Reading Assignment

- Read David Wall's paper, "Global Register Allocation at Link Time."

Priority-Based Register Allocation

Alternatives to Chaitin-style register allocation are presented in:

- Hennessy and Chow, "The priority-based coloring approach to register allocation," ACM TOPLAS, October 1990.
- Larus and Hilfinger, "Register allocation in the SPUR Lisp compiler," SIGPLAN symposium on Compiler Construction, 1986.

These papers suggest two innovations:

1. Use of a *Priority Value* to choose nodes to color in an Interference Graph.

   A Priority measures
   (Spill cost)/(Size of Live Range)

   The idea is that small live ranges with a high spill cost are ideal candidates for register allocation.
   As the size of a live range grows, it becomes less attractive for register allocation (since it "ties up" a register for a larger portion of a program).

2. Live Range Splitting

   Rather than spill an entire live range that can't be colored, the live range is split into two or more smaller live ranges that may be colorable.

Large vs. Small Live Ranges

- A large live range has less spill code. Values are directly read from and written to a register.
  But, a large live range is harder to allocate, since it may conflict with many other register candidates.
- A small live range is easier to allocate since it competes with fewer register candidates.
  But, more spill code is needed to load and save register values across live ranges.
- In the limit a live range can shrink to a single definition or use of a register.
  But, then we really don't have an effective register allocation at all!

Terminology

In an Interference Graph:

- A node with fewer neighbors than colors is termed *unconstrained*. It is trivial to color.
- A node that is not unconstrained is termed *constrained*. It may need to be split or spilled.


```
PriorityRegAlloc(proc, regCount) {
   ig ← buildInterferenceGraph(proc)
   unconstrained ←
       { n ∈ nodes(ig) | neighborCount(n) < regCount }
   constrained ←
       { n ∈ nodes(ig) | neighborCount(n) ≥ regCount }

   while( constrained ≠ ϕ ) {
      for ( c ∈ constrained such that not colorable(c)
             and canSplit(c) ) {
         c1, c2 ← split(c)
         constrained ← constrained - {c}
         if ( neighborCount(c1) < regCount )
               unconstrained ← unconstrained U { c1}
         else  constrained ← constrained U {c1}
         if ( neighborCount(c2) < regCount )
               unconstrained ← unconstrained U { c2}
         else  constrained ← constrained U {c2}
         for ( d ∈ neighbors(c) such that
               d ∈ unconstrained and
                neighborCount(d) ≥ regCount ){
                 unconstrained ← unconstrained - {d}
               constrained ← constrained U {d}
      }     } // End of both for loops
```



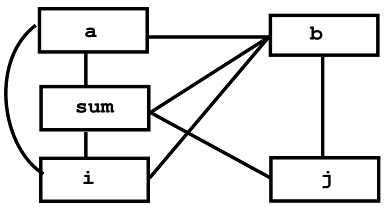
```
      /* At this point all nodes in constrained are
          colorable or can't be split */

      Select p ∈ constrained such that
                 priority(p) is maximized
       if ( colorable(p) )
             color(p)
       else  spill(p)
   } // End of While
   color all nodes ∈ unconstrained
}
```


How to Split a Constrained Node

- There are many possible partitions of a live range; too many to fully explore.
- Heuristics are used instead. One simple heuristic is:
  1. Remove the first basic block (or instruction) of the live range. Put it into a new live range, NR.
  2. Move successor blocks (or instructions) from the original live range into NR, as long as NR remains colorable.
  3. Single Basic Blocks (or instructions) that can't be colored are spilled.

Example

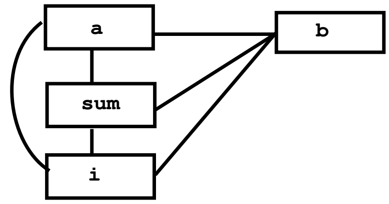
```
int sum(int a[], int b[]) {
  int sum = 0;
  for (int i=0; i<1000; i++)
     sum += a[i];
  for (int j=0; j<1000; j++)
     sum += b[j];
  return sum;
}
```



Assume we want a 3-coloring.

We first simplify the graph by removing unconstrained nodes (those with < 3 neighbors).

