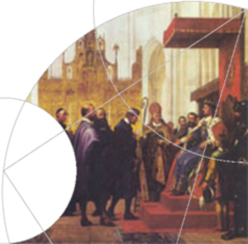
Optimizations

Cosmin E. Oancea
cosmin.oancea@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

December 2012 Compiler Lecture Notes



Structure of a Compiler

Program text

↓

Lexical analysis

↓

Symbol sequence

↓

Syntax analysis

↓

Syntax tree

↓

Typecheck

↓

Syntax tree

↓

Intermediate code generation → Interm code + Optimizations

↑

Machine code generation

↑

Symbolic machine code

↑

Register allocation

↑

Ditto with named registers

↑

Assembly and linking

↑

Binary machine code

1. Data-Flow Analysis
   - Common-Subexpression Elimination
   - Jump-to-Jump Elimination
   - Index-Checking Elimination
2. Loop Optimizations
   - Hoisting Loop-Invariant Computation
   - Prefetching
3. Function Calls
   - Inlining
   - Tail-Call Optimization
4. Specialization

Data-Flow Analysis

Global analysis is used to derive information that can drive optimizations. Example: *Liveness analysis.*

Information can flow forwards or backwards through the program. Typical Structure:

- Successor/predecessor on basic blocks ($succ[B_i]$ or $pred[B_i]$).
- Define $gen[i]$ and $kill[i]$ sets.
- Define equations for $in[i]$ and $out[i]$.
- Initialize $in[i]$ and $out[i]$.
- Iterate to a fix point.
- Use $in[i]$ or $out[i]$ for optimizations.

Common-Subexpression elimination

Goal: remove redundant computations.

Example: statement `a[i] := a[i]+1` translates to:

```
t1 := 4*i
t2 := a+t1
t3 := M[t2]
t4 := t3+1
t5 := 4*i
t6 := a+t5
M[t6] := t4
```

Potential for optimization:

- Term `4*i` is computed twice.
- `a+t1` and `a+t5` are equal, since `t1`=`t5`.

Available-Assignments Analysis

Instruction i	$gen[i]$	$kill[i]$
LABEL l	$\emptyset$	$\emptyset$
$x := y$	$\emptyset$	$assg(x)$
$x := k$	$\{x := k\}$	$assg(x)$
$x :=$ **unop** y where $x \neq y$	$\{x :=$ **unop** $y\}$	$assg(x)$
$x :=$ **unop** x	$\emptyset$	$assg(x)$
$x :=$ **unop** k	$\{x :=$ **unop** $k\}$	$assg(x)$
$x := y$ **binop** z where $x \neq y$ and $x \neq z$	$\{x := y$ **binop** $z\}$	$assg(x)$
$x := y$ **binop** z where $x = y$ or $x = z$	$\emptyset$	$assg(x)$
$x := y$ **binop** k where $x \neq y$	$\{x := y$ **binop** $k\}$	$assg(x)$
$x := x$ **binop** k	$\emptyset$	$assg(x)$
$x := M[y]$ where $x \neq y$	$\{x := M[y]\}$	$assg(x)$
$x := M[x]$	$\emptyset$	$assg(x)$
$x := M[k]$	$\{x := M[k]\}$	$assg(x)$
$M[x] :=$ y	$\emptyset$	$loads$
$M[k] :=$ y	$\emptyset$	$loads$
GOTO l	$\emptyset$	$\emptyset$
IF x **relop** y THEN l_t ELSE l_f	$\emptyset$	$\emptyset$
$x :=$ CALL $f(args)$	$\emptyset$	$assg(x)$

$assg(x)$: Assignements that use x on the left or right-hand sides,
$loads$: Statements of form $y := M[\cdot]$.

