Programming Languages and Compiler Design

Programming Language Semantics
Compiler Design Techniques

Yassine Lakhnech & Laurent Mounier

{lakhnech,mounier}@imag.fr
http://www-verimag.imag.fr/~lakhnech
http://www-verimag.imag.fr/~mounier.

Master of Sciences in Informatics at Grenoble (MoSIG)
Grenoble Universités
(Université Joseph Fourier, Grenoble INP)
Code Optimization
Objective (of this chapter)

- give some indications on general optimization techniques:
  - data-flow analysis
  - register allocation
  - software pipelining
  - etc.
- describe the main data structures used:
  - control flow graph
  - intermediate code (e.g., 3-address code)
  - Static Single Assignment form (SSA)
  - etc.
- see some concrete examples

But not a complete panorama of the whole optimization process
(e.g.: a real compiler, for a modern processor)
**Objective of the optimization phase**

Improve the *efficiency* of the target code, while preserving the source semantics.

*efficiency* $\rightarrow$ several (antagonist) criteria

- execution time
- size
- memory used
- energy consumption
- etc.

$\Rightarrow$ no optimal solution, no general algorithm

$\Rightarrow$ a bunch of optimization techniques:

- inter-dependant each others
- sometimes heuristic based
Two kinds of optimizations

Independant from the target machine
“source level” or “assembly level” pgm transformations:
• dead code elimination
• constant propagation, constant folding
• code motion
• common subexpressions elimination
• etc.

Dependant from the target machine
optimize the use of the hardware resources:
• machine instruction
• memory hierarchy (registers, cache, pipeline, etc.)
• etc.
Overview

1. Introduction
2. Some optimizations independant from the target machine
3. Some optimizations dependant from the target machine
Some optimizations independant from the target machine
Main principle

Input: initial intermediate code
Output: optimized intermediate code

Several steps:
1. generation of a control flow graph (CFG)
2. analysis of the CFG
3. transformation of the CFG
4. generation of the output code
Intraprocedural 3-address code (TAC)

“high-level” assembly code:

- binary logic and arithmetic operators
- use of temporary memory location $ti$
- assignments to variables, temporary locations
- a label is assigned to each instruction
- conditional jumps $goto$

Examples:

- $l$: $x := y \text{ op } x$
- $l$: $x := \text{ op } y$
- $l$: $x := y$
- $l$: $goto \ l'$
- $l$: if $x \text{ oprel } y$ $goto \ l'$
**Basic block (BB)**

A maximal instruction sequence $S = i_1 \cdots i_n$ such that:

- $S$ execution is never “broken” by a jump
  $\Rightarrow$ no `goto` instruction in $i_1 \cdots i_{n-1}$
- $S$ execution cannot start somewhere in the middle
  $\Rightarrow$ no `label` in $i_2 \cdots i_n$

$\Rightarrow$ execution of a basic block is **atomic**

Partition of a 3-address code BBs:

1. computation of Basic Block heads:
   1st inst., inst. target of a jump, inst. following a jump
2. computation of Basic Block tails:
   last inst, inst. before a Basic Block head

$\Rightarrow$ a **single traversal** of the TAC
Control Flow Graph (CFG)

A representation of how the execution may progress inside the TAC

→ a graph \((V, E)\) such that:

\[
V = \{ B_i \mid B_i \text{ is a basic block} \}
\]

\[
E = \{ (B_i, B_j) \mid \text{“last inst. of } B_i \text{ is a jump to 1st inst of } B_j \text{”} \ \vee \\
\text{“1st inst of } B_j \text{ follows last inst of } B_i \text{ in the TAC”} \}
\]
Example

Give the Basic Blocks and CFG associated to the following TAC sequence:

0. \( x := 1 \)
1. \( y := 2 \)
2. \( \text{if } c \text{ goto 6} \)
3. \( x := x+1 \)
4. \( z := 4 \)
5. \( \text{goto 8} \)
6. \( z := 5 \)
7. \( \text{if } d \text{ goto 0} \)
8. \( z := z+2 \)
9. \( r := 1 \)
10. \( y := y-1 \)
Optimizations performed on the CFG

Two levels:

Local optimizations:

• computed inside each BB
• BBs are transformed independent each others

Global optimizations:

• computed on the CFG
• transformation of the CFG:
  • code motion between BBs
  • transformation of BBs
  • modification of the CFG edges
Local optimizations

• algebraic simplification, strength reduction
  → replace costly computations by less expensive ones

• copy propagation
  → suppress useless variables
  (i.e., equal to another one, or equal to a constant)

