Register Allocation & Liveness Analysis

CS502

Purdue University is an Equal Opportunity/Equal Access institution.

- In IR tree code generation, we use an unlimited number of TEMP's for local scalar variables and temporary results.
  - In the final assembly code, these TEMP's must be assigned to a limited number of hardware registers which can be used in the instructions.
- The values of two TEMP's cannot reside in the same hardware register if they may be both *live* at the same time.

.

Purdue University is an Equal Opportunity/Equal Access institution.

How many hardware registers are needed?

- Even for a single assignment statement with an arbitrarily long right-hand expression, we can't find a bound on the number of registers required to store the intermediate results
- This can be easily proved by mathematical induction applied to an arbitrarily deep binary operation tree:
  - For an n-deep binary tree, we need n registers, in the worst case, to complete the computation

Purdue University is an Equal Opportunity/Equal Access institution.

Live Variables

- The following is the definition of static liveness, which is conservative:

  A variable is said to be live at program point p if there exist an execution path from p to the program exit (or the procedure exit if the analysis is done at procedure level) in which the variable may be read before it is rewritten. Otherwise, it is said to be dead at the point p.

- We will discuss how to analyze the liveness of variables, i.e. in what program points a variable remains live, when we cover dataflow analysis

Purdue University is an Equal Opportunity/Equal Access institution.

Interference Graph

If two variables may be simultaneously live at any program point, we say they interfere with each other in register allocation.

We construct an interference graph to identify interferences between variables

– Each node represents a variable

– Each edge represents an interference

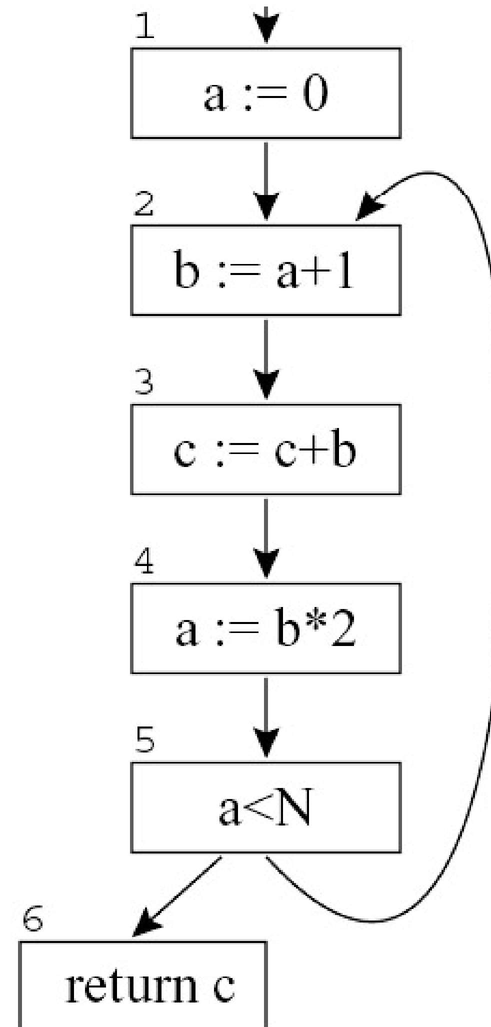
Purdue University is an Equal Opportunity/Equal Access institution.

- For the next example (a function body and its flow graph), we first intuitively decide what are the live variables at each program point.
- We then draw its interference graph
  - Later, we will introduce a compiler algorithm to determine the live variables.
- We assume that by compiler analysis, we have decided the variables are not aliases to each other. Hence we can directly allocate them to registers
  - We will introduce alias analysis in later lectures

Purdue University is an Equal Opportunity/Equal Access institution.

Example:

$$
\begin{aligned}
& a \leftarrow 0 \\
L_1: \; & b \leftarrow a + 1 \\
& c \leftarrow c + b \\
& a \leftarrow b * 2 \\
& \text{if } a < N \text{ goto } L_1 \\
& \text{return } c
\end{aligned}
$$



Purdue University is an Equal Opportunity/Equal Access institution.

- We have used liveness information to construct an interference graph for register allocation.
- We now discuss compiler algorithms for analyzing live variables.
- Recall the definition of live variables:

A variable is said to be *live* at program point *p* if there exist an execution path from *p* to the program exit (or the procedure exit if the analysis is done at procedure level) in which the variable may be read before it is rewritten. Otherwise, the variable is said to be *dead* at the point *p*.

