Theory ofCompilation
236360

ErezPetrunk

Lecture 10:
Code Generation & Optimizations
Compiler

Source text

Lexical Analysis

Syntax Analysis

Semantic Analysis

Inter. Rep. (IR)

Code Gen.

Executable code

characters

tokens

(Associate Syntax Tree)

AST

Annotated AST

You are here
Code Generation: Backend

• Generate code for a specific machine.
• Assume no errors at this point.
• Input: Intermediate language code + symbol table
• Output: target language code
target languages

IR + Symbol Table → Code Gen. → Absolute machine code

IR + Symbol Table → Code Gen. → Relative machine code

IR + Symbol Table → Code Gen. → Assembly
Assembly to Runnable

Assembly ➔ Assembler ➔ Object code (relocatable) ➔ Linker ➔ executable ➔ Loader ➔ Program in memory
From IR to ASM: Challenges

• **Mapping IR to ASM operations**
  – what instruction(s) should be used to implement an IR operation?
  – how do we translate code sequences

• **Call/return of routines**
  – managing activation records

• **Memory allocation**

• **Register allocation**

• **Optimizations**
Basic Blocks

• An important notion.
• Start by breaking the IR into basic blocks
• A basic block is a sequence of instructions with
  – A single entry (to first instruction), no jumps to the middle of the block
  – A single exit (last instruction)
  – Thus, code execute as a sequence from first instruction to last instruction without any jumps
• We put an edge from B1 to B2 when the last statement of B1 may jump to B2
Example

B1
\[ t_1 := 4 \times i \]
\[ t_2 := a[ t_1 ] \]
\[ \text{if } t_2 \leq 20 \text{ goto } B_3 \]

B2
\[ t_3 := 4 \times i \]
\[ t_4 := b[ t_3 ] \]
\[ \text{goto } B_4 \]

B3
\[ t_5 := t_2 \times t_4 \]
\[ t_6 := \text{prod} + t_5 \]
\[ \text{prod} := t_6 \]
\[ \text{goto } B_4 \]

B4
\[ t_7 := i + 1 \]
\[ i := t_2 \]
\[ \text{Goto } B_5 \]
A directed graph $G=(V,E)$
- nodes $V =$ basic blocks
- edges $E =$ control flow
  - $(B1,B2) \in E$ if control from $B1$ flows to $B2$

A loop is a strongly connected component of the graph that has a single entry point.
An inner loop is a loop that has no sub-loop.

```
prod := 0
i := 1

\begin{align*}
   t_1 &:= 4 * i \\
   t_2 &:= a [ t_1 ] \\
   t_3 &:= 4 * i \\
   t_4 &:= b [ t_3 ] \\
   t_5 &:= t_2 * t_4 \\
   t_6 &:= prod + t_5 \\
   prod &:= t_6 \\
   t_7 &:= i + 1 \\
   i &:= t_7 \\
   \text{if } i \leq 20 \text{ goto } B_2
\end{align*}
```
Determining Basic Blocks

• **Input:** A sequence of three-address statements
• **Output:** A list of basic blocks with each three-address statement in exactly one block
• **Method**
  – Determine the set of leaders (first statement of a block)
    • The first statement is a leader
    • Any statement that is the target of a conditional or unconditional jump is a leader
    • Any statement that immediately follows a goto or conditional jump statement is a leader
  – For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program
for i from 1 to 10 do
for j from 1 to 10 do
  a[i, j] = 0.0;
for i from 1 to 10 do
  a[i, i] = 1.0;
for i from 1 to 10 do
  for j from 1 to 10 do
    a[i, j] = 0.0;
  for i from 1 to 10 do
    a[i, i] = 1.0;
Optimizations

• Optimization employs “program understanding“ = program analysis.

• An example: to allocate registers efficiently we need to analyze liveness of variables.

• Analysis boundaries:
  – inside basic block boundaries.
  – all blocks in a method
  – all blocks in the program (whole program analysis)

Accuracy
Cost
Variable Liveness

- A statement $x = y + z$
  - defines $x$
  - uses $y$ and $z$
- A variable $x$ is live at a program point if its value is used at a later point without being defined beforehand
- If $x$ is defined in instruction $i$, used in instr. $j$, and there is a computation path from $i$ to $j$ that does not modify $x$, then instr. $j$ is using the value of $x$ that is defined in instr. $i$.

\[
\begin{align*}
y &= 42 \\
z &= 73 \\
x &= y + z \\
\text{print}(x); \\
\end{align*}
\]
\begin{itemize}
  \item x undef, y live, z undef
  \item x undef, y live, z live
  \item x is live, y dead, z dead
  \item x is dead, y dead, z dead
\end{itemize}

(showing state after the statement)
Computing Liveness Information

- Between basic blocks - dataflow analysis (next lecture)

- Within a single basic block?
  - scan basic block backwards
  - update next-use for each variable
Computing Liveness Information

• **INPUT:** A basic block B of three-address statements. Initially all non-temporary variables in B marked live on exit.

• **OUTPUT:** For each statement i: a = b + c in B, liveness and next-use information of a, b, and c at i.