Example for Available Assignments

1: $i := 0$
2: $a := n * 3$
3: `IF` $i < a$ `THEN` *loop* `ELSE` *end*
4: `LABEL` *loop*
5: $b := i * 4$
6: $c := p + b$
7: $d := M[c]$
8: $e := d * 2$
9: $f := i * 4$
10: $g := p + f$
11: $M[g] := e$
12: $i := i + 1$
13: $a := n * 3$
14: `IF` $i < a$ `THEN` *loop* `ELSE` *end*
15: `LABEL` *end*

i	$pred[i]$	$gen[i]$	$kill[i]$
1		1	1, 5, 9, 12
2	1	2	2
3	2		
4	3, 14		
5	4	5	5, 6
6	5	6	6, 7
7	6	7	7, 8
8	7	8	8
9	8	9	9, 10
10	9	10	10
11	10		7
12	11		1, 5, 9, 12
13	12	2	2
14	13		
15	3, 14		

Note: Assignment 2 and assignment 13 both represented by 2.

Fix-Point Iteration

$$out[i] = gen[i] \cup (in[i] \setminus kill[i]) \quad (1)$$

$$in[i] = \bigcap_{j \in pred[i]} out[j] \quad (2)$$

Initialized to the set of all assignments, except for $in[1] = \emptyset$.

	Initialisation		Iteration 1		Iteration 2	
i	*in*[*i*]	*out*[*i*]	*in*[*i*]	*out*[*i*]	*in*[*i*]	*out*[*i*]
1		1, 2, 5, 6, 7, 8, 9, 10		1		1
2	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1	1, 2	1	1, 2
3	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2	1, 2	1, 2	1, 2
4	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2	1, 2	2	2
5	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2	1, 2, 5	2	2, 5
6	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5	1, 2, 5, 6	2, 5	2, 5, 6
7	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6	1, 2, 5, 6, 7	2, 5, 6	2, 5, 6, 7
8	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7	1, 2, 5, 6, 7, 8	2, 5, 6, 7	2, 5, 6, 7, 8
9	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8	1, 2, 5, 6, 7, 8, 9	2, 5, 6, 7, 8	2, 5, 6, 7, 8, 9
10	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9	1, 2, 5, 6, 7, 8, 9, 10	2, 5, 6, 7, 8, 9	2, 5, 6, 7, 8, 9, 10
11	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 8, 9, 10	2, 5, 6, 7, 8, 9, 10	2, 5, 6, 8, 9, 10
12	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 8, 9, 10	2, 6, 8, 10	2, 5, 6, 8, 9, 10	2, 6, 8, 10
13	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	2, 6, 8, 10	2, 6, 8, 10	2, 6, 8, 10	2, 6, 8, 10
14	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	2, 6, 8, 10	2, 6, 8, 10	2, 6, 8, 10	2, 6, 8, 10
15	1, 2, 5, 6, 7, 8, 9, 10	1, 2, 5, 6, 7, 8, 9, 10	2	2	2	2

Iteration 3 = iteration 2.

Used in Common-Subexpression Elimination

The computation 5: $b := i * 4$ is available at 9: $f := i * 4$.

1: $i := 0$
2: $a := n * 3$
3: IF $i < a$ THEN *loop* ELSE *end*
4: LABEL *loop*
5: $b := i * 4$
6: $c := p + b$
7: $d := M[c]$
8: $e := d * 2$
9: $f := i * 4$
10: $g := p + f$
11: $M[g] := e$
12: $i := i + 1$
13: $a := n * 3$
14: IF $i < a$ THEN *loop* ELSE *end*
15: LABEL *end*

Used in Common-Subexpression Elimination

The computation 5: $b{:=}i{*}4$ is available at 9: $f{:=}i{*}4$.