• constant folding
  → perform operations between constants

• common subexpressions
  → suppress duplicate computations
  (already computed before)

• dead code elimination → suppress useless instructions
  (which do not influence pgm execution)
Example of local optimizations

Initial code:

\[
\begin{align*}
a & := x \ast\ast 2 \\
b & := 3 \\
c & := x \\
d & := c \ast c \\
e & := b \ast 2 \\
f & := a + d \\
g & := e \ast f 
\end{align*}
\]
Example of local optimizations

Algebraic simplification:

\[
\begin{align*}
a &= x \times 2 & a &= x \times x \\
b &= 3 & b &= 3 \\
c &= x & c &= x \\
d &= c \times c & d &= c \times c \\
e &= b \times 2 & e &= b \ll 1 \\
f &= a + d & f &= a + d \\
g &= e \times f & g &= e \times f
\end{align*}
\]
Example of local optimizations

Copies propagation:

\[
\begin{align*}
a & := x \times x \\
b & := 3 \\
c & := x \\
d & := c \times c \\
e & := b \ll 1 \\
f & := a + d \\
g & := e \times f
\end{align*}
\]
Example of local optimizations

Constant folding:

\[
\begin{align*}
a & := x \ast x \\
b & := 3 \\
c & := x \\
d & := x \ast x \\
e & := 3 \ll 1 \\
f & := a + d \\
g & := e \ast f
\end{align*}
\]
Example of local optimizations

Elimination of common subexpressions:

```
a := x * x  a := x * x
b := 3      b := 3
```
```c
:= x      c := x
d := x * x  d := a
e := 6      e := 6
f := a + d  f := a + d
g := e * f  g := e * f
```
Example of local optimizations

Copies propagation:

\[
\begin{align*}
  a & := x \times x & a & := x \times x \\
  b & := 3 & b & := 3 \\
  c & := x & c & := x \\
  d & := a & d & := a \\
  e & := 6 & e & := 6 \\
  f & := a + d & f & := a + a \\
  g & := e \times f & g & := 6 \times f 
\end{align*}
\]
Example of local optimizations

Dead code elimination (+ strength reduction):

\[
a := x \times x \quad a := x \times x \quad a := x \times x
\]
\[
b := 3
\]
\[
c := x
\]
\[
d := a
\]
\[
e := 6
\]
\[
f := a + a \quad f := a + a \quad f := a \ll 1
\]
\[
g := 6 \times f \quad g := 6 \times f \quad g := 6 \times f
\]
Local optimization: a more concrete example

Initial source program: addition of matrices

```
for (i=0 ; i < 10 ; i++)
    for (j=0 ; j < 10 ; j++)
```

Basic blocks:

- **B1**: i := 0
- **B2**: if i > 10 goto B7
- **B3**: j := 0
- **B4**: if j > 10 goto B6
- **B5**
- **B6**: i := i + 1
  
  goto B2
- **B7**: end
Control Flow Graph

B1

B2

B6

B3

B4

B5

B7
Initial Block B5

B5:

\[ t1 := 4 \times i \]
\[ t2 := 40 \times j \]
\[ t3 := t1 + t2 \]
\[ t4 := A[t3] \]
\[ t5 := 4 \times i \]
\[ t6 := 40 \times j \]
\[ t7 := t5 + t6 \]

\[ t8 := B[t7] \]
\[ t9 := t4 + t8 \]
\[ t10 := 4 \times i \]
\[ t11 := 40 \times j \]
\[ t12 := t10 + t11 \]

\[ S[t12] := t9 \]
\[ j := j + 1 \]

goto B4
Optimization of B5 (1/4)

B5:  
\[
\begin{align*}
  t1 & := 4 \times i \\
  t2 & := 40 \times j \\
  t3 & := t1 + t2 \\
  t4 & := A[t3] \\
  t5 & := 4 \times i \\
  t6 & := 40 \times j \\
  t7 & := t5 + t6
\end{align*}
\]

\[
\begin{align*}
  t8 & := B[t7] \\
  t9 & := t4 + t8 \\
  t10 & := 4 \times i \\
  t11 & := 40 \times j \\
  t12 & := t10 + t11 \\
  S[t12] & := t9 \\
  j & := j + 1 \\
  \text{goto B4}
\end{align*}
\]

A same value is assigned to temporary locations t1, t5, t10
### Optimization of B5 (2/4)

