

CMPT 379

Compilers

Anoop Sarkar

<http://www.cs.sfu.ca/~anoop>

11/27/07

1

Code Generation

- Instruction selection
- Register allocation
- Stack frame allocation ✓
- Static or global allocation ✓
- Basic blocks and Flow graphs
- Transformations on Basic blocks

11/27/07

2

Code Generation

- Produce code that is correct
- Produce code that is of high quality (size and speed)
- The problem of generating optimal code is *undecidable*
- In practice, we need heuristics that generate good, but perhaps not optimal, code

11/27/07

3

Instruction Costs

- Since optimal code generation is not possible a useful way to think about the problem is as an *optimization* problem
- Each instruction can be assigned a cost
 - For complex instruction sets some instructions can be more preferable than others
- Using registers have zero cost, while using memory locations is costlier
- If each instruction is equally expensive, this will minimize the number of instructions as well

11/27/07

4

Register Allocation

- Code generation either directly to assembly or from 3-address code (TAC)
- For each location, we have to find a register to store values or temporary values
 - Problem: limited number of registers
- Compiler has to find optimal assignment of locations to registers
 - Register use can involve stacked temporaries or other ways to reuse registers
- If no more registers available, we *spill* a location into memory

11/27/07

5

Register Allocation

- Bind locations to registers for all or part of a function
- Dynamic Optimization Problem
 - Not compile-time, but run-time frequency is what counts
- Heuristics
 - Allocate registers for variables likely to be used frequently
 - Keep temporaries in registers → minimize their number
- Register Allocation using **Liveness Analysis**

11/27/07

6

Basic Blocks

- Functions transfer control from one place (the caller) to another (the called function)
- Other examples include any place where there are branch instructions
- A *basic block* is a sequence of statements that enters at the start and ends with a branch at the end
- Remaining task of code generation is to create code for basic blocks and branch them together

11/27/07

7

Blocks

```
main()
{
    int a = 0; int b = 0;
    {
        int b = 1;
        {
            int a = 2; printf("%d %d\n", a, b);
        }
        {
            int b = 3; printf("%d %d\n", a, b);
        }
        printf("%d %d\n", a, b);
    }
    printf("%d %d\n", a, b);
}
```

11/27/07

8

Partition into Basic Blocks

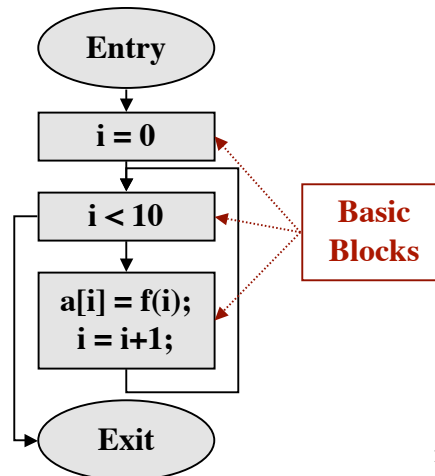
- Input: sequence of TAC instructions
 1. Determine set of leaders, the 1st statement of each basic block
 - a) The 1st statement is a leader
 - b) Any statement that is the target of a conditional jump or goto is a leader
 - c) Any statement immediately following a conditional jump or goto is a leader
 2. For each leader, the basic block contains all statements upto the next leader

11/27/07

9

Control Flow Graph (CFG)

