编译原理/Compiler
Liveness Analysis

周雅倩
复旦大学计算机科学技术学院
zhouyaqian@fudan.edu.cn
2018/12/13

References

- http://www.stanford.edu/class/cs143/
  - Partial contents are copied from the slides of Alex Aiken
  - CS 412/413 Spring 2005
- http://en.wikipedia.org/wiki/Optimizing_compiler

Outline

- Optimizing Compiler
- Control-flow Graph
  - Calculation of Liveness

Optimizations

- Code transformations to improve program
  - Mainly: improve execution time
  - Also: reduce program size
- Can be done at high level or low level
  - E.g., constant folding
- Optimizations must be safe
  - Execution of transformed code must yield same results as the original code for all possible executions

General Categories of Optimization

- Programming Language–independent VS Language-dependent
- Machine Independent VS Machine Dependent

Programming Language-independent VS Language-dependent

- Most high-level languages share common programming constructs and abstractions: decision (if, switch, case), looping (for, while, repeat.. until, do.. while), and encapsulation (structures, objects).
- Thus similar optimization techniques can be used across languages.
- However, certain language features make some kinds of optimizations difficult.
  - The existence of pointers in C and C++ makes it difficult to optimize array accesses (see alias analysis).
  - Some language features make certain optimizations easier.
  - Functional language optimizations.

Machine Independent VS Machine Dependent

- Many optimizations that operate on abstract programming concepts (loops, objects, structures) are independent of the machine targeted by the compiler.
- Many of the most effective optimizations are those that best exploit special features of the target platform.
- E.g.: Instructions which do several things at once, such as decrement register and branch if not zero.

Types of Optimizations

- Peephole optimizations
- Local optimizations
- Global optimizations
- Loop optimizations
- Interprocedural, whole-program or link-time optimization
- Machine code optimization

Constant folding

- Constant folding is the process of simplifying constant expressions at compile time.
- Terms in constant expressions are typically simple literals, such as the integer 2, but can also be variables whose values are never modified, or variables explicitly marked as constant.
- Consider the statement:

```
i = 320 * 200 * 32;
```


Constant propagation

- Constant propagation is the process of substituting the values of known constants in expressions at compile time. Such constants include those defined above, as well as intrinsic functions applied to constant values.

```
int a = 30;
int b = 9 - a / 5;
int c;
c = b * 4;
if (c > 10) {
  c = c - 10;
}
return c * (60 / a);
```

→

```
int a = 30;
int b = 3;
int c;
c = b * 4;
if (c > 10) {
  c = c - 10;
}
return c * 2;
```

→

```
int a = 30;
int b = 3;
int c;
c = 12;
if (true) {
  c = 2;
}
return c * 2;
```

→

```
int c;
if (true) {
  c = 2;
}
return c * 2;
```

↓

```
return 4;
```


Loop Optimization

Loop optimization is the process of the increasing execution speed and reducing the overheads associated of loops..

```
for(k = 1; k<100; k++){
    x = i+1;
    m = i+10*k;
    n = j+10*k;
}
```

→

```
k=1
label L1
if (100<k) goto L2
x=i+1
T1=10*k
m=i+T1
T2=10*k
n=j+T2
k=k+1
goto L1
label L2
```

→

```
x=i+1
m=i
n=j
k=1
label L1
if (100<k) goto L2
m=m+10
n=n+10
k=k+1
goto L2
label L2
```


Static Code Analyses

- Alias analysis
- Pointer analysis
- Shape analysis
- Escape analysis
- Array access analysis
- Dependence analysis
- Control flow analysis
- Data flow analysis
  - Use-define chain analysis
  - Live variable analysis
  - Available expression analysis

CONTROL-FLOW GRAPH

Live and Dead

- The first value of x is dead (never used)
- The second value of x is live (may be used)
- Liveness is an important concept

Optimization Safety

- Safety of code transformations usually requires certain information that may not be explicit in the code
- Example: dead code elimination

```
x = y + 1;  √        x = y + 1;  ?
y = 2 * z;           y = 2 * z;
x = y + z;           if (d) x = y+z;
z = 1;  √            z = 1;  √
z = x;               z = x;
```

What statements are dead and can be removed?