• **Algorithm:**
  
  Start at the last statement in B and scan backwards
  
  – For each statement i: a = b + c in B:
    
    1. Set i’s output: the information currently found in the symbol table for next use and liveness of a, b, and c.
    
    2. In the symbol table, set a to "not live" and "no next use."
    
    3. In the symbol table, set b and c to "live" and the next uses of b and c to i
Computing Liveness Information

- Start at the last statement in B and scan backwards
  - For each statement \( i: a = b + c \) in B:
    1. Set \( i \)'s output: the information currently found in the symbol table for next use and liveness of \( a, b, \) and \( c \).
    2. In the symbol table, set \( a \) to "not live" and "no next use."
    3. In the symbol table, set \( b \) and \( c \) to "live" and the next uses of \( b \) and \( c \) to \( i \)

\[
\begin{align*}
  x & = y \times 1 \\
  y & = x + 3 \\
  z & = y \times 3 \\
  x & = x \times z
\end{align*}
\]

\( x \) dead; \( y \) live; \( z \) dead;
\( x \) live; \( y \) dead; \( z \) dead;
\( x \) live; \( y \) live; \( z \) dead;
\( x \) live; \( y \) dead; \( z \) live;
\( x \) dead; \( y \) dead; \( z \) dead;  \hspace{1cm} \text{(Given)}

Can we change the order between steps 2 and 3?
common-subexpression elimination

- common-subexpression elimination

  \[
  \begin{align*}
  a &= b + c \\
  b &= a - d \\
  c &= b + c \\
  d &= a - d
  \end{align*}
  \]

- Easily identified by DAG representation.
DAG Representation of Basic Blocks

\[
\begin{align*}
  a &= b + c \\
  b &= a - d \\
  c &= b + c \\
  d &= a - d \\
\end{align*}
\]
Use indices to distinguish variable assignments

\[
\begin{align*}
a &= b + c \\
b &= a - d \\
c &= b + c \\
d &= a - d
\end{align*}
\]

We will talk later on how to build the DAG.
DAG and Dead Code

\[
\begin{align*}
    a &= b + c \\
    b &= b - d \\
    c &= c + d \\
    e &= b + c
\end{align*}
\]

- Perform **dead code elimination**.

\[
\begin{array}{c}
+ \\
+ \\
- \\
+ \\
\end{array}
\begin{array}{c}
+ \\
+ \\
- \\
+ \\
\end{array}
\begin{array}{c}
b_0 \\
c_0 \\
d_0 \\
e \\
\end{array}
\]

- **a not used out of block**
- **e used out of block**
algebraic identities

\[
\begin{align*}
  a &= x^2 \\
  b &= x \times 2 \\
  e &= x \times 2^i \\
  c &= x \div 2^i \\
  d &= 1 \times x \\
  f &= x + 0
\end{align*}
\]

\[
\begin{align*}
  a &= x \times x \\
  b &= x + x \\
  e &= \text{shift}(x, i) \\
  c &= \text{shift}(x, -i) \\
  d &= x \\
  f &= x
\end{align*}
\]
Next

- Register allocation using full liveness information across basic blocks
  - We will see how to compute it next time
Simple code generation

• registers
  – used as operands of instructions
  – can be used to store temporary results
  – can (should) be used as loop indices due to frequent arithmetic operation
  – used to manage administrative info (e.g., runtime stack)

• Number of registers is limited
• Need to allocate them in a clever way
• Critical for program efficiency.
simple code generation

• Assume machine instructions of the form
  • LD reg, mem
  • ST mem, reg
  • OP reg,reg,reg

• Assume that we have all registers available for our use
  – Ignore registers allocated for stack management
  – Treat all registers as general-purpose
simple code generation

• Translate each 3AC instruction separately.

  Maintain:

• Register descriptor for a register \( R \): which variables are in \( R \).

• Address descriptor for a variable \( v \): where \( v \)'s value can be found.
simple code generation

• Register descriptor keeps track of the variable names whose current value is in that register.
  – Initially, all register descriptors are empty.
  – Each register will hold the value of zero or more names.

• An address descriptor for a program variable keeps locations where its current value can be found.
  – Location may be a register, a memory address, a stack location, or some set of more than one of these
  – Information can be stored in the symbol-table entry for that variable
simple code generation

For each three-address statement $x := y \text{ op } z$,
1. Invoke \textit{getreg} $(x := y \text{ op } z)$ to select registers $R_x$, $R_y$, and $R_z$.
2. If $R_y$ does not contain $y$, issue: “LD $R_y$, y' “, for a location $y'$ of $y$.
3. If $R_z$ does not contain $z$, issue: “LD $R_z$, z’ “, for a location $z'$ of $z$.
4. Issue the instruction “OP $R_x$,R_y,R_z”
5. Update the address descriptors of $x$, $y$, $z$, if necessary.
   - $R_x$ is the only location of $x$ now, and
   - $R_x$ contains only $x$ (remove $R_x$ from other address descriptors).
updating descriptors

• For the instruction \texttt{LD R, x}
  a) Change the register descriptor for register R so it holds only x.
  b) Change the address descriptor for x by adding register R as an additional location.

• For the instruction \texttt{ST x, R}
  – change the address descriptor for x to include its own memory location.
updating descriptors

• For an operation such as ADD Rx, Ry, Rz, implementing a 3AC instruction $x = y + z$
  a) Change the register descriptor for Rx so that it holds only $x$.
  b) Change the address descriptor for $x$ so that its only location is Rx. Note that the memory location for $x$ is not now in the address descriptor for $x$.
  c) Remove Rx from the address descriptor of any variable other than $x$.