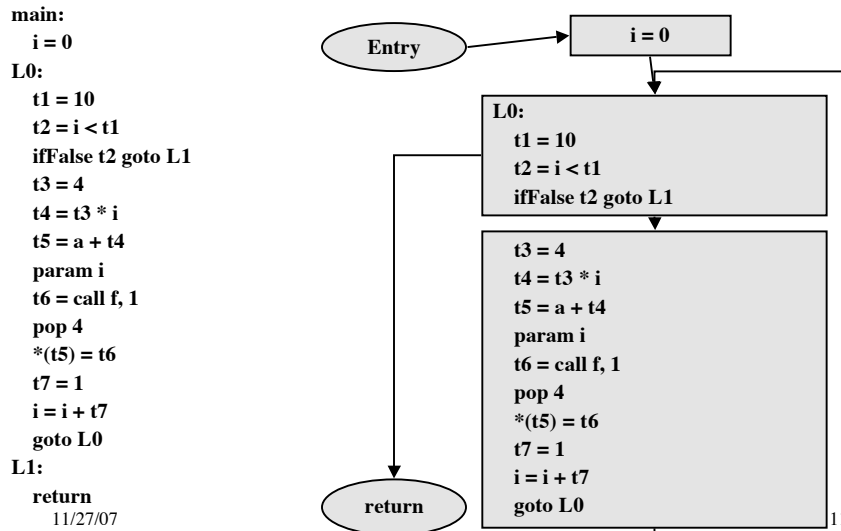
```
int main() {  
  extern int f(int);  
  int i;  
  int *a;  
  for (i = 0;  
       i < 10;  
       i = i + 1)  
    { a[i] = f(i); }  
}
```



11/27/07

10

Control Flow Graph in TAC



Dataflow Analysis

- Compute Dataflow Equations over Control Flow Graph
- in = all variables coming into basic block
 - def = variable is defined, e.g. $x := 0$
 - use = variable is used, e.g. $y := x + 1$
- out = all variables going out of basic block
- Liveness Analysis:

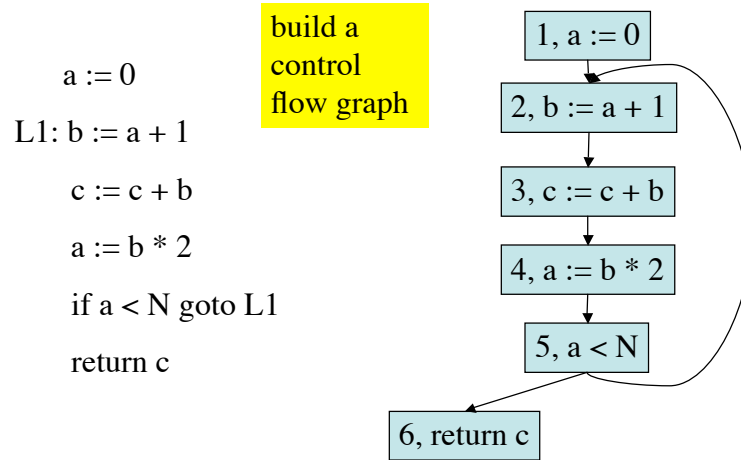
$$\text{in}[\text{BB}] := \text{use}[\text{BB}] \cup (\text{out}[\text{BB}] - \text{def}[\text{BB}])$$

$$\text{out}[\text{BB}] := \bigcup \text{in}[s] : \text{forall } s \in \text{succ}[\text{BB}]$$
- Computation by fixed-point analysis

11/27/07

12

Liveness Analysis



11/27/07

13

Liveness Analysis

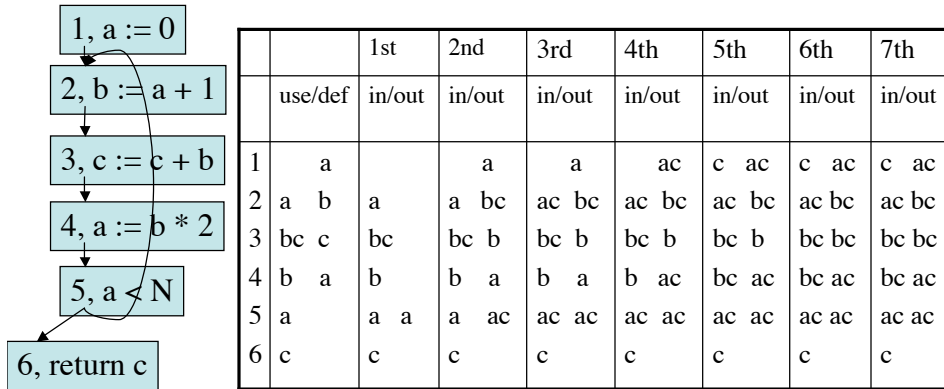
- Liveness Analysis:
 - $\text{in}[\text{BB}] := \text{use}[\text{BB}] \cup (\text{out}[\text{BB}] - \text{def}[\text{BB}])$
 - $\text{out}[\text{BB}] := \cup \text{in}[s] : \text{forall } s \in \text{succ}[\text{BB}]$
- Fixed point computation:
 - for each n : $\text{in}[n] := \{\}$; $\text{out}[n] := \{\}$
 - repeat
 - for each n :
 - $\text{in}'[n] := \text{in}[n]$; $\text{out}'[n] := \text{out}[n]$
 - $\text{in}[n] := \text{use}[n] \cup (\text{out}[n] - \text{def}[n])$
 - $\text{out}[n] := \cup \text{in}[s] : \text{forall } s \in \text{succ}[n]$
 - until $\text{in}'[n] == \text{in}[n] \ \&\& \ \text{out}'[n] == \text{out}[n]$ for all n