Purdue University is an Equal Opportunity/Equal Access institution.

What does the algorithm compute?

For each basic block *B*, we want to determine

- LVin(B): the set of variables live right before B
- LVout(B): the set of variables live right after B

(remember: “live” means having a potential use in the future)

- To compute these, we use two pieces of information from *B*:
  - USE(B): is the set of variable that are used in B before they ever get re-written in B, i.e. having upwardly exposed uses.
  - DEF(B): is the set of variables rewritten in B

Purdue University is an Equal Opportunity/Equal Access institution.

Basic Equations

$$LVin(B) = (LVout(B) - DEF(B)) \cup USE(B)$$
$$LVout(B) = \cup_{S \in Succ(B)} LVin(S)$$

- If we have n basic blocks, then we have simultaneous equations over 2 * n variables: LVin(B) and LVout(B) for all B.
- Borrow the idea from numerical algorithms, we can solve this iteratively
- DEF(B) and USE(B) are “constants”
- Initialize all LVin and LVout variables to empty sets.

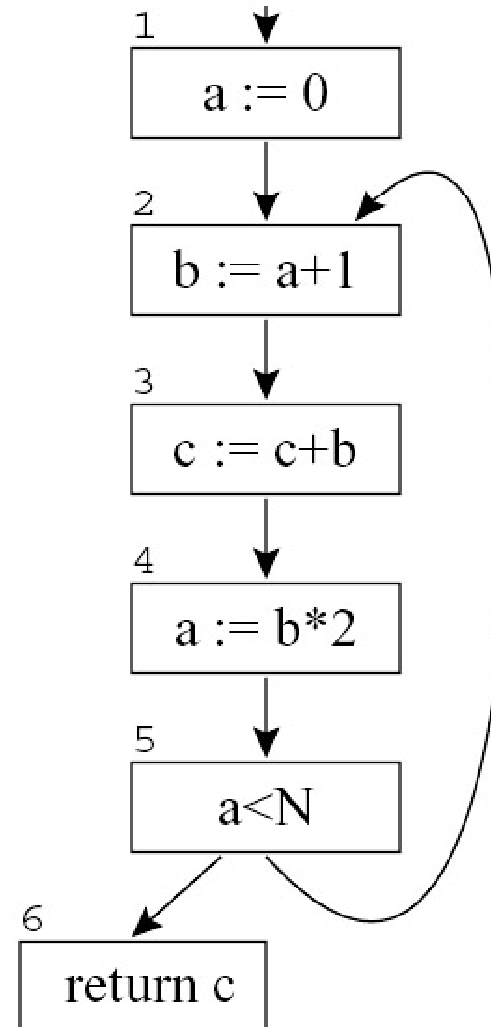
Purdue University is an Equal Opportunity/Equal Access institution.

- The two equations become two operators (or two filters) that are monotonically non-decreasing for each B
- The variables are bounded from above ➔ The iterations must converge, reaching the “fixed point”, regardless of the order to apply the operators in each iteration
- However, some order makes the convergence faster
- In the following examples, we make B simple, to contain a single instruction
- Let us apply our algorithm to an example we saw before.

Purdue University is an Equal Opportunity/Equal Access institution.

Example:

$$
\begin{aligned}
& a \leftarrow 0 \\
L_1 : \ & b \leftarrow a + 1 \\
& c \leftarrow c + b \\
& a \leftarrow b * 2 \\
& \text{if } a < N \text{ goto } L_1 \\
& \text{return } c
\end{aligned}
$$



Purdue University is an Equal Opportunity/Equal Access institution.

Algorithm Using a Work List

- In the naïve implementation, the algorithm iterates over all basic blocks until all blocks see no changes.
- A better implementation uses a worklist
- Initially include all basic blocks in the worklist.
- Until the worklist becomes empty, remove a basic block, B, from the list and recompute LV_{out} and LV_{in}.
  - If LV_{in} changes, then for each predecessor, P, of B
    - If P is not yet in the worklist, append P to the worklist
- We can redo the example using this implementation
- Worst case complexity?

Purdue University is an Equal Opportunity/Equal Access institution.