• When we process a copy statement $x = y$, after generating the load for $y$ into register Ry, if needed, and after managing descriptors as for all load statements (rule 1):
  a) Add $x$ to the register descriptor for Ry.
  b) Change the address descriptor for $x$ so that its only location is Ry.
\[ t = A - B \\
LD \ R1,A \\
LD \ R2,B \\
\text{SUB} \ R2,R1,R2 \]

\[ u = A - C \\
LD \ R3,C \\
\text{SUB} \ R1,R1,R3 \]

\[ v = t + u \\
\text{ADD} \ R3,R2,R1 \]

\[ A B C D = \text{live outside the block} \]
\[ t,u,v = \text{temporaries in local storage} \]
A = D
LD R2, D

t = A - B
u = A - C
v = t + u
A = D
D = v + u

D = v + u
ADD R1, R3, R1

exit
ST A, R2
ST D, R1

A B C D = live outside the block
t, u, v = temporaries in local storage
Design of getReg

• Many possible design choices

• Simple rules
  – If $y$ is currently in a register, pick a register already containing $y$ as $R_y$. No need to load this register
  – If $y$ is not in a register, but there is a register that is currently empty, pick one such register as $R_y$

• Complicated case
  – $y$ is not in a register, but there is no free register
Design of getReg

• Instruction: $x = y + z$
• $y$ is not in a register, no free register
• let $R$ be a taken register holding value of a variable $v$
• possibilities:
  – if $v$ is $x$, the value computed by the instruction, we can use it as $R_y$ (it is going to be overwritten anyway)
  – if $v$ is not used later (dead), we can use $R$ as $R_y$
  – if the value $v$ is available somewhere other than $R$, we can allocate $R$ to be $R_y$ (just by updating descriptors).
  – otherwise: spill the value to memory by $\text{ST} \ v, R$
Global register allocation

• so far we assumed that register values are written back to memory at the end of every basic block

• want to save load/stores by keeping frequently accessed values in registers (e.g., loop counters)

• idea: compute “weight” for each variable
  – for each use of v in B prior to any definition of v add 1 point
  – for each occurrence of v in a following block using v add 2 points, as we save the store/load between blocks
  – \( \text{cost}(v) = \sum_B \text{use}(v, B) + 2 \cdot \text{live}(v, B) \)
    • use\((v, B)\) is the number of times v is used in B prior to any definition of v
    • live\((v, B)\) is 1 if v is live on exit from B and is assigned a value in B
  – after computing weights, allocate registers to the “heaviest” values
Example

1. Use of a prior to a def
2. 2 uses of b prior to a def
3. Use of c prior to a def
4. a defined in the block and live on exit
5. Use of a prior to a def
6. Use of c prior to a def
7. Use of c prior to a def
8. b defined and live on exit

- Use of a prior to a def:
  - \( f = a - d \)
  - cdef
- Use of c prior to a def:
  - \( b = d + f \)
  - acdf
- a defined in the block and live on exit:
  - \( cdef \)
- Use of a prior to a def:
  - \( b = d + c \)
- Use of c prior to a def:
  - \( e = a + f \)
- b defined and live on exit:
  - \( a = b + c \)

Costs:
- \( \text{cost}(a) = \Sigma_B \text{use}(a,B) + 2*\text{live}(a,B) = 4 \)
- \( \text{cost}(b) = 6 \)
- \( \text{cost}(c) = 3 \)
- \( \text{cost}(d) = 6 \)
- \( \text{cost}(e) = 4 \)
- \( \text{cost}(f) = 4 \)
Example

LD R1,b
LD R2,d

LD R3,c
ADD R0,R1,R3
SUB R2,R2,R1
LD R3,f
ADD R3,R0,R3
ST e, R3

LD R3,c
ADD R1,R2,R3
LD R3,f
SUB R3,R0,R3
ST e, R3

SUB R3,R0,R2
ST f,R3

ST b,R1
ST a,R2

B1

B2

B3

B4
Register Allocation by Graph Coloring

- Spilling operations are costly - minimize them.
- Finding register allocation that minimizes spilling is NP-Hard.
- We use heuristics from graph coloring.

Main idea
- register allocation = coloring of an interference graph
- every node is a variable
- edge between variables that “interfere” = are both live at the same time
- number of colors = number of registers
Example

• Analyze liveness
• Build interference graph
• If interfering vertices get the same register (color), then a spill is needed.
But note that interference is conservative

\[
\begin{align*}
a &= \text{read}(); \\
b &= \text{read}(); \\
c &= \text{read}(); \\
a &= a + b + c; \\
\text{if } (a<10) \{ \\
    d &= c + 8; \\
    \text{print}(c); \\
\} \text{ else if } (a<20) \{ \\
    e &= 10; \\
    d &= e + a; \\
    \text{print}(e); \\
\} \text{ else } \{ \\
    f &= 12; \\
    d &= f + a; \\
    \text{print}(f); \\
\}
\end{align*}
\]

print(d);
Example: Interference Graph

```
a = read();
b = read();
c = read();
a = a + b + c;
if (a<10) goto B2 else goto B3

d = c + 8;
print(c);
if (a<20) goto B4 else goto B5

e = 10;
d = e + a;
print(e);
f = 12;
d = f + a;
print(f);

print(d);
```
Summary: Code Generation

• Depends on the target language and platform.
• Basic blocks and control flow graph show program executions paths.
• Determining variable liveness in a basic block.
  – useful for many optimizations.
  – Most important use: register allocation.
• Simple code generation.
Optimizations
Optimization

- Improve performance.
- **Must maintain program semantics**
  - optimized program must be “observable” equivalent.
- In contrast to the name, we seldom obtain optimum.
- We do not improve an inefficient algorithm, we do not fix bugs.
- Classical question: how much time should we spend on compile-time optimizations to save on running time?
  - With parameters unknown.

- Optimize running time (most popular),
- Optimize size of code,
- Optimize memory usage,
- optimize energy consumption.
Where does inefficiency come from?