11/27/07

14

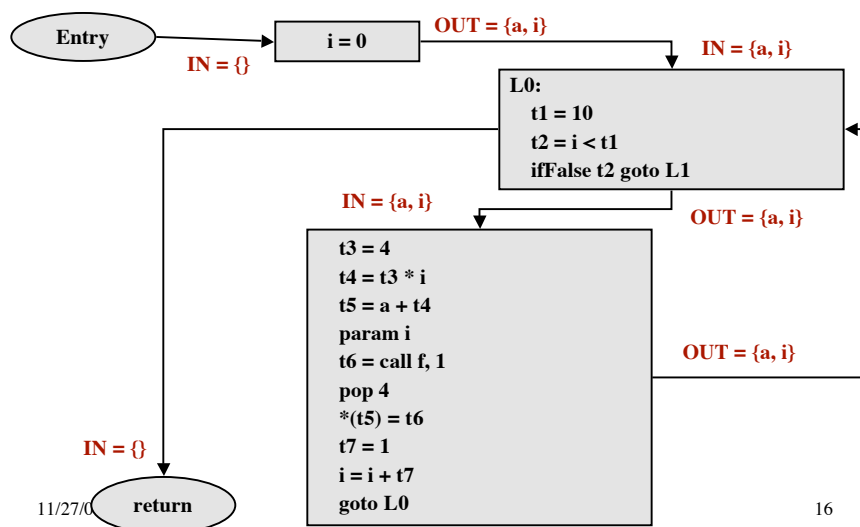
$in[BB] := use[BB] \cup (out[BB] - def[BB])$
 $out[BB] := \cup in[s] : \text{forall } s \in succ[BB]$

Liveness Analysis



11 can we do this faster? try going from 6 down to 1 instead 15

Liveness Analysis



11/27/0

16

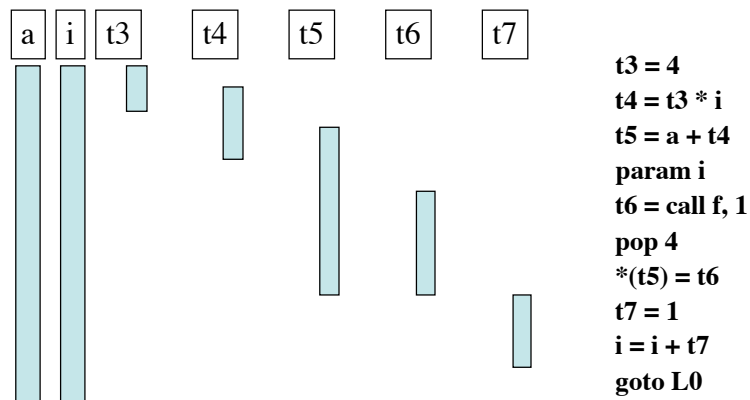
Register Allocation

- Do liveness analysis on Control Flow Graph
 - Straightforward (iteration-less) computation within basic block
 - Compute live ranges for each location
- Build interference graph
 - Two locations are connected if their live ranges overlap