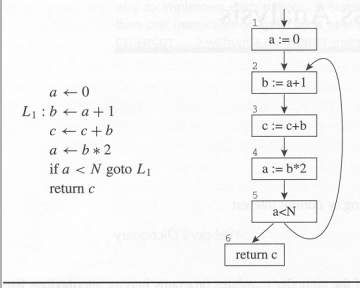
Optimizations and Control Flow

- Application of optimizations requires information
  - Dead code elimination: need to know if variables are dead when assigned values
- Required information:
  - Not explicit in the program
  - Must compute it statically (at compile-time)
  - Must characterize all dynamic (run-time) executions
- Control flow makes it hard to extract information
  - Branches and loops in the program
  - Different executions = different branches taken, different number of loop iterations executed

Control-Flow Graphs

- A control-flow graph is a directed graph with
  - Basic blocks as nodes (mostly)
  - An edge from block A to block B if the execution can flow from the last instruction in A to the first instruction in B
    - E.g., the last instruction in A is jump LB
    - E.g., the execution can fall-through from block A to block B
- Frequently abbreviated as CFG

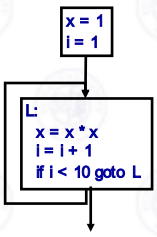
Example(1)



GRAPH 10.1. Control-flow graph of a program.

Nodes	Edges
1,2, 3,4, 5,6	1→2 2→3 3→4 4→5 5→2, 5→6

Example(2)



- The body of a method (or procedure) can be represented as a control-flow graph
- There is one initial node
- All "return" nodes are terminal

Liveness

- live - a variable that holds a value that may be needed in the future.
- A variable x is live at statement s if
  - There exists a statement s' that uses x
  - There is a path from s to s'
  - That path has no intervening assignment to x
- Discovering live variables is *liveness analysis*

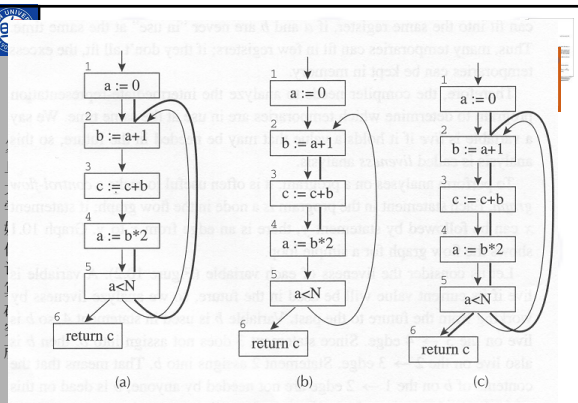


FIGURE 10.2. Liveness of variables a, b, c.

Computing Liveness

- We can express liveness in terms of information transferred between adjacent statements
- Liveness is a boolean property (true or false)

Flow-graph terminology

- out-edges
  - lead from node to successor node
  - 5→2 and 5 → 6 are out-edges of 5
- in-edge
  - come from a predecessor node
  - 5 → 2 is an in-edge of 2
- succ[n]
  - set of all successors of n
  - succ[5] = {2,6}
- pred[n]
  - set of all predecessors of n
  - pred[2] = {1,5}

Flow-graph terminology(cont.)

- use(n) n:node
  - set of all variables used (right-side) at a node
  - use(3) = {b,c}
- use(v) v:variable
  - set of graph nodes that use a variable
  - use(b)={3,4}

Flow-graph terminology(cont.)

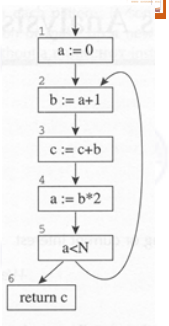
- def(n)
  - set of variables defined by a graph node
  - def(4) = {a}
- def(v)
  - set of graph nodes that define a variable
  - def(a) = {1,4}

Flow-graph terminology(cont.)