<table>
<thead>
<tr>
<th>B5</th>
<th>( t_1 := 4 \times i )</th>
<th>( t_2 := 40 \times j )</th>
<th>( t_3 := t_1 + t_2 )</th>
<th>( t_4 := A[t_3] )</th>
<th>( t_6 := 40 \times j )</th>
<th>( t_7 := t_1 + t_6 )</th>
<th>( t_8 := B[t_7] )</th>
<th>( t_9 := t_4 + t_8 )</th>
<th>( t_{11} := 40 \times j )</th>
<th>( t_{12} := t_1 + t_{11} )</th>
<th>( S[t_{12}] := t_9 )</th>
<th>( j := j + 1 )</th>
<th>goto B4</th>
</tr>
</thead>
</table>

A same value is assigned to temporary locations \( t_2, t_6, t_{11} \)
Optimization of B5 (3/4)

B5:
\[
\begin{align*}
  t1 & := 4 \times i \\
  t2 & := 40 \times j \\
  t3 & := t1 + t2 \\
  t4 & := A[t3] \\
  t7 & := t1 + t2
\end{align*}
\]

\[
\begin{align*}
  t8 & := B[t7] \\
  t9 & := t4 + t8 \\
  t12 & := t1 + t2 \\
  S[t12] & := t9 \\
  j & := j + 1 \\
  \text{goto B4}
\end{align*}
\]

A same value is assigned to temporary locations t3, t7, t12
Optimization of B5 (4/4): the final code obtained

B5:  
\[ t_1 := 4 \times i \]  
\[ t_2 := 40 \times j \]  
\[ t_3 := t_1 + t_2 \]  
\[ t_4 := A[t_3] \]  
\[ t_8 := B[t_3] \]  
\[ t_9 := t_4 + t_8 \]  
\[ S[t_3] := t_9 \]  
\[ j := j + 1 \]  
goto B4
Global optimizations
Global optimization: the principle

Typical examples of global optimizations:
- constant propagation through several basic blocks
- elimination of global redundancies
- code motion: move invariant computations outside loops
- dead code elimination

How to “extrapolate” local optimizations to the whole CFG?

1. associate (local) properties to entry/exit points of BBs
   (set of active variables, set of available expressions, etc.)

2. propagate them along CFG paths
   → enforce consistency w.r.t. the CFG structure

3. update each BB (and CFG edges) according to these global properties

⇒ a possible technique: data-flow analysis
Data-flow analysis

Static computation of data related properties of programs

• (local) properties \( \varphi_i \) associated to some pgm locations \( i \)

• set of data-flow equations:
  \( \rightarrow \) how \( \varphi_i \) are transformed along pgm execution

Rks:
• forward vs backward propagation (depending on \( \varphi_i \))
• cycles inside the control flow \( \Rightarrow \) fix-point equations!

• a solution of this equation system:
  \( \rightarrow \) assigns “globaly consistent” values to each \( \varphi_i \)

Rk: such a solution may not exist . . .

• decidability may require abstractions and/or approximations
Example: elimination of redundant computations

An expression $e$ is redundant at location $i$ iff

• it is computed at location $i$
• this expression is computed on every path going from the initial location to location $i$
  \[ \text{Rk: we consider here syntactic equality} \]
• on each of these paths: operands of $e$ are not modified between the last computation of $e$ and location $i$

Optimization is performed as follows:

1. computation of available expressions (data-flow analysis)
2. $x := e$ is redundant at loc $i$ if $e$ is available at $i$
3. $x := e$ is replaced by $x := t$
   (where $t$ is a temp. memory containing the value of $e$)
Elimination of redundant computation: an example
Data-flow equations for available expressions (1/2)

For a basic block $b$, we note:

- $In(b)$ : available expressions when entering $b$
- $Kill(b)$: expressions made non available by $b$ (because an operand of $e$ is modified by $b$)
- $Gen(b)$: expressions made available by block $b$ (computed in $b$, operands not modified afterwards)
- $Out(b)$ : available expressions when exiting $b$

\[
Out(b) = (In(b) \setminus Kill(b)) \cup Gen(b) = F_b(In(b))
\]

$F_b$ = transfer function of block $b$
Data-flow equations for available expressions (2/2)

How to compute $In(b)$?

- if $b$ is the initial block:
  \[
  In(b) = \emptyset
  \]

- if $b$ is not the initial block:
  An expression $e$ is available at its entry point iff it is available at the exit point of each predecessor of $b$ in the CFG
  \[
  In(b) = \bigcap_{b' \in Pre(b)} Out(b')
  \]

⇒ forward data-flow analysis along the CFG paths

Q: cycles inside the CFG ⇒ fix-points computations greatest \textit{vd} least solutions?
Solving the data-flow equations (1/2)

Let \((E, \leq)\) a partial order.