• Redundancy in original program:
  – Sometimes the programmer uses redundancy to make programming easier, knowing that the compiler will remove it.
  – Sometimes the programmer is not very good.

• Redundancy because of high level language:
  – E.g., accessing an array means computing \( i \times 4 \) inside a loop repeatedly.

• Redundancy due to translation.
  – The initial compilation process is automatic and not very clever.
Register Allocation by Graph Coloring

• variables that interfere with each other cannot be allocated the same register
• graph coloring
  – classic problem: how to color the nodes of a graph with the lowest possible number of colors
  – bad news: problem is NP-complete (to even approximate)
  – good news: there are pretty good heuristic approaches
Heuristic Graph Coloring

• idea: color nodes one by one, coloring the “easiest” node last
• “easiest nodes” are ones that have lowest degree
  – fewer conflicts
• algorithm at high-level
  – find the least connected node
  – remove least connected node from the graph
  – color the reduced graph recursively
  – re-attach the least connected node
  – color the re-attached node
Heuristic Graph Coloring

stack: $\varepsilon$

stack: $b$

stack: $cb$

stack: $acb$
Heuristic Graph Coloring
Heuristic Graph Coloring

Result: 3 registers for 6 variables
Can we do with 2 registers?
Heuristic Graph Coloring

• two sources of non-determinism in the algorithm
  – choosing which of the (possibly many) nodes of lowest degree should be detached
  – choosing a free color from the available colors
Heuristic Graph Coloring

• The above heuristic gives a coloring of the graph.
• But what we really need is to color the graph with a given number of colors = number of available registers.
• Many times this is not possible.
  – (Telling whether it is possible is NP-Hard.)
• We’d like to find the maximum sub-graph that can be colored.
• Vertices that cannot be colored will represent variables that will not be assigned a register.
Similar Heuristic

1. Iteratively remove any vertex whose degree < k (with all of its edges).
2. Note: no matter how we color the other vertices, this one can be colored legitimately!
Similar Heuristic

1. Iteratively remove any vertex whose degree < k (with all of its edges).
2. Note: no matter how we color the other vertices, this one can be colored legitimately!

4. Now all vertices are of degree >=k (or graph is empty)
5. If graph empty: color the vertices one-by-one as in previous slides. Otherwise,
6. Choose any vertex, remove it from the graph. Implication: this variable will not be assigned a register. Repeat this step until we have a vertex with degree <k and go back to (1).
Similar Heuristic

1. Iteratively remove any vertex whose degree < k (with all of its edges).
2. Note: no matter how we color the other vertices, this one can be colored legitimately!

4. Now all vertices are of degree >=k (or graph is empty)
5. If graph empty: color the vertices one-by-one as in previous slides. Otherwise,
6. Choose any vertex, remove it from the graph. Implication: this variable will not be assigned a register. Repeat this step until we have a vertex with degree <k and go back to (1).

Source of non-determinism: choose which vertex to remove in (6). This decision determines the number of spills.
More Running Time Optimization

• Need to understand how the run characteristics (which are often unknown).
  – Usually the program spends most of its time in a small part of the code. If we optimize that, we gain a lot.
  – Thus, we invest more in inner loops.
  – Example: place together functions with high coupling.

• Need to know the operating system and the architecture.

• We will survey a few simple methods first, starting with building a DAG.
Representing a basic block computation with a DAG

- Leaves are variable or constants, marked by their names of values.
- Inner vertices are marked by their operators.
- We also associate variable names with the inner vertices according to the computation advance.

\[(1)\]
\[
\begin{align*}
    t_1 & := 4 \times i \\
    t_2 & := a[t_1] \\
    t_3 & := 4 \times i \\
    t_4 & := b[t_3] \\
    t_5 & := t_2 \times t_4 \\
    t_6 & := \text{prod} + t_5 \\
    \text{prod} & := t_6 \\
    t_7 & := i + 1 \\
    i & := t_7 \\
    \text{if } i \leq 20 \text{ goto (1)}
\end{align*}
\]
Building the DAG

For each instruction  \( x: = y + z \)

- Find the current location of \( y \) and \( z \),
- Build a new node marked “+” and connect as a parent to both nodes (if such parent does not exist); associate this node with “\( x \)”
- If \( x \) was previously associated with a different node, cancel the previous association (so that it is not used again).
- Do not create a new node for copy assignment such as \( x := y \). Instead, associate \( x \) with the node that \( y \) is associated with.
  - Such assignments are typically eliminated during the optimization.
Using the DAG

\[(1)\]
\[
\begin{align*}
t_1 &:= 4 \times i \\
t_2 &:= a \left[ t_1 \right] \\
t_3 &:= 4 \times i \\
t_4 &:= b \left[ t_3 \right] \\
t_5 &:= t_2 \times t_4 \\
t_6 &:= \text{prod} + t_5 \\
\text{prod} &:= t_6 \\
t_7 &:= i + 1 \\
i &:= t_7 \\
\text{if } i \leq 20 \text{ goto (1)} \\
i &:= i + 1
\end{align*}
\]
**Uses of DAGs**

- Automatic identification of common expressions
- Identification of variables that are used in the block
- Identification of values that are computed but not used.
- Identifying computation dependence (allowing code movements)
- Avoiding redundant copying instructions.
Aliasing Problems

• What’s wrong about the following optimization?

\[
\begin{align*}
  x &:= a[i] \quad x := a[i] \\
  a[j] &:= y \quad \not= a[j] := y \\
  z &:= a[i] \quad z := x
\end{align*}
\]

