongly connected component.

We will assume that the flow graph is in SSA normal form. For simplicity, we will assume that every conditional statement bases its choice on a boolean variable.

The main idea of the algorithm is to assign to each variable a set of possible values. For each variable v, we will collect the set of possible values in $\Theta(v)$.

Let $\mathcal{G}$ be the flow graph. Let $\mathcal{G}' \subseteq \mathcal{G}$ be the component that we are analyzing. Let V be the variables that occur in $\mathcal{G}'$. let $W \subseteq V$ be the variables that have an assignment in $\mathcal{G}'$.

Detection of Constants (3)

The assignment sets Θ are initialized as follows:

- For a variable $v \in V \backslash W$, put $\Theta(v) = \{v\}$.
- For a variable in $v \in W$, put $\Theta(v) = \emptyset$.

After initialization, the sets Θ are saturated as follows:

- If the flow graph $\mathcal{G}'$ has an assignment $v = f(v_1, \ldots, v_n)$, and
- there exist values $z_1 \in \Theta(v_1), \ldots, z_n \in \Theta(v_n)$,
- for every v_i, for every conditional statement with boolean variable b on the path from the assignment statement for v_i to the assignment statement for v, (the start of the block if $v_i \in V \backslash W$) we have $\mathbf{t} \in \Theta(b)$ if the path selects **true**, or $\mathbf{f} \in \Theta(b)$ if the path selects **false**, or a symbolic expression in $\Theta(b)$.

Detection of Constants (4)

If all this is true, and $\mathbf{Eval}(f(z_1, \ldots, z_n)\ ) \notin \Theta(v)$, we put

$$\Theta(v) = \Theta(v) \cup \{\mathbf{Eval}(f(z_1, \ldots, z_n))\}.$$

The algorithm can be implemented 'change driven': Whenever something changes in some $\Theta(v)$, one needs to check only the assignments that use v.

In the current form, the algorithm will not terminate. The reason for this is that some of the $\Theta(v)$ may be infinite.

In order to make the algorithm terminate, we add the following rule: If, for some $v \in V$, we have $|\Theta(v)| \geq 2$, we put $\Theta(v) = \mathbf{inf}$. We have $\mathbf{inf} \cup \{x\} = \{x\} \cup \mathbf{int} = \mathbf{inf}$.

One could remove the size restriction for enumeration types, in order to detect unreachable **cases** in **switch** statements.

Detection of Constants (5)

It remains to define the evaluation rules Eval. A few evaluation rules:

$0 \times A$	$\Rightarrow$	0
$t = t$	$\Rightarrow$	$\mathbf{t}$
c_1 op c_2	$\Rightarrow$	can be computed
		if c_1, c_2 are known.
$\mathbf{f}$ and A	$\Rightarrow$	$\mathbf{f}$
$\mathbf{t}$ or A	$\Rightarrow$	$\mathbf{t}$
etc.		

AC operators (associative commutative) should be sorted with constant part before non-constant part, in order to improve the chance of partial evaluation.

Problems with Analysis

- Primitive types only: It is almost impossible to analyze objects.
- Code involving pointers is problematic. There seems to exist no theoretical solution.
- It will not 'take out' variables from arrays. (See next slide)
- Outline function calls are completely blocked from optimization.