1: $i := 0$
2: $a := n * 3$
3: IF $i < a$ THEN *loop* ELSE *end*
4: LABEL *loop*
5: $b := i * 4$
6: $c := p + b$
7: $d := M[c]$
8: $e := d * 2$
9: $f := b$
10: $g := p + f$
11: $M[g] := e$
12: $i := i + 1$
13: $a := a$
14: IF $i < a$ THEN *loop* ELSE *end*
15: LABEL *end*

Used in Common-Subexpression Elimination

The computation 5: $b{:=}i{*}4$ is available at 9: $f{:=}i{*}4$.

```
1:  i := 0
2:  a := n * 3
3:  IF i < a THEN loop ELSE end
4:  LABEL loop
5:  b := i * 4
6:  c := p + b
7:  d := M[c]
8:  e := d * 2
9:  f := b
10: g := p + f      ←   will not be eliminated
11: M[g] := e
12: i := i + 1
13: a := a
14: IF i < a THEN loop ELSE end
15: LABEL end
```

1. Data-Flow Analysis
   - Common-Subexpression Elimination
   - Jump-to-Jump Elimination
   - Index-Checking Elimination
2. Loop Optimizations
   - Hoisting Loop-Invariant Computation
   - Prefetching
3. Function Calls
   - Inlining
   - Tail-Call Optimization
4. Specialization

Jump-to-Jump Elimination

Avoid successive jumps: `[... GOTO` l_1`, ..., LABEL` l_1`, GOTO` l_2 `...]`.

instruction	*gen*	*kill*
LABEL l	$\{l\}$	$\emptyset$
GOTO l	$\emptyset$	$\emptyset$
IF c THEN l_1 ELSE l_2	$\emptyset$	$\emptyset$
any other	$\emptyset$	the set of all labels

$$in[i] = \begin{cases} gen[i] \setminus kill[i] & \text{if } out[i] \text{ is empty} \\ out[i] \setminus kill[i] & \text{if } out[i] \text{ is non-empty} \end{cases} \quad (3)$$

$$out[i] = \bigcap_{j \in succ[i]} in[j] \quad (4)$$

A jump i: GOTO l can be replaced with i: GOTO l', if $l' \in in[i]$.

1. Data-Flow Analysis
   - Common-Subexpression Elimination
   - Jump-to-Jump Elimination
   - Index-Checking Elimination
2. Loop Optimizations
   - Hoisting Loop-Invariant Computation
   - Prefetching
3. Function Calls
   - Inlining
   - Tail-Call Optimization
4. Specialization

Index-Check Elimination

Checks if `i` is within the array-size bounds when used in `a[i]`.

Idea: use `IF-THEN-ELSE` to check bounds and analyze whether the condition reduces statically to *true* or *false*.

Example: If a's lowest/highest index is 0/10, translate
`for i:=0 to 9 do a[i]:=0;` to:

```
i := 0
LABEL for1
IF i ≤ 9 THEN for2 ELSE for3
LABEL for2
IF i < 0 THEN error ELSE ok1
LABEL ok1
IF i > 10 THEN error ELSE ok2
LABEL ok2
t := i * 4
t := a + t
M[t] := 0
i := i + 1
GOTO for1
LABEL for3
```

Inequalities

Collect inequalities of the form $p \leq q$ and $p < q$, where p and q are either variables or constants.

In order to ensure a finite number of inequalities, use an universe Q of inequalities derived from program's condition(al)s. For example:

- $when(x < 10) = \{x < 10\}$, $whennot(x < 10) = \{10 \leq x\}$.
- $when(x = y) = \{x \leq y,\ y \leq x\}$, $whennot(x = y) = \emptyset$.

Our example program provides the following universe:

$$Q = \{i \leq 9,\ 9 < i,\ i < 0,\ 0 \leq i, 10 < i,\ i \leq 10\}$$

Fixpoint-iteration computes $in[i] =$ the set of inequalities (from Q) that are true (hold) at the beginning of instruction i.