• The problem is with the side effect due to aliasing.
• Typically, we conservatively assume aliasing: upon assignment to an array element we assume no knowledge in array entries.
• The problem is when we do not know if aliasing exists.
• Relevant to pointers as well.
• Relevant to routine calls when we cannot determine the routine side-effects.
• **Aliasing is a major obstacle for program optimizations.**
Optimization Methods

• In the following slides we review various optimization methods, stressing performance optimizations.
• Main goal: eliminate redundant computations.
• Some methods are platform dependent.
  – In most platforms addition is faster than multiplication.
• Some methods do not look useful on their own, but their combination is effective.
Basic Optimizations

• **Common expression elimination:**
  – DAG identifies common expressions in a basic block; we can eliminate repeated computation.
  – Next lecture: data flow analysis will determine common expressions across basic blocks.

• **Copy propagation:**
  – *Given an assignment* $x := y$, *we attempt to use* $y$ *instead of* $x$.
  – Possible outcome: $x$ becomes dead and we can eliminate the assignment.
Code motion

- Code motion is useful in various scenarios.
- Identify inner-loop code,
- Identify an expression whose sources do not change in the loop, and
- Move this code outside the loop!

while (x - 3 < y) {
  // ... instructions that do
  // not change x
}

\[
t_1 = x - 3;
\]

while (t_1 < y) {
  // ... instructions that do
  // not change x or t_1
}
Induction variables & Strength Reduction

- Identify loop variables, and their relation to other variables.
- Eliminate dependence on induction variables as much as possible
  (1) \( i = 0; \)
  (2) \( t_1 = i \times 4; \quad \rightarrow t_1 = t_1 + 4 \)
  (3) \( t_2 = a[t_1] \)
  (4) if (\( t_2 > 100 \)) goto (19)
  (5) …
  …
  (17) \( i = i + 1 \)
  (18) goto (2)
  (19) …
- Why is such code (including multiplication by 4) so widespread?

In many platforms addition is faster than multiplication (strength reduction)

\( t_1 \) must be initialized outside the loop

Not just S.R.! We have removed dependence of \( t_1 \) in \( i \).

Thus, instructions 1 and 17 become irrelevant.
Peephole (חור הצצה) Optimization

• Optimizing long code sequences is hard
• A simple and effective alternative (though not optimal) is peephole optimization:
  – Check a “small window” of code and improve only this code section.
  – Identify local optimization opportunities
  – Rewrite code “in the window”

• For example:
  – x := x * 1;
  – a := a + 0;
peephole optimizations

• Some optimizations that do not require a global view:
• Simplifying algebraic computations:
• \( x := x^2 \rightarrow x := x \times x \)
• \( x := x \times 8 \rightarrow x := x \ll 3 \)
• Code rearrangement:
  (1) if \( x == 1 \) goto (3)
  (2) goto (19)
  (3) …
  ↓
  (1) if \( x \neq 1 \) goto (19)
  (2) …
peephole optimizations

• Eliminate redundant instructions:
  (1) \(a := x\)
  (2) \(x := a\)
  (3) \(a := \text{someFunction}(a)\);
  (4) \(x := \text{someOtherFunction}(a, x)\);
  (5) \(\text{if } a > x \text{ goto (2)}\)

• Execute peephole optimizations within basic block only and do not elide the first instruction.
void quicksort ( m , n )
int m , n ; {
    int i , j ;
    int v , x ;
    if ( n <= m ) return ;
    i = m – 1 ; j = n ; v = a [ n ];
    while (1) {
        do i = i + 1 ; while ( a [ i ] < v ) ;
        do j = j – 1 ; while ( a [ j ] > v ) ;
        if ( i >= j ) break ;
        x = a [ i ] ; a [ i ] = a [ j ] ; a [ j ] = x ;
    }
    if ( n <= m ) return ;
    quicksort ( m , j ) ; quicksort ( i + 1 , n ) ; }

(1) i := m – 1
(2) j := n
(3) t_1 := 4 * n
(4) v := a [ t_1 ]
(5) i := i + 1
(6) t_2 := 4 * i
(7) t_3 := a [ t_2 ]
(8) if t_3 < v goto (5)
(9) j := j – 1
(10) t_4 := 4 * j
(11) t_5 := a [ t_4 ]
(12) if t_5 > v goto (9)
(13) if i >= j goto (23)
(14) t_6 := 4 * i
(15) x := a [ t_6 ]
(16) t_7 := 4 * i
(17) t_8 := 4 * j
(18) t_9 := a [ t_8 ]
(19) a [ t_7 ] := t_9
(20) t_{10} := 4 * j
(21) a [ t_{10} ] := x
(22) goto (5)
(23) t_{11} := 4 * i
(24) x := a [ t_{11} ]
(25) t_{12} := 4 * i
(26) t_{13} := 4 * n
(27) t_{14} := a [ t_{13} ]
(28) a [ t_{12} ] := t_{14}
(29) t_{15} := 4 * n
(30) a [ t_{15} ] := x
void quicksort ( m , n )
int m , n ; {
    int i , j ;
    int v , x ;
    if ( n <= m ) return ;
    i = m - 1 ; j = n ; v = a [ n ];
    while (1) {
        do i = i + 1 ; while ( a [ i ] < v ) ;
        do j = j - 1 ; while ( a [ j ] > v ) ;
        if ( i >= j ) break ;
        x = a [ i ] ; a [ i ] = a [ j ] ; a [ j ] = x ;
    }
    if ( n <= m ) return ;
    quicksort ( m , j ) ; quicksort ( i + 1 , n ) ;
}