- For \(X \subseteq E, a \in E\):
  - \(a\) is an upper bound of \(X\) if \(\forall x \in X. x \leq a\)
  - \(a\) is a lower bound of \(X\) if \(\forall x \in X. a \leq x\)
- The least upper bound \((\text{lub}, \sqcup)\) is the smallest upper bound
- The great lower bound \((\text{glb}, \sqcap)\) is the largest lower bound
- \((E, \leq)\) is a lattice if every subset of \(E\) admits a lub and a glb.
- A function \(f : 2^E \rightarrow 2^E\) is monotonic if:
  \[
  \forall X, Y \subseteq E \quad X \leq Y \Rightarrow f(X) \leq f(Y)
  \]
- \(X = \{x_0, x_1, \ldots x_n, \ldots\} \subseteq E\) is an (increasing) chain if \(x_0 \leq x_1 \leq \ldots x_n \leq \ldots\)
- A function \(f : 2^E \rightarrow 2^E\) is (\(\sqcup\)-)continuous if \(\forall\) increasing chain \(X, f(\sqcup X) = \sqcup f(X)\)
Solving the data-flow equations (2/2)

Fix-point equation: solution?

- properties are finite sets of expressions \( \mathcal{E} \)
- \((\mathcal{2}^\mathcal{E}, \subseteq)\) is a complete lattice
  - \(\bot\): least element, \(\top\): greatest element
  - \(\sqcap\): greatest lower bound, \(\sqcup\): least upper bound
- data-flow equations are defined on monotonic and continuous operators \((\cup, \cap)\) on \((\mathcal{2}^\mathcal{E}, \subseteq)\)
- Kleene and Tarski theorems:
  - the set of solution is a complete lattice
  - the greatest (resp. least) solution can be obtained by successive iterations w.r.t. the greatest (resp. least) element of \(\mathcal{2}^\mathcal{E}\)

\[
\text{Ifp}(f) = \cup\{f^i(\bot) | i \in \mathbb{N}\} \quad \text{gfp}(f) = \cap\{f^i(\top) | i \in \mathbb{N}\}
\]
Back to the example

```
x := ...
a := ...
b := ...
y := y + 1
x := a + b
y := c
z := x + 1
v := a + b
r := a + b
end
```

In = \{a + b\}
Out = \{a + b\}

In = 0
Out = 0
**Generalization**

- Data-flow properties are expressed as finite sets associated to entry/exit points of basic blocks: \( \text{In}(b) \), \( \text{Out}(b) \)

- For a **forward** analysis:
  - property is “false” (\( \perp \)) at entry of initial block
  - \( \text{Out}(b) = F_b(\text{In}(b)) \)
  - \( \text{In}(b) \) depends on \( \text{Out}(b') \), where \( b' \in \text{Pred}(b) \)
    (\( \sqcap \) for “\( \forall \) paths”, \( \sqcup \) for “\( \exists \) path”)

- For a **backward** analysis:
  - property is “false” (\( \perp \)) at exit of final block
  - \( \text{In}(b) = F_b(\text{Out}(b)) \)
  - \( \text{Out}(b) \) depends on \( \text{In}(b') \), where \( b' \in \text{Succ}(b) \)
**Data-flow equations: forward analysis**

| Forward analysis, least fix-point | $\text{In}(b) = \begin{cases} \bot & \text{if } b \text{ is initial} \\ \bigsqcup_{b' \in \text{Pre}(b)} \text{Out}(b') & \text{otherwise.} \end{cases}$  \\
<table>
<thead>
<tr>
<th></th>
<th>$\text{Out}(b) = F_b(\text{In}(b))$</th>
</tr>
</thead>
</table>
| Forward analysis, greatest fix-point | $\text{In}(b) = \begin{cases} \bot & \text{if } b \text{ is initial} \\ \bigsqcap_{b' \in \text{Pre}(b)} \text{Out}(b') & \text{otherwise.} \end{cases}$  \\
| | $\text{Out}(b) = F_b(\text{In}(b))$ |
### Data-flow equations: backward analysis