Equations for Inequalities

$$in[i] = \begin{cases} \bigcap_{j \in pred[i]} in[j] & \\ \qquad \text{if } pred[i] \text{ has more than one element} & \\ in[pred[i]] \cup when(c) & \\ \qquad \text{if } pred[i] \text{ is } \texttt{IF}\ c\ \texttt{THEN}\ i\ \texttt{ELSE}\ j & \\ in[pred[i]] \cup whennot(c) & \\ \qquad \text{if } pred[i] \text{ is } \texttt{IF}\ c\ \texttt{THEN}\ j\ \texttt{ELSE}\ i & \\ (in[pred[i]] \setminus conds(Q,x)) \cup equal(Q,x,p) & \\ \qquad \text{if } pred[i] \text{ is of the form } x := p & \\ in[pred[i]] \setminus upper(Q,x) & \\ \qquad \text{if } pred[i] \text{ is of the form } x := x + k \text{ where } k \geq 0 & \\ in[pred[i]] \setminus lower(Q,x) & \\ \qquad \text{if } pred[i] \text{ is of the form } x := x - k \text{ where } k \geq 0 & \\ in[pred[i]] \setminus conds(Q,x) & \\ \qquad \text{if } pred[i] \text{ is of a form } x := e \text{ not covered above} & \\ in[pred[i]] & \\ \qquad \textit{otherwise} & \end{cases}$$

$conds(Q,x)$: inequalities in x

$upper(Q,x)$: inequalities of form $x < p$ or $x \leq p$

$lower(Q,x)$: inequalities of form $p < x$ or $p \leq x$

$equal(Q,x,p)$: inequalities from Q, which are consequences of $x = p$

Limitations of Data-Flow Analysis

Can never be exact:

- many analysis problems are undecidable, i.e., one cannot solve them accurately.
- Tradeoff between efficient computation and precision.
- Use conservative approximation: optimize only when you are sure the assumptions hold.

1. Data-Flow Analysis
   - Common-Subexpression Elimination
   - Jump-to-Jump Elimination
   - Index-Checking Elimination
2. Loop Optimizations
   - Hoisting Loop-Invariant Computation
   - Prefetching
3. Function Calls
   - Inlining
   - Tail-Call Optimization
4. Specialization

Hoisting Loop-Invariant Computation

A term is loop invariant if it is computed inside a loop but has the same value at each iteration.

Solution: Unroll the loop once and do common-subexpression elimination.

The loop-invariant term is now computed in the unrolled part and reused in the subsequent loop. Example:

```
while (c) {
  body
}
```

$\longrightarrow$

```
if (c) {
  body;
  while (c) {
    body
  }
}
```

```
do
  body
while (c)
```

$\longrightarrow$

```
body;
while (c) {
  body
}
```

Disadvantage: Code size.

Memory Prefetching

```
sum = 0;
for (i=0, i<100000; i++) {
  sum += a[i];
}
```

Problem: the array does not fit in the cache so we constantly wait (for IO). Some architectures offer a *prefetch* instruction that downloads from memory into cache and is safe under addressing errors:

```
sum = 0;
for (i=0, i<100000; i++) {
  if (i&3==0) prefecth(a[i+32]);
  sum += a[i];
}
```

NB!: We assumed that the cache-line size is 4 words.

We can deploy loop body to avoid testing.

1. Data-Flow Analysis
   - Common-Subexpression Elimination
   - Jump-to-Jump Elimination
   - Index-Checking Elimination
2. Loop Optimizations
   - Hoisting Loop-Invariant Computation
   - Prefetching
3. Function Calls
   - Inlining
   - Tail-Call Optimization
4. Specialization

Inlining

A function call: x = f(exp_1,...,exp_n);

where f is declared as:

```
type0 f(type1 x1,...,typen xn)
{
  body
  return(exp);
}
```

can be replaced with:

```
{
  type1 x1 = exp1;
  ...
  typen xn = expn;
  body
  x = exp;
}
```

Variables need to be renamed whenever necessary, e.g., rename $x_1 \ldots x_n$ as needed.