- If the basic blocks contain more than a single instruction
- Then the liveness information within each basic block, B, can be easily computed “locally”, using information in $LV_{out}(B)$.
- We revisit the previous example, but create a bigger basic block than before

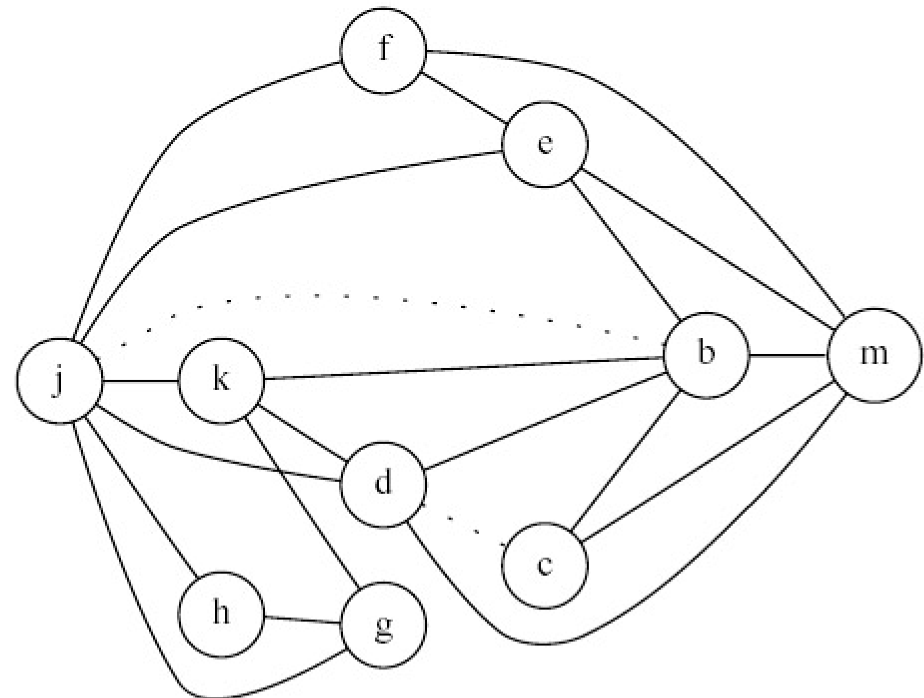
Purdue University is an Equal Opportunity/Equal Access institution.

- Next we see another example, which is a segment of a function body, assuming we continue to compute liveness information from the end of this program segment
- We build the interference graph for this program segment alone

Purdue University is an Equal Opportunity/Equal Access institution.

Example:

```
live-in: k j
      g := mem[j+12]
      h := k - 1
      f := g * h
      e := mem[j+8]
      m := mem[j+16]
      b := mem[f]
      c := e + 8
      d := c
      k := m + 4
      j := b
live-out: d k j
```



Purdue University is an Equal Opportunity/Equal Access institution.

The main objective of register allocation is to change as many memory references into register references. This is important for two main reasons:

- Accessing an operand from a register is faster than accessing it from the memory.
- Modern CPU hardware can easily detect dependencies between register operands. Hence, parallel operations and data-forwarding (in pipeline) can be exploited by the hardware.

Unfortunately, register allocation, as an optimization problem, is NP-complete except in special cases (e.g. when the targeted code segment can be represented as an expression tree without MOVE's).

Register allocation is one of the examples of scheduling limited resources to minimize the cost (or maximize the efficiency) of computation.

Resource scheduling problems can often be modeled by mathematical programming, e.g. linear integer programming. In a compiler, however, it is often desirable to use methods which are simpler to implement in a compiler.

Graph coloring is a relatively simple method which can be used for some of the scheduling problems, e.g. for register allocation.

To apply graph-coloring to register allocation, we first need to construct an *interference graph*, as discussed in the last chapter.

Next, we *color* the interference graph using K different colors, where K is the number of registers available for allocation. No pair of nodes which are connected by an edge may be assigned the same color.

If it is impossible to color the interference graph with the given K colors, then we will have to keep some of the values (represented by corresponding vertices) in the memory (for at least part of their lifetime).

The compiler should generate the code such that a live value will either reside in a register or in a memory location. Before the program overwrites a register which stores a still-live value, that value must be saved to a memory location.

This is called *spilling* the register, and the memory location to save the spilled value called its *spill location*.