11/27/07

17

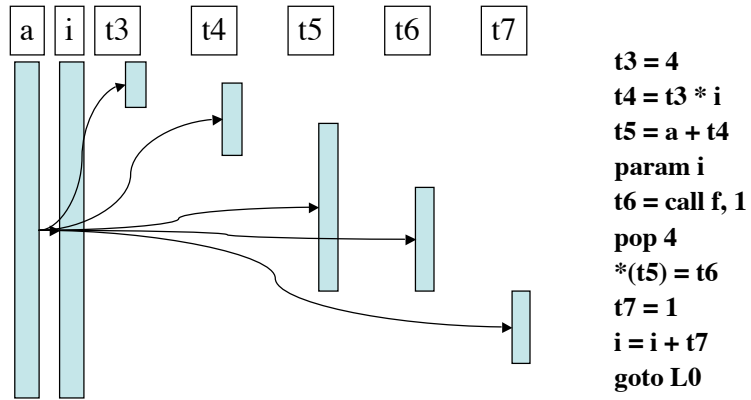
Register Allocation



11/27/07

18

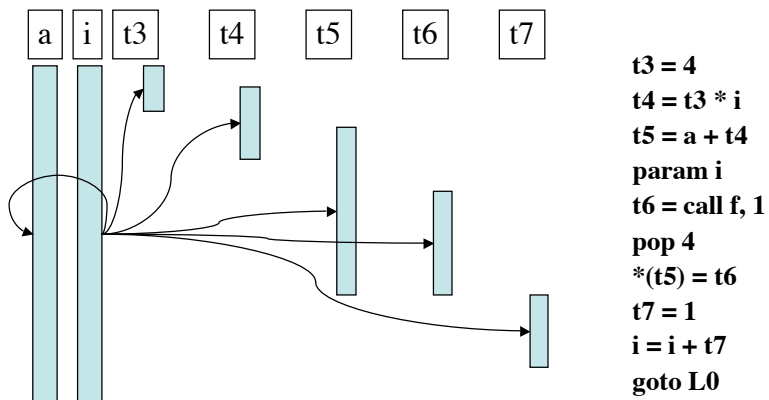
Register Allocation



11/27/07

19

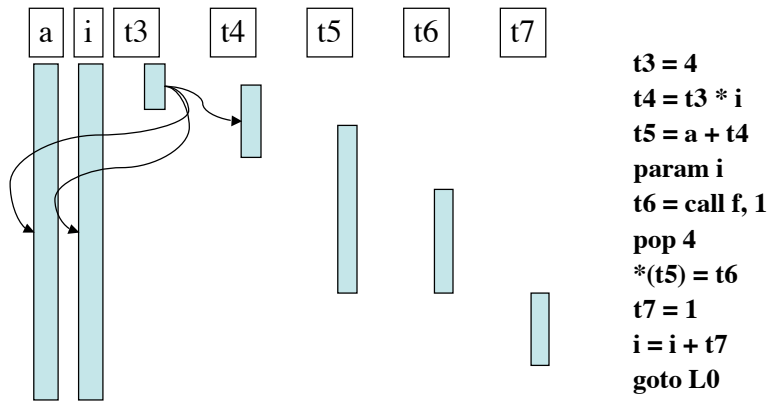
Register Allocation



11/27/07

20

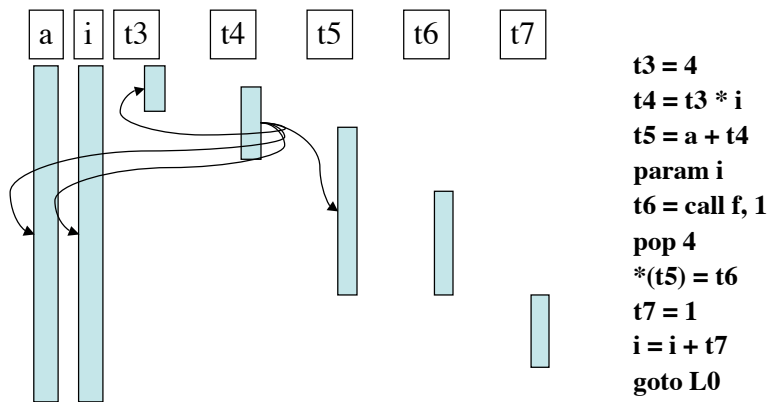
Register Allocation



11/27/07

21

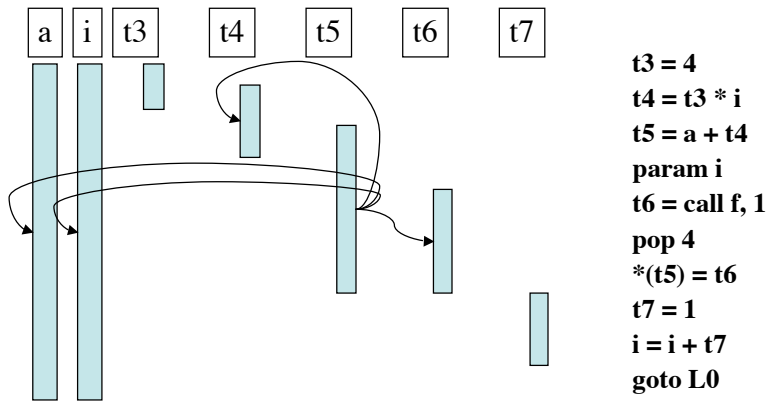
Register Allocation



11/27/07

22

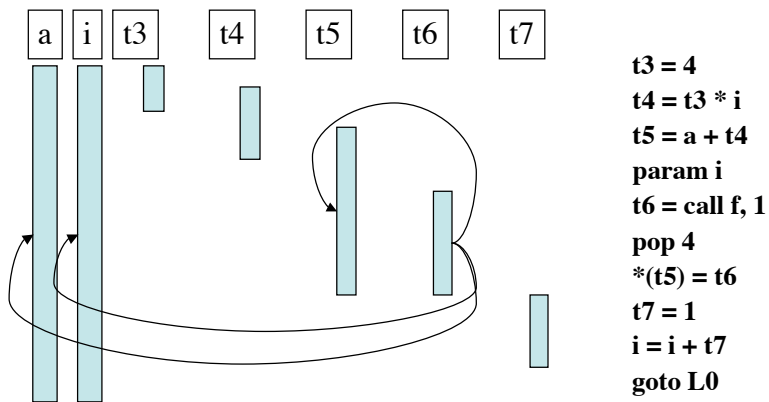
Register Allocation



11/27/07

23

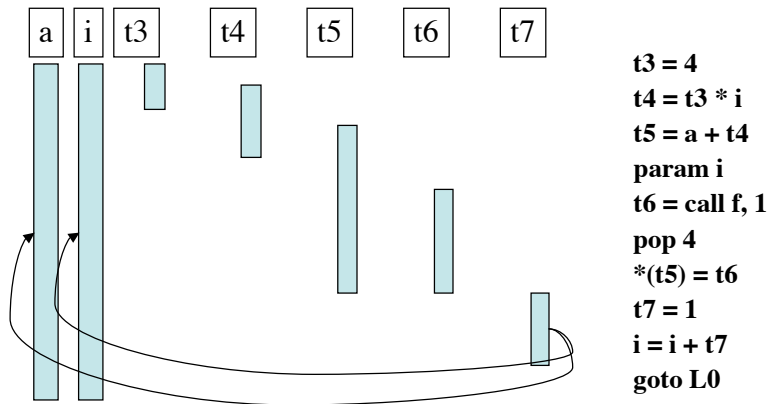
Register Allocation



11/27/07

24

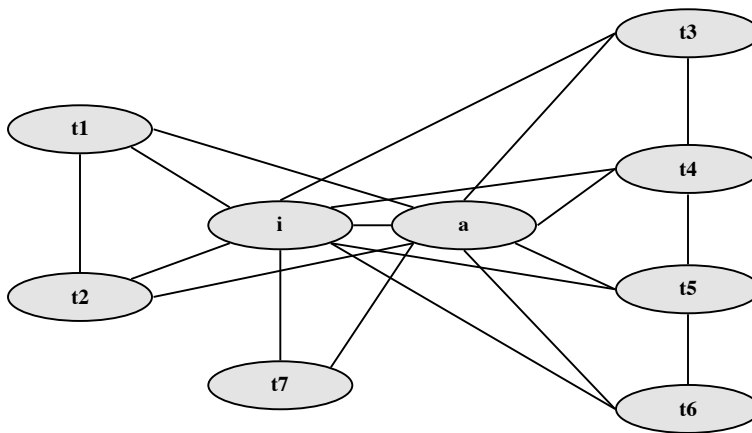
Register Allocation



11/27/07

25

Interference Graph



11/27/07

26

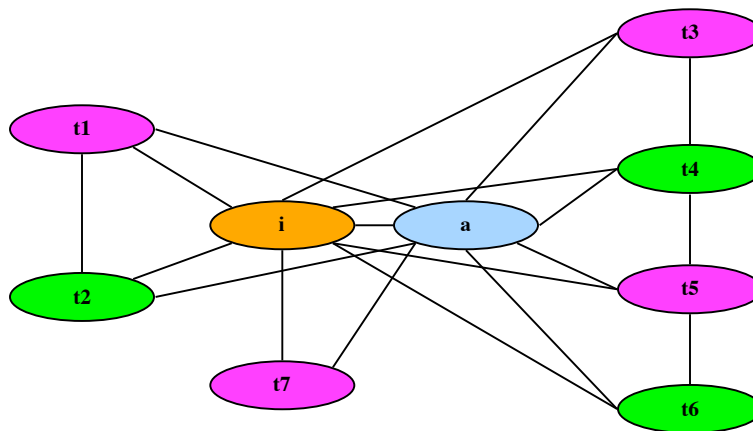
Interference Graph

- Assume we have four registers: 1, 2, 3, 4
- By register allocation we mean: assign each register to a node in the interference graph
- However, we cannot assign the same register to two nodes connected by an edge
- If we have an algorithm that can color a graph with 4 colors, we have a register allocation algorithm!

11/27/07

27

Colored Interference Graph



11/27/07

28

Register Allocation as Graph Coloring

