Register Allocation

Previously

- Graph Coloring Allocators
- Chaitin style
- Briggs enhancements

Today

- Linear-scan Register Allocation
  Traub, Holloway, and Smith

Motivation

Fast compilation times are becoming increasingly important

- dynamic compilation (JIT)
- whole program optimization
- production compilers

Register allocation is often a bottleneck

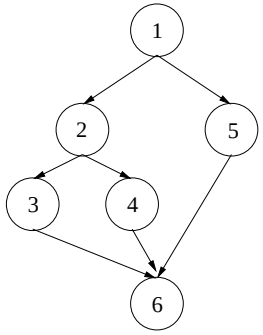
- Non-linear (sometimes not even a low polynomial)
- Inlining exacerbates its problems

Good register allocation is key to attaining peak processor performance.

- Quality of code vs.
- Speed of allocation

Linear Scan Register Allocation

- compute live information same way
- traverse the program in a linear order



(rPostOrder)

- process candidates as they appear (or in rank order)
- heuristically choose candidates to spill

$\Longrightarrow$ runs in linear time

Examples

- Digital's Gem compiler
- Jikes RVM
- Scale
- tcc dynamic code generation project
- Traub et al.

Computing Live Information

"Temporaries" - allocation candidates

- program variables
- compiler-generated temporaries

Temporary live ranges

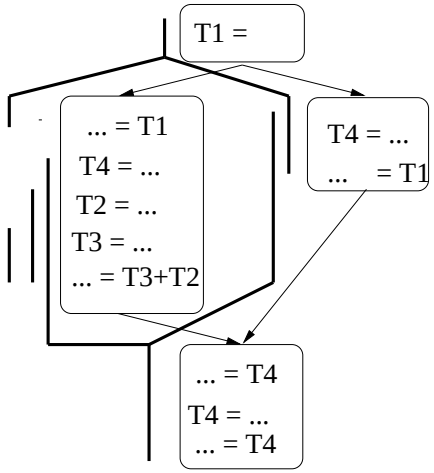
- interval in "linear order" in which the temporary is live
- the interval is at the granularity of instructions and basic blocks

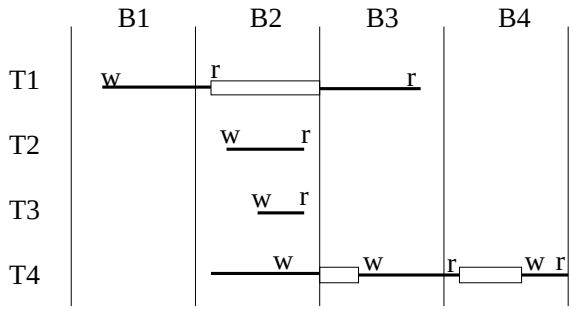
Live range holes

- interval during which no useful value is maintained
- e.g., interval between a use and the next definition

Compute live information in a single reverse pass.

Example Live Range Information



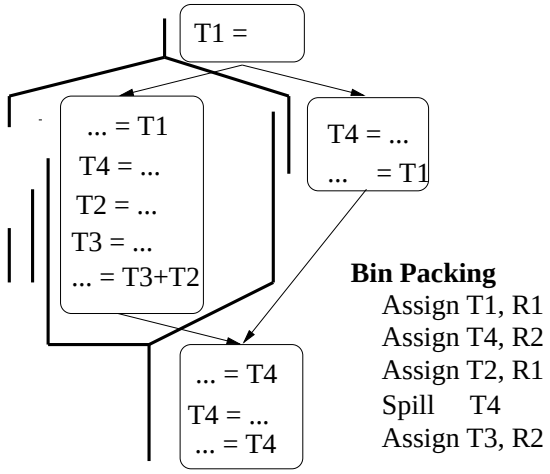


Bin Packing Register Allocation

1. **Compute live ranges**
2. **Treat registers as bins with one valid value at any point**
3. **View a register as containing a hole during free intervals:**
   - reflect arbitrary constraints on register usage
4. **Pack the same bin with two live ranges if**
   - live ranges do not interfere
   - one live range fits in a live range hole of another
5. **Live range is either in a register or memory**

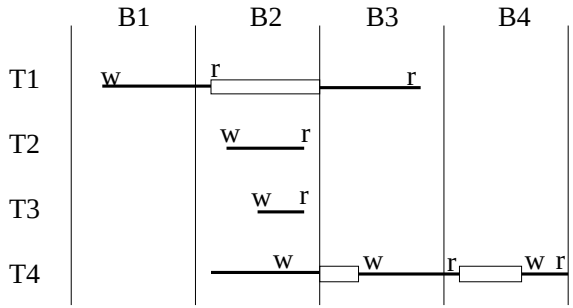
   Assign a temporary t a register r if:
   - t is not assigned, and r contains a hole large enough to hold t's live range.
   - choose r with the smallest such hole.
   - Otherwise, spill, choose lowest cost candidate
6. **One pass for allocation, and another to rewrite the code.**

