PURDUE UNIVERSITY

Department of Computer Sciences

Register Allocation by Graph Coloring

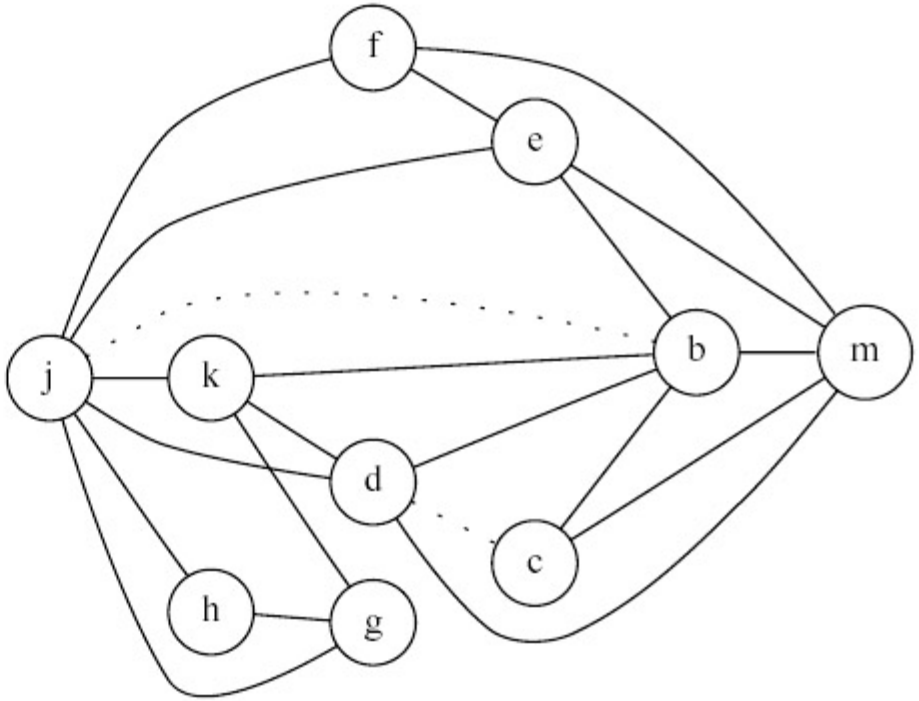
CS502

- We have used liveness information to construct an interference graph for register allocation.
- Optimal register allocation (allocating registers such that the number of memory references are minimized) is an intractable problem
- A common heuristic is based on coloring the interference graph
  - This is an NP-hard issue
- A fundamental difficulty is that register spilling is almost inevitable in practice
  - Which registers to spill depends on information that can only be “guessed”

- We present the simplest form of graph-coloring heuristic algorithms
- We illustrate how register spilling is handled and how it impacts on liveness of variables
- We briefly discusses the issue of “pre-colored” nodes in graph coloring.

```
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
```

Example:



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 Graph 11.1, 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 Chapter 6, 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 $\mathbf{fp}$, $\mathbf{sp}$ 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 $pre\text{-}colored$.

- 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 of Program 11.8.

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.