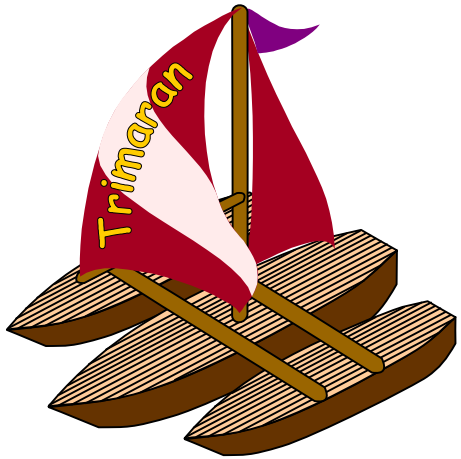


Case Study: A Region-Based Register Allocator

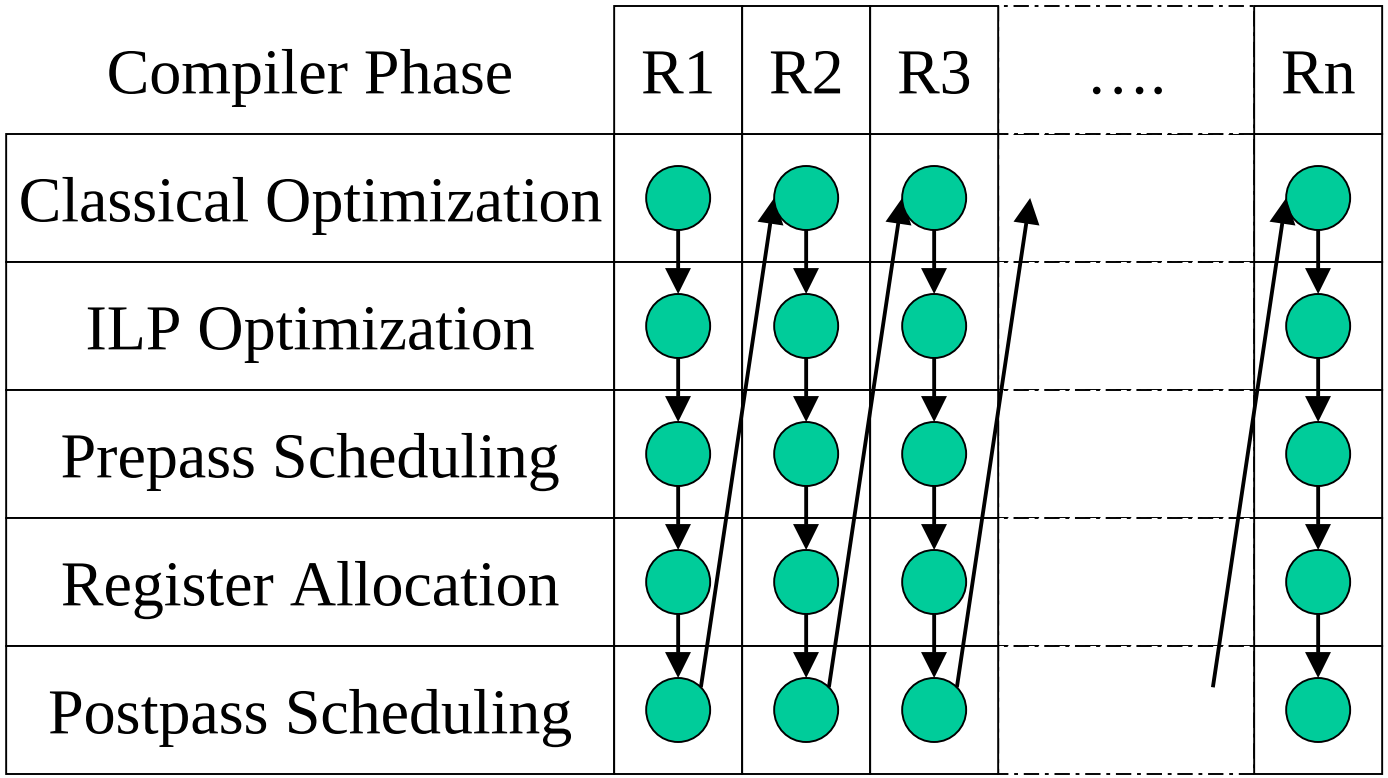
Overview

- The Elcor Module needed a register allocator that exploited the region-based program representation.
- A region-based register allocator was implemented.
  - Technical description to follow
- Register allocator is complete and will be part of the initial release of the system.
  - Design & implementation contributed by Kim, Gopinath, Kathail, Esfahany, and Palem.