Example Bin Packing



Bin Packing

Assign T1, R1
Assign T4, R2
Assign T2, R1
Spill T4
Assign T3, R2



Second-Chance Bin Packing

Give temporaries numerous chances to get a register (live range splitting)

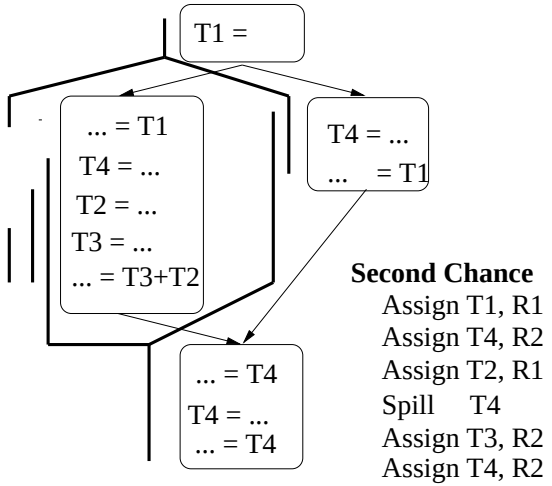
for each instruction in linear order
  for each temporary t
    if (t currently in register r)
      rewrite reference
    else // beginning of live range or spilled
      if ($\exists$ r with large enough hole)
        assign t to r
      else spill lowest cost candidate
  end for
end for
for each edge in control flow graph
  resolve conflicting location assumptions
end for

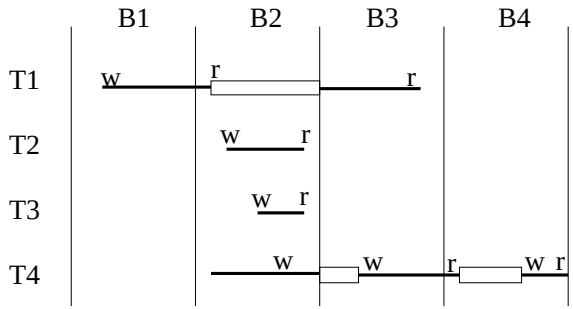
Resolution

- Linear order of code ignores dynamic control flow
- A temporary may reside in different locations across an edge (memory or register)
- Keep a map of locations of a temporary on entry/exit to basic blocks
- Post-process edges to reconcile conflicting assumptions
- Avoid stores when evicting a temporary
- Solve the following backward problem

  1. propagate information on use of consistency to avoid store
  2. guide insertion of resolution instruction across the CFG edge
  3. temporary in register at beginning of basic block
     - different registers coming in → register move
     - in memory & register coming in → register move
  4. temporary in memory at beginning of basic block
     - if in register, spill to memory, but only when necessary

Example Second-Chance Bin Packing





Experimental Evaluation

- Machine SUIF
- Implemented
  1. second-chance bin packing
  2. George & Appel coloring
  3. factors out common routines (e.g., live ranges) into shared libraries
- Platform: Alpha running Digital UNIX 4.0
- After register allocation:
  1. dead-code elimination
  2. peep-hole optimization
  3. copy propagation
- Measure
  1. dynamic instruction counts
  2. execution time
  3. register allocation time

Code Quality

Benchmark	ratio: binpacking / graph coloring	
	Instruction Count	Execution Time
alvinn	1.000	0.995
doduc	1.002	1.018
eqntott	1.000	1.003
espresso	1.013	1.060
fpppp	1.052	1.043
li	1.018	0.966
tomcatv	1.000	0.995
wave5	1.000	
compress	1.002	1.020
m88ksim	1.008	1.024
sort	1.035	1.082
wc	1.000	1.011

The instruction counts are from recent results, and the execution times are from the paper.

Spill Code

Percentage of total dynamic instructions due to spill code:

Benchmark	Second-chance binpacking	Graph coloring
alvinn	0%	0%
doduc	0.493%	0.518%
eqntott	0. 001%	0.000%
espresso	0. 901%	0.163%
fpppp	17. 111%	13.521%
li	0%	0%
tomcatv	0%	0%
wave5	0. 046%	0.032%
compress	0%	0%
m88ksim	0.033%	0.049%
sort	1.438%	0.996%
wc	0%	0%

Allocation Time

Procedure (code)	Average number of Register candidates	Average number of Interference edges	Allocation time seconds Graph coloring	Allocation time seconds binpack
cvrin.c (espresso)	245	1061	0.4	1.5
twldrv.f (fpppp)	6218	51796	8.8	3.7
fpppp.f (fpppp)	6697	116926	15.8	4.5
field() (wave5)	7611	86741	14.9	4.9

Summary

- Register allocation is NP-complete
- Heuristic solutions abstract away from register assignment
- Bin packing may achieve code close to coloring code
- For large graphs, compile time of bin packing better than coloring
- Many compilation scenarios prefer linear algorithms

Next Time

Dynamic Compilation

M. Arnold, S. Fink, D. Grove, M. Hind, P. Sweeney, A Survey of Adaptive Optimization in Virtual Machines, IEEE Computer, 92(2):449-466, February 2005.