- First pass: use as many symbolic registers as needed including registers for stack pointers, frame pointers, etc.
- Register Interference Graph
 - Two nodes in the graph are connected if their live ranges overlap
- Color interference graph
 - Result is register assignment -- k colors for k registers

11/27/07

29

Register Allocation as Graph Coloring

- Second pass: assign physical registers to symbolic ones
 - Construct a **register interference graph** (nodes are symbolic registers and edge denotes that they cannot be assigned to the same physical register)
 - Attempt to k -color the interference graph, where k is the number of available registers
 - k -coloring a graph is NP-complete

11/27/07

30

Register Allocation as Graph Coloring

- Algorithm for solving whether a graph G is k -colorable:
- Pick any node n with fewer than k neighbours
- Remove n and adjacent edges to create a new graph G'
- k -coloring of G' can be extended to k -coloring to G by assigning to n a color that is not assigned to any of n 's neighbours
- If we cannot extend G' to G , then k -coloring of G is not possible

11/27/07

31

Register Allocation as Graph Coloring

- If every node in G has more than k neighbours, k -coloring of G is not possible
- Take some node n and spill into memory, remove it from the graph and continue k -coloring
- Spilling = generating code to store contents of register to memory and when location is used generate code to load from memory into an available register (by spilling another location)

11/27/07

32

Register Allocation as Graph Coloring

- Many different heuristics for picking a node n to spill
- E.g. avoid introducing spilling symbolic registers that are inside loops or heavily visited regions of code
- C allows a *register* and a *volatile* keyword to direct the compiler whether a variable contains a value that is heavily used.
- Special case: Register Allocation for Expression Trees (Maximal Munch suffices for this task)

11/27/07

33

Summary

- Code generation: from Intermediate Representation (IR) to Assembly
- Three Address Code (TAC) can be easily converted to a *control flow graph*
- The control flow graph allows sophisticated dataflow analysis
- The liveness of each location can be used for register allocation
- Register Allocation as heuristic graph coloring.

11/27/07

34