<table>
<thead>
<tr>
<th>Backward analysis, least fix-point</th>
<th>$\text{Out}(b) = \begin{cases} \bot &amp; \text{if } b \text{ is final} \ \bigsqcup_{b' \in \text{Succ}(b')} \text{In}(b') &amp; \text{otherwise.} \end{cases}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\text{In}(b) = F_b(\text{Out}(b))$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Backward analysis, greatest fix-point</th>
<th>$\text{Out}(b) = \begin{cases} \bot &amp; \text{if } b \text{ is final} \ \bigsqcup_{b' \in \text{Succ}(b')} \text{In}(b') &amp; \text{otherwise.} \end{cases}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\text{In}(b) = F_b(\text{Out}(b))$</td>
</tr>
</tbody>
</table>
Active Variable

- A variable $x$ is inactive at location $i$ if it is not used in every CFG-path going from $i$ to $j$, where $j$ is:
  - either a final instruction
  - or an assignment to $x$.

- An instruction $x := e$ at location $i$ is useless if $x$ is inactive at location $i$.

$\Rightarrow$ useless instructions can be removed . . .

Rk: used means

“in a right-hand side assignment or in a branch condition”.
Data-flow analysis for inactive variables

We compute the set of **active** variables . . .

**Local analysis**

\( \text{Gen}(b) \) is the set of variables \( x \) s.t. \( x \) is used in block \( b \), and, in this block, any assignment to \( x \) happens after the (first) use of \( x \).

\( \text{Kill}(i) \) is the set of variables \( x \) assigned in block \( b \).

**Global analysis** : backward analysis, \( \exists \) a CFG-path (least solution)

\[
\text{Out}(b) = \bigcup_{b' \in \text{Succ}(b)} \text{In}(b')
\]

\[
\text{In}(b) = (\text{Out}(b) \setminus \text{Kill}(b)) \cup \text{Gen}(b)
\]

\( \text{Out}(b) = \emptyset \) if \( b \) is final.
Computation of functions $Gen$ and $Kill$

Recursively defined on the syntax of a basic bloc $B$:

$$B ::= \varepsilon \mid B ; x := a \mid B ; \text{if } b \text{ goto } l \mid B ; \text{goto } l$$

<table>
<thead>
<tr>
<th>Expression</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Gen(B)$</td>
<td>$Gen_l(B, \emptyset)$</td>
</tr>
<tr>
<td>$Kill(B)$</td>
<td>$Kill_l(B, \emptyset)$</td>
</tr>
<tr>
<td>$Gen_l(B ; x := a, X)$</td>
<td>$Gen_l(B, X \setminus {x} \cup \text{Used}(a))$</td>
</tr>
<tr>
<td>$Gen_l(B ; \text{if } b \text{ goto } l, X)$</td>
<td>$Gen_l(B, X \cup \text{Used}(b))$</td>
</tr>
<tr>
<td>$Gen_l(B ; \text{goto } l, X)$</td>
<td>$Gen_l(B, X)$</td>
</tr>
<tr>
<td>$Gen_l(\varepsilon, X)$</td>
<td>$X$</td>
</tr>
<tr>
<td>$Kill_l(B ; x := a, X)$</td>
<td>$Kill_l(B, X \cup {x})$</td>
</tr>
<tr>
<td>$Kill_l(B ; \text{if } b \text{ goto } l, X)$</td>
<td>$Kill_l(B, X)$</td>
</tr>
<tr>
<td>$Kill_l(B ; \text{goto } l, X)$</td>
<td>$Kill_l(B, X)$</td>
</tr>
<tr>
<td>$Kill_l(\varepsilon, X)$</td>
<td>$X$</td>
</tr>
</tbody>
</table>

$\text{Used}(\varepsilon)$: set of variables appearing in expression $\varepsilon$
Removal of useless instructions

1. Compute the sets \( In(B) \) and \( Out(B) \) of active variables at entry and exit points of each blocks.

2. Let \( F : Code \times 2^{Var} \rightarrow Code \)
   \( F(b, X) \) is the code obtained when removing useless assignments inside \( b \), assuming that variables of \( X \) are active at the end of \( b \) execution.

   \[
   F(B; x := a, X) = \begin{cases} 
   F(B, X) & \text{if } x \not\in X \\
   F(B, (X \setminus \{x\}) \cup \text{Used}(a)); x := a & \text{if } x \in X
   \end{cases}
   \]

   \[
   F(B; \text{if } b \text{ goto } l, X) = F(B, X \cup \text{Used}(b)); \text{if } b \text{ goto } l
   \]

   \[
   F(B; \text{goto } l, X) = F(B, X); \text{goto } l
   \]

   \[
   F(\epsilon, X) = \epsilon
   \]

3. Replace each block \( B \) by \( F(B, Out(B)) \).

Rk: this transformation may produce new inactive variables . . .
**Constant propagation**

Example:

- A variable is **constant** at location $l$ if its value at this location can be computed at compilation time.
- At exit point of B1 and B2, $i$ and $j$ are constants
- At entry point of B3, $i$ is not constant, $j$ is constant.
Constant propagation: the lattice

• Each variable takes its value in $D = \mathbb{N} \cup \{\top, \bot\}$, where:
  • $\top$ means “non constant value”
  • $\bot$ means “no information”

• Partial order relation $\leq$:
  if $v \in D$ then $\bot \leq v$ and $v \leq \top$.