- Liveness
  - variable live on edge if directed path from edge to a use that does not go through any def
  - b live on 2 → 3 and 3 → 4
- live-in
  - at a node if live on any in-edges
- live-out
  - at a node if live on any out-edges

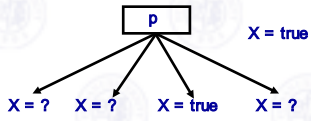
Exercise (1)

- What is succ[3]? pred[3]?
- Variables in: use(3)? def(3)?
- Nodes in: use(b)? def(b)?
- What variables are live-in and live-out at nodes 1, 2, 5?



Liveness Rule 1

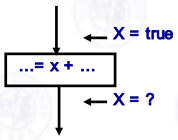
- $L_{out}(x, p) = \cup_{\text{for all } i} \{ L_{in}(x, s_i) \mid s_i \text{ a successor of } p \}$



L is a Boolean variable to represent liveness.

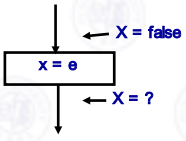
Liveness Rule 2

- $L_{in}(x, s) = true$ if s refers to x on the rhs



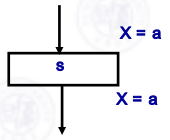
Liveness Rule 3

- $L_{in}(x, x = e) = false$ if e does not refer to x



Liveness Rule 4

- $L_{in}(x, s) = L_{out}(x, s)$ if s does not refer to x



Calculation of Liveness

- Liveness, *live-in* and *live-out*, calculated from *use*, *def* and *succ*
- Variable is:

1. *live-in* at node *n* if variable in *use[n]*
   - use[2]={a}, *a* is live-in at 2
   - use[3]={c,b} c and b are live-in at 3

Calculation of Liveness(cont.)

2. *live-out* at all nodes *m* in *pred[n]* if variable *live-in* at *n*
   - pred[2]={1,5}, *a* is live-out at 1 and 5
   - pred[3]={2}, *c* is live-out at 2 since live-in at 3

Calculation of Liveness(cont.)

3. *live-in* at node *n* if variable *live-out* at *n* and not in *def[n]*.
   - def[2] = {b}, *c* live-out at 2 so *live-in* at 2
   - def[1] = {a}, c is live-out at node 1 by rule 2., c is live-in at node 1 by rule 3.

Computing Liveness Set

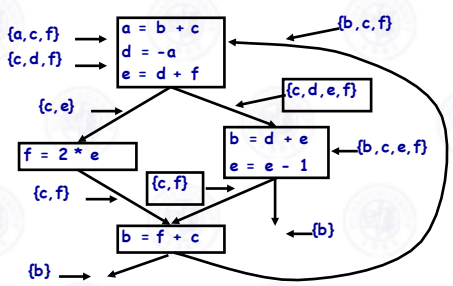
- We have the following equations
  - $in[n] = use[n] \cup (out[n] - def[n])$
  - $out[n] = \cup_{s \in succ[n]} in[s]$

Time Complexity

- For N nodes and N variables
- Each *for* iterates for N nodes.
- Each union is O(N) for N variables; consider merging two lists of length N
- *for* is then $O(N^2)$
- Each repeat iteration must add something to *in* or *out* sets. Each of N *in* and *out* sets could have at most N variables each, if only one *in* or *out* variable changes per repeat then $2N^2$ the maximum or $O(N^2)$.
- Worst case is then $O(N^4)$.
- Ordering usually reduces repeat to 2 or 3 so algorithm runs in $O(N^2)$

Example

Compute live variables for each point



Other varieties

- Basic blocks - Recall that basic blocks have no internal labels or jumps. Nodes can be merged to produce a graph with fewer nodes.
- One variable at a time - ALGORITHM 10.4 calculates liveness for all variables in parallel. Liveness of individual variables can be calculated by:
  - start at each *use* node in flow-graph for variable
  - trace back via *predecessor* edges (using depth-first search) noting liveness of variable
  - stop when *def* of node contains variable

Representation of Sets

- Dataflow equations (*in*, *out*, etc.) require set representation.
  - Sorted lists of variables - a list for each set. Union done by merging lists and eliminating duplicates.
  - Arrays of bits – a bit entry for each *in, out, def, use* set. *pred* and *succ* (assuming can branch to any other node) require another bit entry). Union done by bitwise OR.
  - HashSet class in Java is a convenient solution.