Features of the Register Allocator

- Region Based
- Priority Based Coloring
  - Chow & Hennessy's approach
    - Frequency based priority function
- Finer Grained Live Ranges
- Region Reconciliation
- Handles Predicated Instructions

Phase Order



In our actual implementation, this is performed at the granularity of procedures.

Advantages

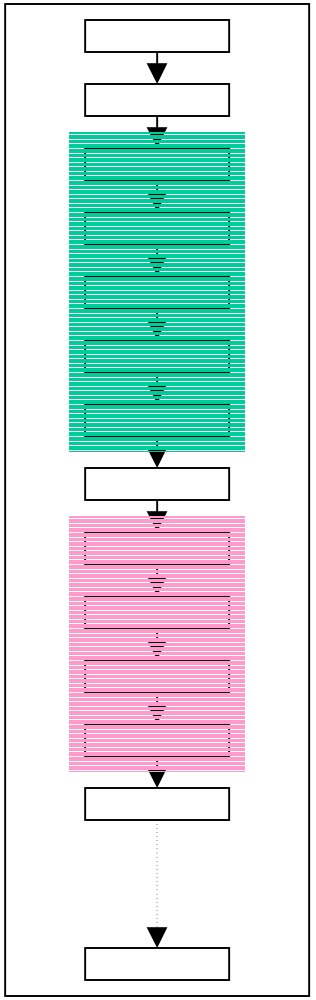
- Compiler is in complete control over the size and contents of the compilation unit
- The size of the compilation unit is typically smaller than functions, reducing the importance of the algorithmic complexity of the applied ILP transformations.
- The use of profile information to select regions allows the compiler to select compilation units that more accurately reflect the dynamic behavior of the program.
- Reference
  - “Region Based Compilation” Richard Hanks, Wen-mei Hwu and Bob Rau

Finer Grained Live Ranges

- We use operations as the units of live range construction
  - Chow & Hennessy used basic blocks
- Problems with Basic Block-based LR construction:
  - Imprecise LR construction
    - Problem is exacerbated by Superblocks and Hyperblocks
  - Code Predication is not visible in Basic Block