Tail-Call Optimization

A tail call is a call happening just before returning from the (current) function, `f`, e.g., `return(g(x,y));`

We use the following observations:

- None of `f`'s variables are *live* after the call.
- If `f`'s epilogue is empty (except for the return jump), then `g` can return directly to `f`'s return address.
- Since `f`'s activation record contains nothing useful at this point, we can reuse the space for `g`'s activation record.

Hence the program is more efficient in both runtime and memory space.

Tail-call optimization is very important for functional languages as it makes tail recursion as efficient as loops.

Tail-Call Example

`return(g(x,y));` via stack-based *caller saves*.

$M[FP + 4 * m + 4] := R0$
$\ldots$
$M[FP + 4 * m + 4 * (k + 1)] := Rk$
$FP := FP + framesize$
$M[FP + 4] := \texttt{x}$
$M[FP + 4 * n] := \texttt{y}$
$M[FP] := returnaddress$
GOTO g
LABEL *returnaddress*
$result := M[FP + 4]$
$FP := FP - framesize$
$R0 := M[FP + 4 * m + 4]$
$\ldots$
$Rk := M[FP + 4 * m + 4 * (k + 1)]$
$M[FP + 4] := result$
GOTO $M[FP]$

Tail-Call Example

`return(g(x,y));` via stack-based *caller saves*.

$M[FP + 4 * m + 4] := R0$ $\longleftarrow$ Eliminated since no variables are live
$\dots$
$M[FP + 4 * m + 4 * (k + 1)] := Rk$
$FP := FP + framesize$
$M[FP + 4] := \mathtt{x}$
$M[FP + 4 * n] := \mathtt{y}$
$M[FP] := returnaddress$
GOTO g
LABEL *returnaddress*
$result := M[FP + 4]$
$FP := FP - framesize$
$R0 := M[FP + 4 * m + 4]$
$\dots$
$Rk := M[FP + 4 * m + 4 * (k + 1)]$
$M[FP + 4] := result$
GOTO $M[FP]$

Tail-Call Example

`return(g(x,y));` via stack-based *caller saves*.

$FP := FP + framesize$ $\longleftarrow$ Eliminated when we recycle activation records
$M[FP + 4] := \mathtt{x}$
$M[FP + 4 * n] := \mathtt{y}$
$M[FP] := returnaddress$
GOTO g
LABEL *returnaddress*
$result := M[FP + 4]$
$FP := FP - framesize$

$M[FP + 4] := result$
GOTO $M[FP]$

Tail-Call Example

`return(g(x,y));` via stack-based *caller saves*.

$M[FP+4] := \mathtt{x}$
$M[FP+4*n] := \mathtt{y}$
$M[FP] := \mathit{returnaddress}$
`GOTO g`
`LABEL` *returnaddress*
$\mathit{result} := M[FP+4]$ $\longleftarrow$ Going off against each other (copy propagation)

$M[FP+4] := \mathit{result}$
`GOTO` $M[FP]$

Tail-Call Example

`return(g(x,y));` via stack-based *caller saves*.

$M[FP+4] := \mathtt{x}$
$M[FP+4*n] := \mathtt{y}$
$M[FP] := returnaddress$ $\longleftarrow$ Eliminated when we recycle the return address
GOTO g
LABEL *returnaddress*

GOTO $M[FP]$

Tail-Call Example

`return(g(x,y));` via stack-based *caller saves*.

$M[FP+4] := \mathtt{x}$
$M[FP+4*n] := \mathtt{y}$

GOTO g

GOTO $M[FP]$ $\longleftarrow$ Dead code

Tail-Call Example

`return(g(x,y));` via stack-based *caller saves*.