Node `j` is removed. We now have:

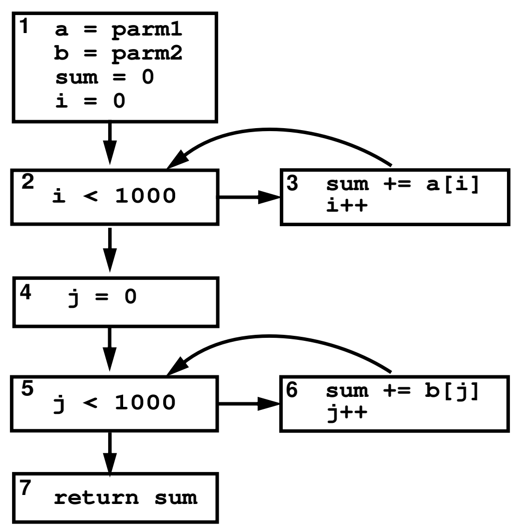


At this point, each node has 3 neighbors, so either spilling or splitting is necessary.

A spill really isn't attractive as each of the 4 register candidates is used within a loop, magnifying the costs of accessing memory.

Coloring by Priorities

We'll color constrained nodes by priority values, with preference given to large priority values.



	a	b	sum	i
Cost	11	11	42	41
Cost/Size	11/3	11/6	42/7	41/3

Variables `i`, `sum` and `a` are assigned colors `R1`, `R2` and `R3`.

Variable `b` can't be colored, so we will try to split it. `b`'s live range is blocks 1 to 6, with 1 as `b`'s entry point.

Blocks 1 to 3 can't be colored, so `b` is spilled in block 1. However, blocks 4 to 6 form a split live range that can be colored (using `R3`).

We will reload `b` into `R3` in block 4, and it will be register-allocated throughout the second loop. The added cost due to the split is minor—a store in block 1 and a reload in block 4.

Choice of Spill Heuristics

We have seen a number of heuristics used to choose the live ranges to be spilled (or colored).

These heuristics are typically chosen using one's intuition of what register candidates are most (or least) important. Then a heuristic is tested and "fine tuned" using a variety of test programs.

Recently, researchers have suggested using machine learning techniques to automatically determine effective heuristics.

In "Meta Optimization: Improving Compiler Heuristics with Machine Learning," Stephenson, Amarasinghe, et al, suggest using *genetic programming* techniques in which

priority functions (like choice of spill candidates) are mutated and allowed to "evolve."

Although the approach seems rather random and unfocused, it can be effective. Priority functions *better than* those used in real compilers have been reported, with research still ongoing.

Interprocedural Register Allocation

The goal of register allocation is to keep frequently used values in registers.

Ideally, we'd like to go to memory only to access values that may be aliased or pointed to.

For example, array elements and heap objects are routinely loaded from and stored to memory each time they are accessed.

With alias analysis, optimizations like Scalarization are possible.

```
for (i=0; i<1000; i++)
     for (j=0; j<1000; j++)
        a[i] += i * b[j];
```

is optimized to

```
for (i=0; i<1000; i++){
     int Ai = a[i];
     for (j=0; j<1000; j++)
        Ai += i * b[j];
     a[i] = Ai;
}
```


Attacking Call Overhead

- Even with good global register allocation calls are still a problem.
- In general, the caller and callee may use the same registers, requiring saves and restores across calls.
- Register windows help, but they are inflexible, forcing all subprograms to use the same number of registers.
- We'd prefer a register allocator that is sensitive to the calling structure of a program.

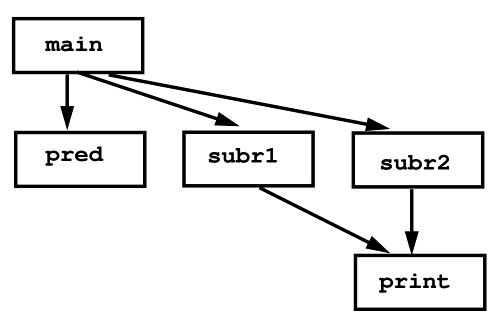
Call Graphs

A *Call Graph* represents the calling structure of a program.

- Nodes are subprograms (procedures and functions).
- Arcs represent calls between subprograms. An arc from A to B denotes that a call to B appears within A.
- For an indirect call (a function parameter or a function pointer) an arc is added to all potential callees.

Example

```
main() {
  if (pred(a,b))
       subr1(a)
  else subr2(b);}

bool pred(int a, int b){
   return a==b; }

subr1(int a) { print(a);}

subr2(int x) { print(2*x);}
```



Wall's Interprocedural Register Allocator

Operates in two phases:

1. Register candidates are identified at the subprogram level.
   Each candidate (a single variable *or* a set of non-interfering live ranges) is compiled as if it *won't* get a register.
   At link-time unnecessary loads and stores are edited away *if* the candidate *is* allocated a register.
2. At link-time, when all subprograms are known and available, register candidates are allocated registers.