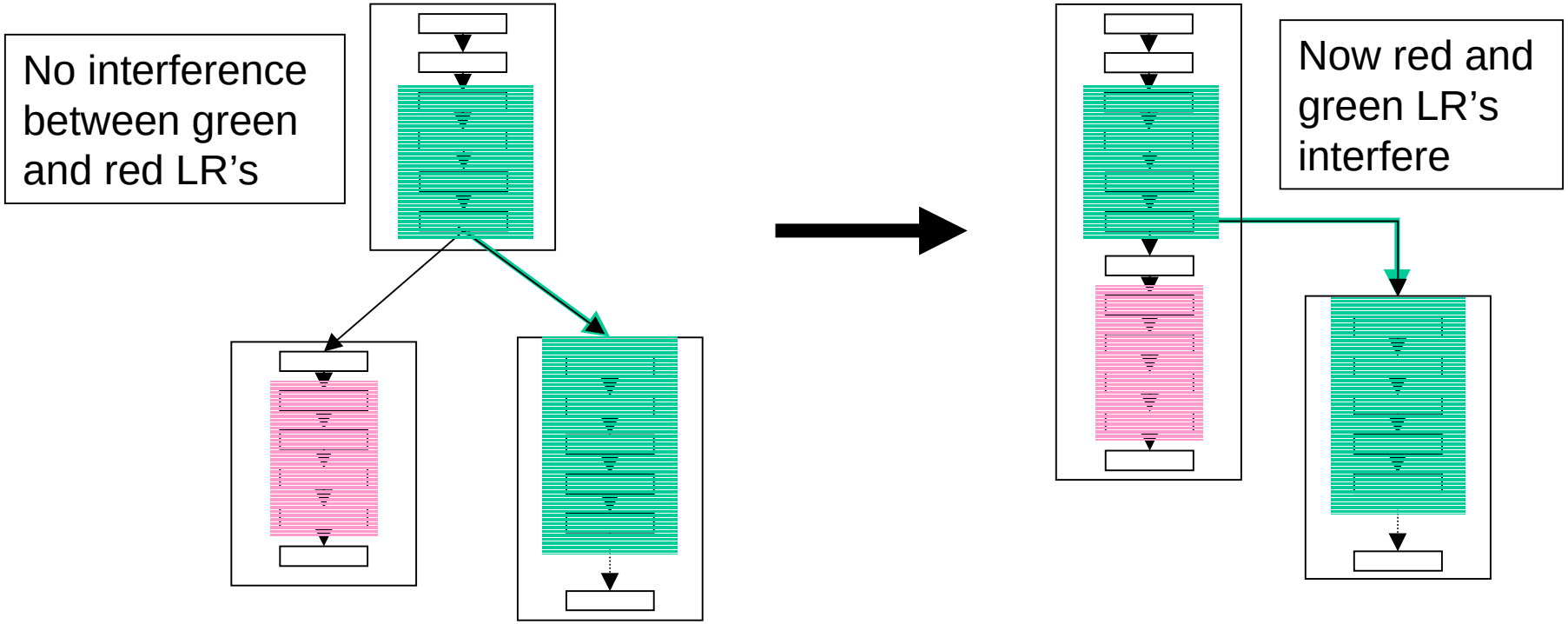
Basic Block Live Range Construction

- This basic block contains two non-interfering live ranges.
- They would be marked as interfering in an interference graph using basic block live range construction.
- They would be allocated to separate physical registers.



Basic Block LR Construction (cont)

- When basic blocks are coalesced to form superblocks, the problem becomes even worse.
- New false interferences are created.



Priority Based Coloring

- Live Ranges are colored (I.e. allocated to registers) in order of their priority.
- Their priority is computed based on:
  - the frequency of execution of their containing region
  - the cost/benefit of keeping them in a register.
- Register assignment incorporates a calling cost computation
  - I.e. assignment to caller-saved vs. callee-saved registers.
  - Lueh & Gross 1997

Priority Based Coloring (cont)

- Within a single block, the priority P(l) of a live range l for a virtual register v is computed by

$$P = S_l / N_l$$

where

S_l = loadsave*u_l + storesave*d_l

N_l = Size of live range (number of operations)

loadsave = cost of a load

storesave = cost of a store

u_l = number of uses of v

d_l = number of definitions of v

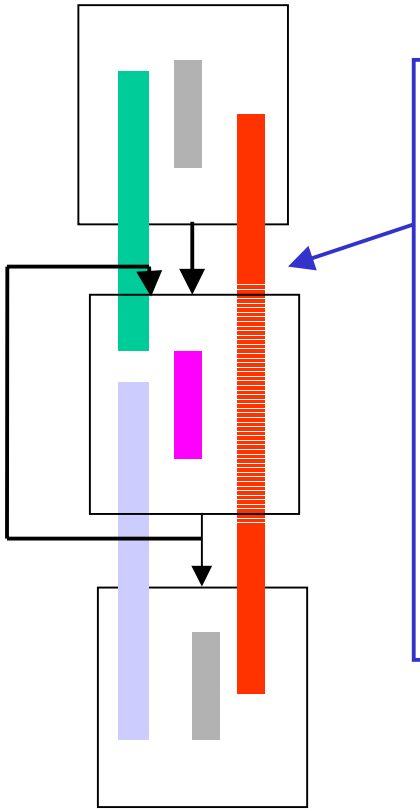
Priority Based Coloring (cont)

- For multi-block regions, the priority of a live range l, P(l), is based on the execution frequency F(b) of each block b in l and $P_b(l)$.

$$P(l) = \sum (P_b(l) * F(b))$$

Register Assignment

- Register assignment is done at the block level
  - with consideration of the use of caller-saved vs. callee-saved registers