$M[FP + 4] := \texttt{x}$
$M[FP + 4 * n] := \texttt{y}$

`GOTO g`

1. Data-Flow Analysis
   - Common-Subexpression Elimination
   - Jump-to-Jump Elimination
   - Index-Checking Elimination
2. Loop Optimizations
   - Hoisting Loop-Invariant Computation
   - Prefetching
3. Function Calls
   - Inlining
   - Tail-Call Optimization
4. Specialization

Specialisering

Ideea: Specialize version of functions for constant-value parameters.
Example:

```
double power(double x, int n)
{
  double p=1.0;
  while (n>0)
    if (n%2 == 0) {
      x = x*x;
      n = n/2;
    } else {
      p = p*x;
      n = n-1;
    }
  return(p);
}
```

Specialized version of power for parameter `n=5`.

Specialisering

Ideea: Specialize version of functions for constant-value parameters.
Example:

```
double power(double x, int n)
{
  double p=1.0;
  while (n>0)
    if (n%2 == 0) {
      x = x*x;
      n = n/2;
    } else {
      p = p*x;
      n = n-1;
    }
  return(p);
}
```

```
double power5(double x)
{
  double p=1.0;
```

Specialized version of power for parameter `n=5`.

Specialisering

Ideea: Specialize version of functions for constant-value parameters.
Example:

```
double power(double x, int n)
{
  double p=1.0;
  while (n>0)
    if (n%2 == 0) {
      x = x*x;
      n = n/2;
    } else {
      p = p*x;
      n = n-1;
    }
  return(p);
}
```

```
double power5(double x)
{
  double p=1.0;
      p = p*x;
```

Specialized version of power for parameter `n=5`.

Specialisering

Ideea: Specialize version of functions for constant-value parameters.
Example:

```
double power(double x, int n)
{
  double p=1.0;
  while (n>0)
    if (n%2 == 0) {
      x = x*x;
      n = n/2;
    } else {
      p = p*x;
      n = n-1;
    }
  return(p);
}
```

```
double power5(double x)
{
  double p=1.0;
      p = p*x;
      x = x*x;
```

Specialized version of power for parameter n=5.

Specialisering

Ideea: Specialize version of functions for constant-value parameters.
Example:

```
double power(double x, int n)
{
  double p=1.0;
  while (n>0)
    if (n%2 == 0) {
      x = x*x;
      n = n/2;
    } else {
      p = p*x;
      n = n-1;
    }
  return(p);
}
```

```
double power5(double x)
{
  double p=1.0;
      p = p*x;
      x = x*x;
      x = x*x;
```

Specialized version of power for parameter n=5.

Specialisering

Ideea: Specialize version of functions for constant-value parameters.
Example:

```
double power(double x, int n)
{
  double p=1.0;
  while (n>0)
    if (n%2 == 0) {
      x = x*x;
      n = n/2;
    } else {
      p = p*x;
      n = n-1;
    }
  return(p);
}
```

```
double power5(double x)
{
  double p=1.0;
      p = p*x;
      x = x*x;
      x = x*x;
      p = p*x;
```

Specialized version of power for parameter `n=5`.

Specialisering

Ideea: Specialize version of functions for constant-value parameters.
Example:

```
double power(double x, int n)
{
  double p=1.0;
  while (n>0)
    if (n%2 == 0) {
      x = x*x;
      n = n/2;
    } else {
      p = p*x;
      n = n-1;
    }
  return(p);
}
```

```
double power5(double x)
{
  double p=1.0;
      p = p*x;
      x = x*x;
      x = x*x;
      p = p*x;
  return(p);
}
```

Specialized version of power for parameter `n=5`.

Specialisering

Ideea: Specialize version of functions for constant-value parameters.
Example:

```
double power(double x, int n)
{
  double p=1.0;
  while (n>0)
    if (n%2 == 0) {
      x = x*x;
      n = n/2;
    } else {
      p = p*x;
      n = n-1;
    }
  return(p);
}
```

```
double power5(double x)
{
  double p=1.0;
      p = p*x;
      x = x*x;
      x = x*x;
      p = p*x;
  return(p);
}
```

Specialized version of power for parameter `n=5`.
Specialization is the implementation method for C++ templates.