Least Fixed Points

- Solutions to EQUATION 10.3 are conservative, may include *live-out* variables that are not used
- Including *extra* variables in solution may prevent reallocating registers but will compute the right answer.
- TABLE 10.7 illustrates 3 possible solutions to EQUATION 10.3

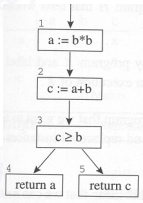
	use	def	X in	X out	Y in	Y out	Z in	Z out
1		a	c	ac	cd	acd	c	ac
2	a	b	ac	bc	acd	bcd	ac	b
3	bc	c	bc	bc	bcd	bcd	b	b
4	b	a	bc	ac	bcd	acd	b	ac
5	a		ac	ac	acd	acd	ac	ac
6	c		c		c		c	

TABLE 10.7. *X* and *Y* are solutions to the liveness equations; *Z* is not a solution.

- X - is least solution
- Y - includes variable d not used in program fragment but satisfies EQUATION 10.3
- Z - fails to satisfy EQUATION 10.3; indicates b and c are never live at same time.

Static Vs. Dynamic Liveness

- b*b positive
- c >= b always true
- node 4 never reached
- *a* not used after node 2
- *a* not *live-out* of node 2
- the above algorithm says *a* live-in at node 4 and therefore live-out at nodes 2 and 3
- In general can not determine control flow in all cases



Conservative Approximation

- Dynamic liveness
  - A variable *a* is dynamically live at node *n* if some execution of the program goes from *n* to a use of *a* without going through any definition of *a*.
- Static liveness
  - A variable *a* is statically live at node *n* if there is some path of control-flow graph edges from *n* to a use of *a* without going through any definition of *a*.

Interference Graphs

- Interference - condition that prevents allocation of two variables to same register (e.g. *live-out* of same node)
- Most common interference - overlapping *live* ranges; when live at same node, require different registers
- Other - when instruction must address a specific register. Example: Push/Pop only addresses eSp.

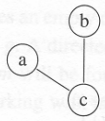
Example

- Figure below shows *ac* and *bc* interfere

	a	b	c
a			x
b			x
c	x	x	

(a) Matrix

	in	*out*
c		ac
ac		bc
bc		bc
bc		ac
ac		ac
c		



(b) Graph

Special treatment of MOVE instructions

- Any non-MOVE instruction that defines *a*, where *live-out* variables are $b_1,\ldots, b_j$, add interference edges (a, b_1), ..., (a, b_j)
- At MOVE a←c where variables $b_1,\ldots, b_j$ are *live-out*, add interference edges (a, b_1), ..., (a, b_j) for any b_i that is *not* the same as *c*.

Analysis

- There are many other global flow analyses
- Most can be classified as either forward or backward
- Most also follow the methodology of local rules relating information between adjacent program points

Forward vs. Backward Analysis

- We've seen one kinds of analysis, the backwards analysis:
  - information is pushed from outputs back towards inputs
  - Liveness is an example
- There is another kind of analysis the forwards analysis:
  - a information is pushed from inputs to outputs
  - Constant propagation is an example

Global Analysis

- Global optimization tasks share several traits:
  - The optimization depends on knowing a property X at a particular point in program execution
  - Proving X at any point requires knowledge of the entire function body
  - It is OK to be conservative. If the optimization requires X to be true, then want to know either
    - X is definitely true
    - Don't know if X is true
  - It is always safe to say "don't know"

Global Analysis (Cont.)

- Global dataflow analysis is a standard technique for solving problems with these characteristics.
- Global constant propagation is one example of an optimization that requires global dataflow analysis.