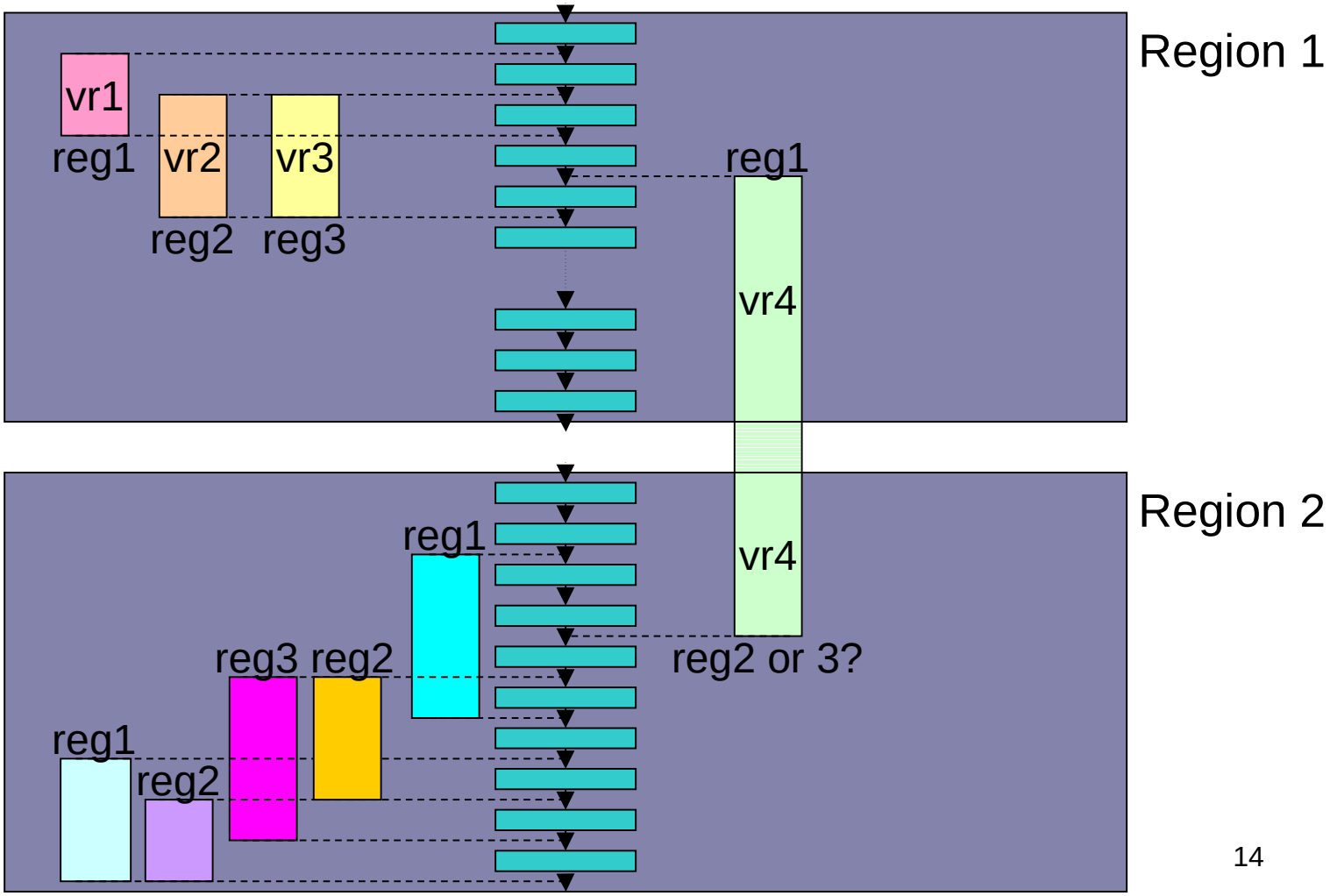


Notice that even if there are no uses of the virtual register in this intervening block,the assignment should account for the fact that it might be saved and restored due to procedure calls.

Reconciling Live Ranges

- In each region, the LR's are bound to physical registers based on their priorities.
  - The same LR in two different regions may be bound to different physical registers.
- Such an LR must be reconciled.
  - Need Patch Up code between each Region
  - may have to insert copy operations
- Reconciliation includes assigning to physical registers the delayed-bound LR's.
  - i.e. live ranges with no defs or uses in the block.