(1) i := m - 1
(2) j := n
(3) t₁ := 4 * n
(4) v := a [ t₁ ]
(5) i := i + 1
(6) t₂ := 4 * i
(7) t₃ := a [ t₂ ]
(8) if t₃ < v goto (5)
(9) j := j - 1
(10) t₄ := 4 * j
(11) t₅ := a [ t₄ ]
(12) if t₅ > v goto (9)
(13) if i >= j goto (23)
(14) t₆ := 4 * i
(15) x := a [ t₆ ]
(16) t₇ := 4 * i
(17) t₈ := 4 * j
(18) t₉ := a [ t₈ ]
(19) a [ t₇ ] := t₉
(20) t₁₀ := 4 * j
(21) a [ t₁₀ ] := x
(22) goto (5)
(23) t₁₁ := 4 * i
(24) x := a [ t₁₁ ]
(25) t₁₂ := 4 * i
(26) t₁₃ := 4 * n
(27) t₁₄ := a [ t₁₃ ]
(28) a [ t₁₂ ] := t₁₄
(29) t₁₅ := 4 * n
(30) a [ t₁₅ ] := x
שלב א' ביטויי משותפים בשוני גלובלי

t_{11} := 4 \times i
x := a[t_{11}]
t_{12} := 4 \times i
a[t_{12}] := t_{13}
t_{13} := 4 \times n
a[t_{13}] := t_{14}
t_{14} := 4 \times i
a[t_{14}] := t_{15}
t_{15} := 4 \times n
a[t_{15}] := x
goto B_2

i := m - 1
j := n
t_1 := 4 \times n
v := a[t_1]
i := i + 1
t_2 := 4 \times i
t_3 := a[t_2]
if t_3 < v goto B_2
j := j - 1
t_4 := 4 \times j
t_5 := a[t_4]
if t_5 > v goto B_3
if i >= j goto B_6

t_6 := 4 \times i
x := a[t_6]
t_7 := 4 \times i
t_8 := 4 \times j
t_9 := a[t_8]
a[t_7] := t_9
t_{10} := 4 \times j
a[t_{10}] := x
goto B_2

t_{11} := 4 \times i
x := a[t_{11}]
t_{12} := 4 \times i
t_{13} := 4 \times n
a[t_{13}] := t_{14}
t_{14} := 4 \times i
a[t_{14}] := t_{15}
t_{15} := 4 \times n
a[t_{15}] := x
$t_7 := 4 * i$
$x := a [ t_6 ]$
$t_8 := 4 * j$
$t_9 := a [ t_8 ]$
$a [ t_7 ] := t_9$
$t_{10} := 4 * j$
$a [ t_{10} ] := x$
goto B_2

$t_{11} := 4 * i$
$x := a [ t_{11} ]$
$t_{12} := 4 * i$
$t_{13} := 4 * n$
$t_{14} := a [ t_{13} ]$
$a [ t_{12} ] := t_{14}$
$t_{15} := 4 * n$
$a [ t_{15} ] := x$
i := m – 1
j := n
t₁ := 4 * n
v := a [ t₁ ]

i := i + 1
t₂ := 4 * i
t₃ := a [ t₂ ]
if t₃ < v goto B₂

j := j – 1
t₄ := 4 * j
t₅ := a [ t₄ ]
if t₅ > v goto B₃

if i >= j goto B₆

t₆ := 4 * i
x := a [ t₆ ]
t₇ := 4 * i
t₈ := 4 * j
t₉ := a [ t₈ ]
a [ t₇ ] := t₉
t₁₀ := 4 * j
a [ t₁₀ ] := x
goto B₂

t₁₁ := 4 * i
x := a [ t₁₁ ]
t₁₂ := 4 * i
t₁₃ := 4 * n
t₁₄ := a [ t₁₃ ]
a [ t₁₂ ] := t₁₄
t₁₅ := 4 * n
a [ t₁₅ ] := x
\[
i := m - 1
\]
\[
j := n
\]
\[
t_1 := 4 * n
\]
\[
v := a[ t_1 ]
\]
\[
i := i + 1
\]
\[
t_2 := 4 * i
\]
\[
t_3 := a[ t_2 ]
\]
\[
\text{if } t_3 < v \text{ goto } B_2
\]
\[
j := j - 1
\]
\[
t_4 := 4 * j
\]
\[
t_5 := a[ t_4 ]
\]
\[
\text{if } t_5 > v \text{ goto } B_3
\]
\[
\text{if } i \geq j \text{ goto } B_6
\]
\[
t_{11} := 4 * i
\]
\[
x := a[ t_{11} ]
\]
\[
t_{12} := 4 * i
\]
\[
t_{13} := 4 * n
\]
\[
t_{14} := a[ t_{13} ]
\]
\[
a[ t_{12} ] := t_{14}
\]
\[
t_{15} := 4 * n
\]
\[
a[ t_{15} ] := x
\]

שתל ביטויים משותפים באופן גלובלי.