Coloring by Simplification

For arbitrary graphs, coloring is an NP-complete problem. On the other hand, there exists a linear-time heuristic method (known since 19th century) which is based on *simplification* of the graph described as follows:

Until the graph is empty, find a vertex, a, whose degree is $< K$. Remove a from the graph and push it to the *coloring stack*. (Some people propose that the node with the lowest degree is removed.)

If such a vertex cannot be found before the graph becomes empty, then the simplification fails.

Otherwise, the graph can be colored in K colors by sequentially coloring the vertices popped off the coloring stack.

The reason the last step mentioned above works is because:

For any vertex m whose degree is $< K$, if the graph $G - \{m\}$ can be colored in K colors, then so can G.

Let us use an arbitrary graph to illustrate the simplification scheme, and see how many colors are needed to successfully color this graph under the scheme.

If the simplification scheme fails, it does not mean that a K-coloring does not exist.

An "optimistic" scheme (used by Briggs et al, 1989, 1994) continues to remove a vertex from the graph and push it into the coloring stack. (Which vertex to remove depends on how we define the *spilling priority*.)

During the coloring phase, we might still find it possible to color the graph with K colors.

Let us use another graph to illustrate this possibility.

If the optimistic scheme still fails, it still does not mean that a K-coloring does not exist. However, to simplify the solution, we will just assume that a K-coloring does not exist and we resort to spilling.

Any vertex that cannot be successfully colored is put in the *spilling list*. We continue to color the rest of the vertices (just to see if there exist more spilled vertices). When this is done, we need to modify the code by inserting memory load and store instructions for the spilled values.

The spilling code inserted above introduces more temporaries with short live ranges. We re-draw the interference graph and re-apply the coloring scheme. We iterate until we can color the modified graph with K colors.

Let us use one of the previous graph with $K = 4$ and $K = 3$ respectively, to show how the scheme works.

The Spilling Cost

When choosing a vertex in the interference graph to spill, the compiler needs to compare the spilling priority among the possible candidates. Such a priority depends on the spilling cost and the degree of the vertex in the graph. Commonly, the vertex with the lowest value of

$$\frac{spilling\ cost}{degree} \tag{1}$$

is considered the best candidate for spilling.

A vertex with a high degree in the interference graph is considered to be a good candidate for spilling because its spilling may yield a better chance for the remaining vertices to be colorable.

The spilling cost of a vertex is the performance penalty paid at run time due to the decision to spill the corresponding variable to the memory. Generally speaking, the more often a variable is referenced at run time, the higher its spilling cost.

Pre-colored nodes

Register-allocation schemes discussed above assume that all hardware registers can be used in the same way. However, as discussed in previous slides different registers can be assigned different roles in order to make function call/return faster:

- A number of registers may be designated to pass function arguments.
- One or two registers may be used to return function value(s).
- A subset of the registers may be designated as saved by the caller and the rest designated as saved by the

callee.

In order to use all registers as fully as possible, in order to reduce memory references, we want almost all registers be eligible for register allocation, (with a few such as **fp**, **sp** and return-address register excluded).

On the other hand, we need to retain the special roles of different registers. To do this, we add all registers (which participate in register allocation) to the interference graph and add appropriate edges to reflect the special constraints. These vertices are called *pre-colored*.

At the entry of the function, the registers which are used to pass arguments should be copied to the for-

mal arguments. These registers are dead after the copying is done, until some of these registers are used to return function result(s).

All callee-save registers are copied to new temporaries. These registers then remain dead until the new temporaries are copied back to them at the exit of the function. The live range of those new temporaries, therefore, expand nearly the whole function.

Any CALL instruction is assumed to define all caller-save registers. Therefore, a variable x which is not live across any CALL will not interfere with any call-save registers. However, if x is live across a CALL,

then it interferes with all caller-save registers. On the other hand, x will also interfere with all those new temporaries which are copied from callee-save registers, causing one of those temporaries to spill (because their spill priority is highest). This will cause x to be allocated to a callee-save register.

It is meaningless to select any pre-colored vertex to spill, so we assign the lowest spilling priority to pre-colored vertices.

We shall show how the interference graph is generated for the example in some of previous slides.

In the discussion above, register copying is introduced in many places. It is quite possible that some of them are unnecessary. A technique called $coalescing$ can be used to eliminate unnecessary copying. Since this technique is very specialized and there are other compiler techniques which can achieve the same or better effect, we will not discuss the coalescing technique in this course.