Reconciling Live Ranges



Predicated Code

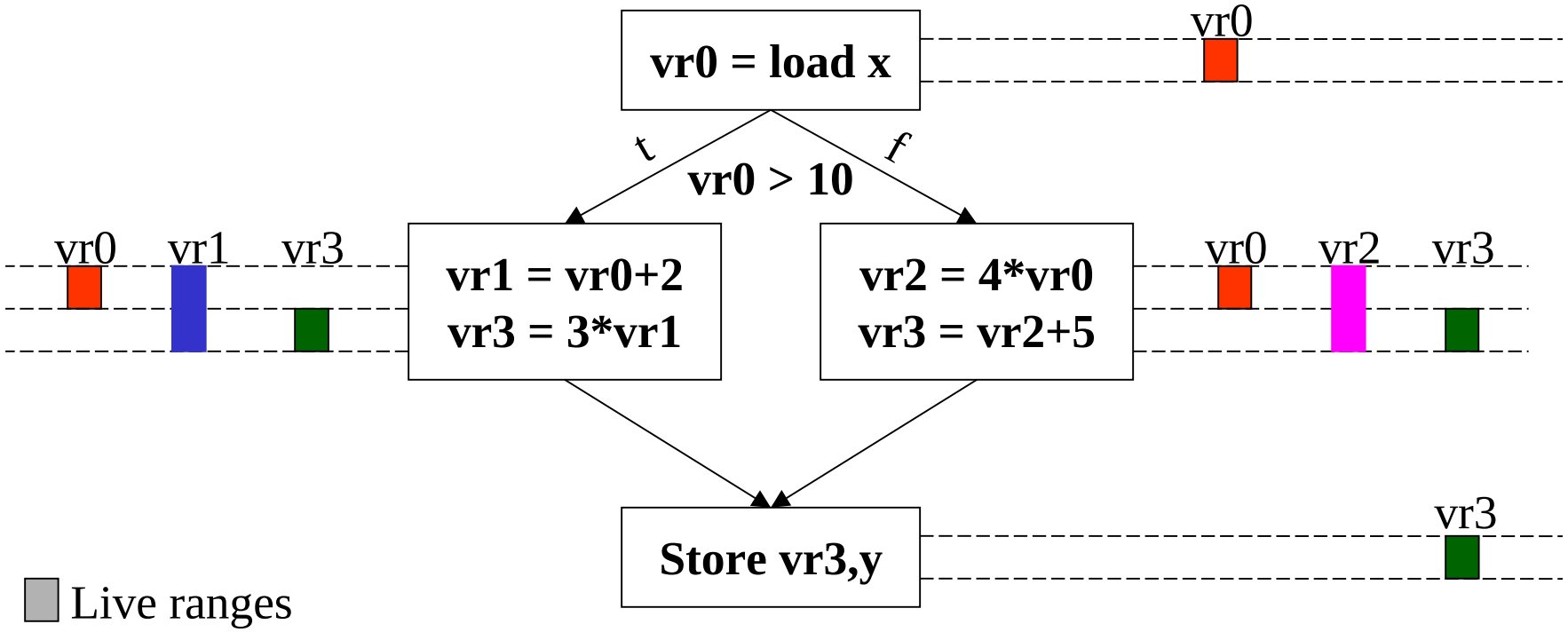
- ILP created by IF-conversion leads to an overly conservative IG in Predicated Code
  - might lead to unnecessary interference relationships
  - increase register pressure
- We used Elcor's PQS (Predicate Query System)
  - Used in register allocator to determine which groups of predicated operations are disjoint.
    - I.e. predicated upon complementary predicate registers.
  - avoids unnecessary interference in the interference graphs.