• The least upper bound $\sqcup$:
  for $x \in D$ and $v_1, v_2 \in \mathbb{N}$
  \[
  \begin{align*}
  x \sqcup \top &= \top \\
  x \sqcup \bot &= x \\
  v_1 \sqcup v_2 &= \top \text{ if } v_1 \neq v_2 \\
  v_1 \sqcup v_1 &= v_1
  \end{align*}
  \]

Rk: relations $\leq$ is extended to functions $Var \rightarrow D$

$$f_1 \leq f_2 \text{ iff } \forall x. f_1(x) \leq f_2(x)$$
Constant propagation: data-flow equations

• property at location 1 is a function $Var \rightarrow D$.
• Forward analysis:

$$\begin{align*}
In(b) &= \begin{cases}
\lambda x. \bot & \text{if } b \text{ is initial}, \\
\bigsqcup \limits_{b' \in \text{Pred}(b)} Out(b') & \text{otherwise}
\end{cases} \\
Out(b) &= F_b(In(b))
\end{align*}$$

Transfer function $F_b$?
a basic block $= \text{sequence of assignements}$

$$b ::= \epsilon | x := e ; b$$

$F_b$ defined by syntactic induction:

$$F_{x := e} ; b(f) = F_b(f[x \mapsto f(e)]) \quad \text{(assuming variable initialization)}$$

$F_\epsilon(f) = f$

Pgm transformation:

$$\forall \text{ block } b, f \in In(b), f(e) = v \Rightarrow x := e \text{ replaced by } x := v$$
Exercise

Constant propagation can be viewed as abstraction of the standard semantics where expressions values are interpreted in another domain \( D \).

1. Write this abstract semantics for the while language in an operational style (relation \( \rightarrow \# \)).

2. Define a program transformation which removes useless computations (i.e., computations between constant operands).

3. Give the equations which express the correctness of this transformation.
Another example of data-flow analysis

A computation of an expression $e$ can be anticipated at loc. $p$ iff:

- all paths from $p$ contains a location $p_i$ s.t. $e$ is computed at $p_i$
- $e$ operands are not modified between $p$ and $p_i$

Example:

```
if (x>0)
    x = i + j;
else
    repeat y = (i + j) * 2; x := x+1 ; until x>10
```

can be changed to

```
tmp = i + j;
if (x>0)
    x = tmp;
else
    repeat y = tmp * 2; x := x+ 1 ; until x>10
```
Interprocedural analysis

```c
main()
{
    int i, j;
    void f()
    {
        int x, y;
        y = i+j; x = y;
    }
    i = 0;
    f();
    j = 1;
}
```

- a dedicated basic block $B_{call}$ for the call instruction
- $In(B_{call}) = In(B_{fin})$, $Out(B_{call}) = Out(B_{fout})$

Rks:

- static binding is be assumed
- parameters?

Exercice: Computation of active variables
Control-flow analysis

→ retrieve program control structures from the CFG?

Application: loop identification

⇒ use of graph-theoretic notions:
  • dominator, dominance relation
  • strongly connected components

Rk1: most loops are easier to identify at syntactic level, but:
  • use of goto instruction still allowed in high-level languages
  • optimization performed on intermediate representations (e.g., CFG)

Rk2: other approaches can be used to identify loops . . .
Loop identification

Node $B_1$ is a **dominator** of $B_2$ ($B_2 \leq B_1$) iff every path from the entry block to $B_2$ goes through $B_1$. $Dom(B) = \{B_i | B_i \leq B\}$.

An edge $(B_1, B_2)$ is a loop **back edge** iff $B_2 \leq B_1$

To find “natural loops”:

1. find a **back edge** $(B_1, B_2)$
2. find $Dom(B_2)$
3. find blocks $B_i \in Dom(B_2)$ s.t. there is a path from $B_i$ to $B_2$ not containing $B_1$. 
Some machine level optimization techniques
Register Allocation

Pb:

- expression operands are much efficiently accessed when lying in registers (instead of RAM)
- the “real” number of registers is finite (and usually small)

⇒ register allocation techniques:

- assigns a register to each operand (variable, temporary location)
- performs the memory exchange (LD, ST) when necessary
- optimality?