\[ t_{11} := 4 \cdot i \]
\[ x := a[t_{11}] \]
\[ t_{12} := 4 \cdot i \]
\[ t_{13} := 4 \cdot n \]
\[ t_{14} := a[t_{13}] \]
\[ a[t_{14}] := t_{14} \]
\[ t_{15} := 4 \cdot n \]
\[ a[t_{15}] := x \]

\[ t_{6} := 4 \cdot i \]
\[ x := a[t_{6}] \]
\[ t_{7} := 4 \cdot i \]
\[ t_{8} := 4 \cdot j \]
\[ t_{9} := a[t_{8}] \]
\[ a[t_{9}] := t_{9} \]
\[ t_{10} := 4 \cdot j \]
\[ a[t_{10}] := x \]

goto B_2

\[ t_{3} := a[t_{2}] \]
\[ \text{if } t_{3} < v \text{ goto } B_2 \]

\[ j := j - 1 \]
\[ t_{4} := 4 \cdot j \]
\[ t_{5} := a[t_{4}] \]
\[ \text{if } t_{5} > v \text{ goto } B_3 \]

\[ i := i + 1 \]
\[ t_{2} := 4 \cdot i \]
\[ t_{3} := a[t_{2}] \]
\[ \text{if } t_{3} < v \text{ goto } B_2 \]

\[ i := m - 1 \]
\[ j := n \]
\[ t_{1} := 4 \cdot n \]
\[ v := a[t_{1}] \]


goto B_2

\[ \text{if } i \geq j \text{ goto } B_2 \]

\[ j := j - 1 \]
\[ t_{4} := 4 \cdot j \]
\[ t_{5} := a[t_{4}] \]
\[ \text{if } t_{5} > v \text{ goto } B_3 \]

\[ i := i + 1 \]
\[ t_{2} := 4 \cdot i \]
\[ t_{3} := a[t_{2}] \]
\[ \text{if } t_{3} < v \text{ goto } B_2 \]

\[ i := m - 1 \]
\[ j := n \]
\[ t_{1} := 4 \cdot n \]
\[ v := a[t_{1}] \]
שלב 'א'
ביהול ביטויים משותפים בסוף שלב

```plaintext
i := m - 1
j := n
t_1 := 4 * n
v := a[t_1]

i := i + 1
{t_2 := 4 * i
t_3 := a[t_2]
if t_3 < v goto B_2

j := j - 1
t_4 := 4 * j
t_5 := a[t_4]
if t_5 > v goto B_3

if i >= j goto B_6

{t_6 := 4 * i
x := a[t_6]
t_7 := 4 * i
{t_8 := 4 * j
t_9 := a[t_8]
t_10 := 4 * j
{a[t_10] := t_9
t_{11} := 4 * i
x := a[t_{11}]
t_{12} := 4 * i
t_{13} := 4 * n
t_{14} := a[t_{13}]
a[t_{12}] := t_{14}
t_{15} := 4 * n
a[t_{15}] := x
goto B_2
```
\[
i := m - 1
j := n
t_1 := 4 \cdot n
v := a[t_1]
\]

\[
i := i + 1
\]

\[
t_2 := 4 \cdot i
t_3 := a[t_2]
\]

\[
\text{if } t_3 < v \text{ goto } B_2
\]

\[
j := j - 1
\]

\[
t_4 := 4 \cdot j
t_5 := a[t_4]
\]

\[
\text{if } t_5 > v \text{ goto } B_3
\]

\[
\text{if } i \geq j \text{ goto } B_6
\]

\[
t_6 := 4 \cdot i	x := a[t_6]
\]

\[
t_7 := 4 \cdot i
t_8 := 4 \cdot j
t_9 := a[t_8]
\]

\[
a[t_9] := t_5
a[t_10] := 4 \cdot j
\]

\[
t_{10} := 4 \cdot j
a[t_{10}] := x
\]

\[
\text{goto } B_2
\]

\[
t_{11} := 4 \cdot i
x := a[t_{11}]
\]

\[
t_{12} := 4 \cdot i
t_{13} := 4 \cdot n
t_{14} := a[t_{13}]
\]

\[
a[t_{14}] := t_{14}
t_{15} := 4 \cdot n
a[t_{15}] := x
\]

\[
\text{goto } B_2
\]
\( t_{11} := 4 \times i \)
\( x := a[t_{11}] \)
\( t_{12} := 4 \times i \)
\( t_{13} := 4 \times n \)
\( t_{14} := a[t_{13}] \)
\( a[t_{12}] := t_{14} \)
\( t_{15} := 4 \times n \)
\( a[t_{15}] := x \)

\( i := m - 1 \)
\( j := n \)
\( t_1 := 4 \times n \)
\( v := a[t_1] \)

\( i := i + 1 \)
\( t_2 := 4 \times i \)
\( t_3 := a[t_2] \)
if \( t_3 < v \) goto \( B_2 \)

\( j := j - 1 \)
\( t_4 := 4 \times j \)
\( t_5 := a[t_4] \)
if \( t_5 > v \) goto \( B_3 \)

if \( i \geq j \) goto \( B_6 \)
\[ t_{11} := 4 \cdot i \]
\[ t_{13} := 4 \cdot n \]
\[ t_{4} := 4 \cdot j \]
\[ t_{5} := a[t_{4}] \]
\[ t_{3} < v \Rightarrow \text{goto } B_3 \]

\[ t_{12} := 4 \cdot i \]
\[ t_{10} := 4 \cdot j \]
\[ a[t_{10}] := x \]
\[ \text{goto } B_2 \]

\[ t_{6} := 4 \cdot i \]
\[ x := a[t_{6}] \]
\[ t_{7} := 4 \cdot i \]
\[ t_{8} := 4 \cdot j \]
\[ t_{9} := a[t_{8}] \]
\[ a[t_{9}] := t_{5} a[t_{6}] := x \]

\[ i := m - 1 \]
\[ j := n \]
\[ t_{1} := 4 \cdot n \]
\[ v := a[t_{1}] \]