PQS: example

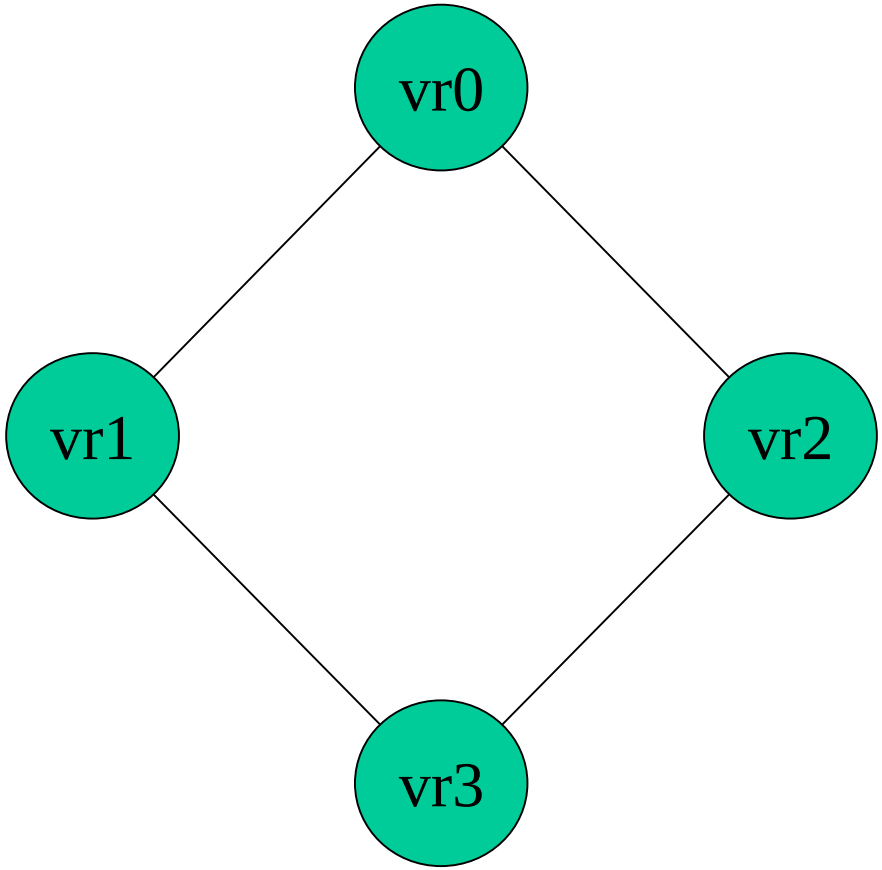
a) Conditional statement

if (x>10) y = 3*(x+2) ; else y = 4*x+5

b) Flow graph and live ranges for each basic block



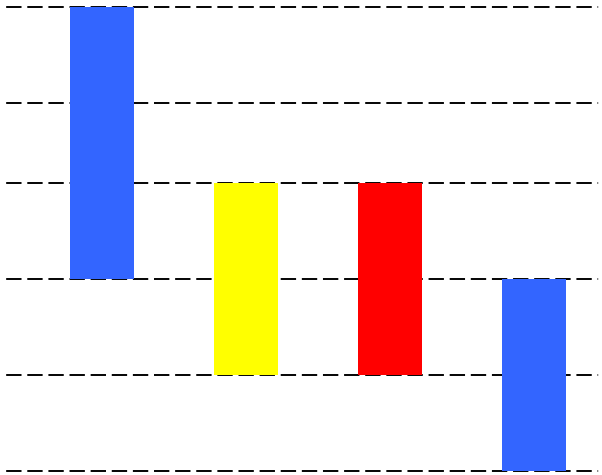
Interference Graph



Predicated block and live ranges

bb0-1-2-3

```
vr0 = load x
p1,p2 = (vr0>10)
vr1 =   vr0+2   if p1;   vr2 = 4*vr0  if p2
vr3 =   3*vr1  if p1;    vr3 = vr2+5  if p2
        store vr3 ,y
```