Several existing techniques:

- optimal code generation for arithmetic expressions
- graph-coloring techniques (more general case)
- etc.
code generation for $(a+b) - (c - (d+e))$
with 2 registers, and instruction format = $\text{OP } Ri, Ri, X$ (where $X=Ri$ or $X=M[x]$)

Solution 1: one register needs to be saved

LD R0, M[a]
ADD R0, R0, M[b]
LD R1, M[d]
ADD R1, R1, M[e]
ST R1, M[t1] ! register R1 needs to be saved ...
LD R1, M[c]
SUB R1, R1, M[t1]
SUB R0, R0, R1

Solution 2: no register to save

LD R0, M[c]
LD R1, M[d]
ADD R1, R1, M[e]
SUB R0, R0, R1
LD R1, M[a]
ADD, R1, R1, M[b]
SUB, R1, R1, R0
Code generation for arithmetic expressions: principle

Evaluation of $e_1 \text{ op } e_2$, assuming:

- $r$ registers are available, evaluation of $e_i$ requires $r_i$ registers
- Instruction format is “op reg, reg, ad” where “ad” is a register or a memory location

Several cases:

- $r_1 > r_2$:
  - After evaluation of $e_1$, $r_1 - 1$ registers available
  - $r_1 - 1 \geq r_2 \Rightarrow r_1 - 1$ registers are enough for $e_2$
  - $\Rightarrow r_1 - r$ register allocations are required

- $r_1 = r_2$:
  - After evaluation of $e_1$, $r_1 - 1$ registers available
  - $r_1 - 1 < r_2 \Rightarrow r_2 (= r_1)$ registers required for $e_2$
  - $\Rightarrow r_1 + 1 - r$ register allocations are required

- $r_1 < r_2$:
  - After evaluation of $e_1$, $r_1 - 1$ registers available
  - $r_1 - 1 < r_2 \Rightarrow r_2 (> r_1)$ registers required for $e_2$
  - $\Rightarrow r_2 + 1 - r$ register allocations are required
  - $r_2 - r$ allocations are enough if $e_2$ is evaluated first!
A two-phase algorithm

Step 1: each AST node is labeled with the number of registers required for its evaluation

\[ rNb : Aexp \rightarrow \mathbb{N} \] (\( rNb(e) \) is the number of registers required to evaluate \( e \))

\[
\begin{align*}
\text{rNb}(e) & = \begin{cases} 
1 & \text{if } e \text{ is a left leaf} \\
0 & \text{if } e \text{ is a right leaf}
\end{cases} \\
\text{rNb}(e_1 \text{ op } e_2) & = \begin{cases} 
\max(\text{rNb}(e_1), \text{rNb}(e_2)) & \text{if } \text{rNb}(e_1) \neq \text{rNb}(e_2) \\
\text{rNb}(e_1) + 1 & \text{if } \text{rNb}(e_1) = \text{rNb}(e_2)
\end{cases}
\end{align*}
\]

Step 2: “optimal” code generation using these labels (exercice)

→ for a binary node \( e_1 \text{ op } e_2 \):

• evaluate the more register demanding sub-expression first
• write the result in a register \( R_i \) (save one if necessary)
• evaluate the other sub-expression, write the result in a register \( R_j \)
• generate \( \text{OP, } R_i, R_i, R_j \)
A more general technique

1. Intermediate code is generated assuming \( \infty \) numbers of “symbolic” registers \( S_i \)

2. Assign a real register \( R_i \) to each symbolic register s.t.
   - if \( R_i \) is assigned to \( S_i \), \( R_j \) is assigned to \( S_j \)
   - then \( \text{Lifetime}(S_i) \cap \text{lifetime}(S_j) \neq \emptyset \) \( \Rightarrow \) \( R_i \neq R_j \)

   where \( \text{Lifetime}(S_i) \): sequences of pgm location where \( S_i \) is active

How to ensure this condition?