\[ i := i + 1 \]
\[ t_{2} := 4 \cdot i \]
\[ t_{3} := a[t_{2}] \]
\[ \text{if } t_{3} < v \Rightarrow \text{goto } B_2 \]

\[ j := j - 1 \]
\[ t_{4} := 4 \cdot j \]
\[ t_{5} := a[t_{4}] \]
\[ \text{if } t_{5} > v \Rightarrow \text{goto } B_3 \]

\[ \text{if } i \geq j \Rightarrow \text{goto } B_6 \]
\[ t_1 := 4 \times n \]
\[ v := a[t_1] \]

\[ i := i + 1 \]
\[ t_2 := 4 \times i \]
\[ t_3 := a[t_2] \]
\[ \text{if } t_3 < v \text{ goto } B_2 \]

\[ j := j - 1 \]
\[ t_4 := 4 \times j \]
\[ t_5 := a[t_4] \]
\[ \text{if } t_5 > v \text{ goto } B_3 \]

\[ i := m - 1 \]
\[ j := n \]
\[ t_1 := 4 \times n \]
\[ v := a[t_1] \]

\[ t_6 := 4 \times i \]
\[ x := a[t_6] \]
\[ t_7 := 4 \times i \]
\[ t_8 := 4 \times j \]
\[ t_9 := a[t_8] \]
\[ a[t_9] := t_4 \]
\[ a[t_9] := t_5 \]

\[ t_{10} := 4 \times j \]
\[ a[t_{10}] := x \]

\[ \text{goto } B_2 \]

\[ t_{11} := 4 \times i \]
\[ t_{12} := 4 \times i \]
\[ t_{13} := 4 \times n \]
\[ t_{14} := a[t_{13}] \]
\[ a[t_{14}] := t_{12} \]
\[ a[t_{14}] := t_{15} \]
\[ t_{15} := 4 \times n \]
\[ a[t_{15}] := x \]

\[ \text{goto } B_2 \]
\(t_{11} := 4 \times i\)
\(x := a[t_{11}]\)
\(t_{12} := 4 \times i\)
\(t_{13} := 4 \times n\)
\(t_{14} := a[t_{12}]\)
\(a[t_{13}] := t_{14}\)
\(a[t_{14}] := x\)
\(a[t_{15}] := x\)

\(t_6 := 4 \times i\)
\(x := a[t_6]\)
\(t_7 := 4 \times i\)
\(t_8 := 4 \times j\)
\(t_9 := a[t_8]\)
\(a[t_9] := t_5\)
\(a[t_10] := 4 \times j\)

\(i := m - 1\)
\(j := n\)
\(t_1 := 4 \times n\)
\(v := a[t_1]\)

\(i := i + 1\)
\(t_2 := 4 \times i\)
\(t_3 := a[t_2]\)
if \(t_3 < v\) goto B

\(j := j - 1\)
\(t_4 := 4 \times j\)
\(t_5 := a[t_4]\)
if \(t_5 > v\) goto B

if \(i \geq j\) goto B
if i >= j goto B

j := j – 1

$t_6 := 4 \times j$

if $t_6 > v$ goto B

i := i + 1

$t_2 := 4 \times i$

$t_3 := a[t_2]$

if $t_3 < v$ goto B

i := m – 1

j := n

$t_1 := 4 \times n$

v := a[t_1]

-- שלב א' – ביטוי ביטויים משותפים בזオープ גלובלי

ל cầu B.

copy propagation: with f:= g, we try to use g and get rid of f.
Global common expression elimination
Copy propagation
Dead code elimination – eliminate redundant code.
Global common expression elimination
Copy propagation
Dead code elimination
Code motion – move expressions outside the loop
Global common expression elimination
Copy propagation
Dead code elimination
Code motion
Induction variables and strength reduction
Global common expression elimination
Copy propagation
Dead code elimination
Code motion
Induction variables and strength reduction
Global common expression elimination
Copy propagation
Dead code elimination
Code motion
Induction variables and strength reduction
Global common expression elimination
Copy propagation
Dead code elimination
Code motion
Induction variables and strength reduction
Dead code elimination (again)
i := m – 1
goto B
j := n

\[ t_1 := 4 \ast n \]
\[ v := a[ t_1 ] \]
\[ t_4 := 4 \ast j \]
\[ t_2 := 4 \ast i \]

\[ t_2 := t_2 + 4 \]
\[ t_3 := a[ t_3 ] \]
if \( t_3 < v \) goto B_2

\[ t_4 := t_4 - 1 \]
\[ t_5 := a[ t_4 ] \]
if \( t_5 > v \) goto B_3

if \( t_2 \geq t_4 \) goto B_6

\[ a[t_2] := t_5 \]
\[ a[t_4] := t_3 \]
goto B_2

\[ t_11 := 4 \ast i \]
\[ x := a[ t_11 ] \]
\[ t_12 := 4 \ast i \]
\[ t_13 := 4 \ast n \]
\[ t_14 := a[ t_{13} ] \]
\[ a[t_{12}] := t_{14} \]
\[ t_{10} := 4 \ast j \]
\[ a[t_{10}] := x \]
\[ t_{14} := a[ t_1 ] \]
\[ a[t_2] := t_{14} \]
\[ a[t_1] := t_3 \]
goto B_2

Global common expression elimination

copy propagation
dead code elimination

code motion

induction variables and reduction in strength
Summary Optimizations

• Register allocation
• Optimizations
  – Not real “optimal” code, but code that is “better” in some respect
• Basic optimizations (common sub-expression elimination, induction variables, etc.)
• Peephole optimizations – often for better efficiency
Next... Data Flow Analysis