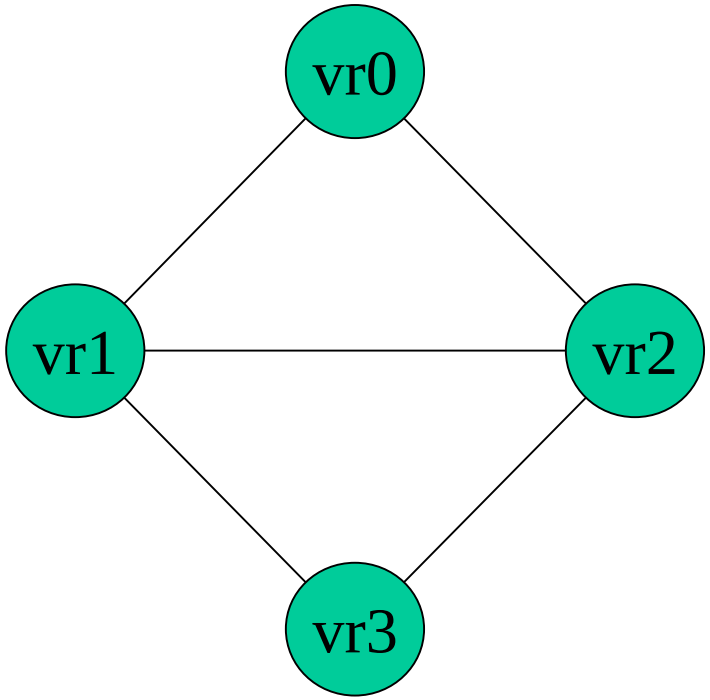

Unconditional live ranges

Live range under p1

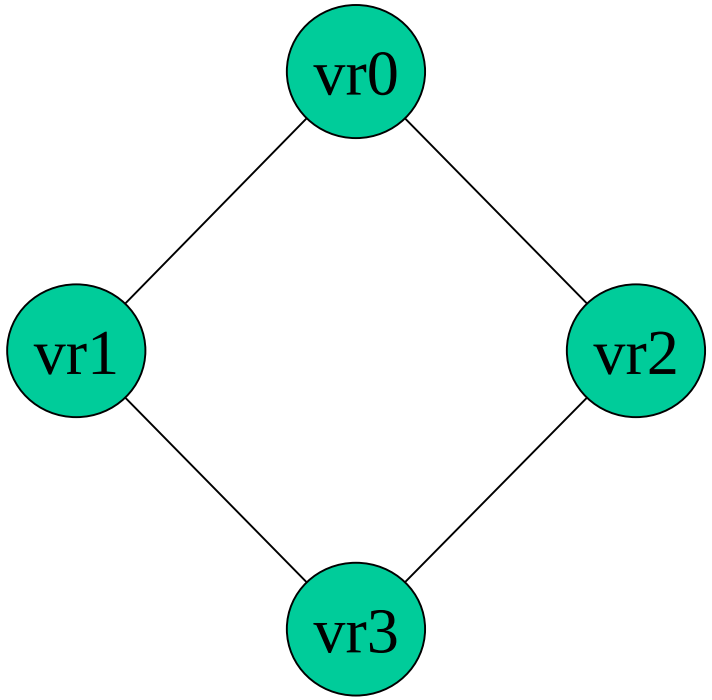

Live range under p2

Interference Graph

b) Without PQS



c) Using PQS



Register Allocator as a Case Study

- HPL-PD architectural features to be addressed
- Module Interface Issues
  - Input, Output, Module Placement
- Implementation Issues
  - Data structures, libraries, tools
  - How the module was inserted in the compilation path
- Conclusions about the infrastructure

HPL-PD-Specific Allocation Issues

- Predication must be handled correctly
  - support provided by Elcor's Predicate Query System (PQS)
- No other aspects of HPL-PD influenced the design or implementation of the register allocator.
  - VLIW instruction was considered as a sequence of operations.

Module Interface Issues

- Input/Output
  - Like all Elcor modules, the register allocator is an IR-to-IR transformer.
  - The input is a program graph in the internal IR
    - instructions have been scheduled
    - register operands are virtual.
  - Output is program graph in the internal IR
    - register operands are physical registers
    - spill operations have been added to the graph
      - but not yet scheduled

Module Interface (cont)

- Module Placement
  - Register Allocator comes after modulo scheduling and prepass acyclic scheduling and is followed by a postpass acyclic scheduler.
  - Adding a new phase of the compilation process is very easy
    - really just adding a procedure call in the main driver routine.
      - Call is to the main routine of the new module
      - Argument is generally the current procedure being compiled.

Main Elcor Driver Routine

```
Common_process_function(procedure *f)
{  . . .   //preprocessing, initialization
   . . .   //classic optimizations
           //(copy propagation, etc)
   . . .   //modulo scheduling
   . . .   //prepass acyclic scheduling
   el_solve_reg_alloc(f)  // register
                          //allocation
   . . . //postpass scheduling
   . . . //finalization
}
```

We just added this call

Implementation Issues

- Data structures used
  - Naturally, the region, operand, and edge classes of the IR were used.
    - In our case, only control edges
      - no register or memory dependence
  - Also a rich class library
    - Hash maps, bit vectors, lists, doubly-linked lists, hash sets, sorted lists, tuples
- Library Routines
  - We used an existing procedure for computing liveness.
  - Also the PQS routines.

Implementation Issues (cont)

- Debugging Support
  - Simulator for functional debugging.
  - It's easy to examine the Rebel text before and after a phase.
    - Modify common_process_function

```
. . .
ir_write(out,f)
el_solve_reg_alloc(f)
ir_write(out,f)
. . .
```

  - Class Browser
    - DaVinci tool for examining the Rebel IR in graphical form.

Conclusions

- Development Time
  - 2 person-months implementation time + 1 person-month testing and debugging
    - Once familiar with infrastructure (several more months)
    - Very short development time for a real register allocator in a serious compiler.

Conclusions (cont)

- Value of Infrastructure
  - Primary value lay in the framework
    - IR, module interfaces, existing modules
  - Tools were useful
    - PQS
    - Liveness Analysis

    but would not have been hard to write from scratch.
    - Easy for user community to enrich the existing infrastructure.