Collision graph \( G_C \):

- Nodes denote lifetime symbolic registers: \( N_i = (S_i, \text{Lifetime}(S_i)) \)
- Edges are the set \( \{(S_1, L_1), (S_2, L_2) \mid L_1 \text{ and } L_2 \text{ overlap}\} \)

\( \Rightarrow \) register allocation with \( k \) real register = \( k \)-coloring problem of \( G_C \)

(i.e., assign a distinct colour to each pair of adjacent nodes)
Example 1

S1 := e1
S2 := e2
...
... S2 ...
S3 := S1+S2
...
S4 := S1*5
...
...
...
... S3 ...
... S3 ...

Collision Graph:

Can be colored with 2 colors \( \Rightarrow \) 2 real registers are enough.
**k-coloring in practice ? (1)**

When $k > 2$, this problem is NP-complete ...

An efficient heuristic:

Repeat:

if exists a node $N$ of $G_C$ such that degree$(N) < k$

($N$ can receive a distinct colour from all its neighbours)

remove $N$ (and corresponding edges) from $G_C$ and push it on a stack $S$

else ($G_C$ is assumed to be non $k$-colourable)

choose a node $N$ \(^1\)

remove $N$ from $G_C$ \(^2\)

until $G_C$ is empty

While $S$ is not empty

pop a node from $S$

add it to $G$, give it a colour not used by one of its neighbours

**Rk:** this algo may sometimes miss $k$-colorable graphs ...
$k$-coloring in practice? (2)

What happens when there is no node of degree $< k$?

1. choose a node $N$ to remove:
   → high degree in $G_C$, not corresponding to an inner loop, etc.

2. remove node $N$:
   → save a register into memory before (register spilling)

Several attempts to improve this algorithm:

node coalescing:

\[ S_1 := S_2, \text{Lifetime}(S_1) \cap \text{Lifetime}(S_2) = \emptyset \]

⇒ nodes associated to $S_1$ and $S_2$ could be merged

pb: it increases the graph degree . . .

lifetime splitting:

long lifetime increases the graph degree

⇒ split it into several parts . . .

pb: where to split?
**Instruction scheduling**

Motivation: exploit the instruction parallelism provided in many target architectures (e.g., VLIW processors, instruction pipeline, etc.)

**Pbs:**

- possible *data dependancies* between consecutive instructions (e.g., \( x := 3 \); \( y := x+1 \))
- possible *resource conflicts* between consecutive instructions (ALU, co-processors, bus, etc.)
- consecutive instructions may require various *execution cycles*
- etc.

⇒ **Main technique:** change the initial instruction sequence (*instruction scheduling*)

- preserve the initial *pgm semantics*
- better exploit the hardware resources

**Rks:** “loop unrolling” and “expression tree reduction” may help . . .
Data dependencies:
→ execution order of 2 instructions should be preserved in the following situation:

**Read After Write (RAW)**: inst. 2 read a data written by inst. 1

**Write After Read (WAR)**: inst. 2 write a data read by inst. 1

**Write After Write (WAW)**: inst. 2 write a data written by inst. 1

Dependency graph $G_D$

- nodes = \{ instructions \}
- edges = \{(i_1, d, i_2) \mid there is a dependency d from i_1 to i_2\}

Rk: if we consider a basic block, $G_D$ is a directed acyclic graph.

Any topological sort of $G_D$ leads to a valid result (w.r.t. pgm semantics).
This sort can be influenced by several factors:

- the resources used by the instruction (∃ a static reservation table)
- the number of cycles it requires (latency)
- etc.
Example

1. Draw the dependency graph $G_D$ associated to the following program
2. Give a topological sort of $G_D$
3. Rewrite this program with a “maximal” parallelism

1. $a := x+1$
2. $x := 2+y$
3. $y := z+1$
4. $t := a*b$
5. $v := a*c$
6. $v := 3+t$
Software pipelining (overview . . .)

Idea: exploit the parallelism between instructions of distinct loop iterations

for k in 1 .. N loop
    r := T[k] ; - inst. A
    x := x + r ; - inst. B
    T[k] := x ; - inst. C
end loop

Assumptions: 3 cycles per instruction, 1 cycle delay when no dependencies

• Initial exec. sequence: A(1), B(1), C(1), A(2), B(2), C(2), . . . A(k), B(k), C(k)
  ⇒ 7 cycles / iteration

• “Pipelined exec. sequence”: A(1), A(2), A(3), B(1), B(2), B(3), C(1), C(2), C(3), . . .
  ⇒ 3 cycles / iteration !

(real life) pbs:

• N not always divisible by the number of instruction in the loop body
  for k in 1 to N-2 step 3 loop A(k) ; A(k+1) ; A(k+2) . . .

• high latency instruction in the loop body

• possible overhead when k is not “large enough”

• . . .