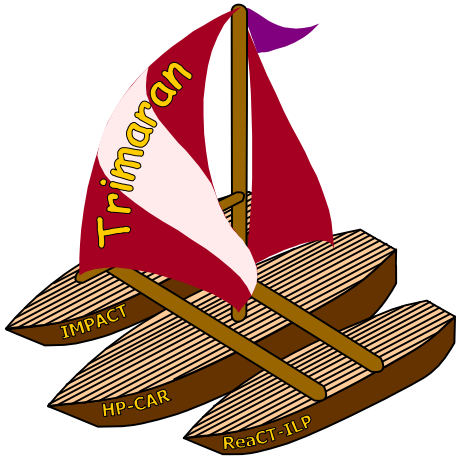

Case Study: A Region-Based Register Allocator

Overview

- 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

Register Allocator as a Case Study

- PlayDoh 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

PlayDoh-Specific Allocation Issues

- Predication must be handled correctly
  - support provided by Elcor's Predicate Query System (PQS)
- No other aspects of PlayDoh 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
    - VCG tool for examining the Rebel IR in graphical form.

Conclusions

- Development Time
  - 2 person-months implementation time + 2 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.