

Linear Scan Register Allocation

ICS 211 - Compilers

by

Vivek Haldar

References

- “tcc: A system for fast, flexible and high-level dynamic code generation”, M. Poletto et al, SIGPLAN PLDI 1997
- “Quality and Speed in Linear Scan Register Allocation”, O. Traub, G. Holloway, M. D. Smith, SIGPLAN PLDI 1998
- “Linear Scan Register Allocation”, M. Poletto and V. Sarkar; TOPLAS 1999

Problems with Graph Coloring

Register Allocation

- Computationally expensive (but produces good code)
- Not suitable for situations in which code is required quickly
 - dynamic compilation
 - JIT compilers
- Code generation time vs. code quality tradeoff

Linear Scan - Definitions

- Arithmetic operations performed directly on temporaries, or *virtual registers*
- Instructions must be numbered according to some order (paper uses DFS order)
- ***Live Interval*** : $[i, j]$ is a live interval for variable v if v is not live at any instruction outside $[i, j]$
 - $[1, N]$ trivial live interval for every variable

Linear Scan - Preliminaries

- Given live variable info (obtained using data flow analysis), live intervals computed in one pass
- Interference between live intervals = overlap

Linear Scan Algorithm

- Live intervals sorted in order of increasing start point
- *Active* : list of live intervals that overlap current point *and* are in registers - sorted in order of increasing *end point*
- R - total number of registers

Linear Scan Algorithm

- For each new interval
 - scan *active* from beginning to end
 - remove expired intervals, make register available for allocation
- If $len(active) = R$, must **spill** from *active* or current interval
 - use heuristic: spill interval whose end is farthest from current point

Linear Scan Example

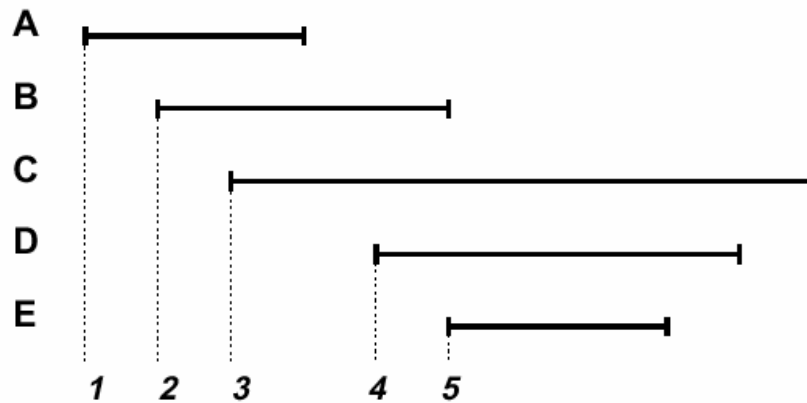


Fig. 2. An example set of live intervals. Letters on the left are variable names; the corresponding live intervals appear to the right. Numbers in italics refer to steps in the linear scan algorithm described in the text.

Assume $R = 2$.

2 - {A:r1, B:r2}

3 - spill C using heuristic

4 - A expires {B:r2, D:r1}

5 - B expires {E:r2, D:r1}

only 1 variable spilled

Note : if we spilled C, 2 variables would have been spilled

Linear Scan - Complexity

- V - number of live intervals
 - length of *active* bounded by R (# of registers)
 - if R constant, algorithm is $O(V)$
- But R can be large in some processor
 - time taken to insert into sorted list *active*
 - balanced tree - $O(V \log R)$
 - linear list - $O(VR)$
- Paper uses linear search

Variations and Heuristics

- Numbering of instructions
 - DFS or linear?
 - Produce roughly similar code
- Spilling heuristics
 - usage counts or “farthest from current”
 - similar results
 - favor “farthest from current” - simpler

Linear Scan - Compile Times

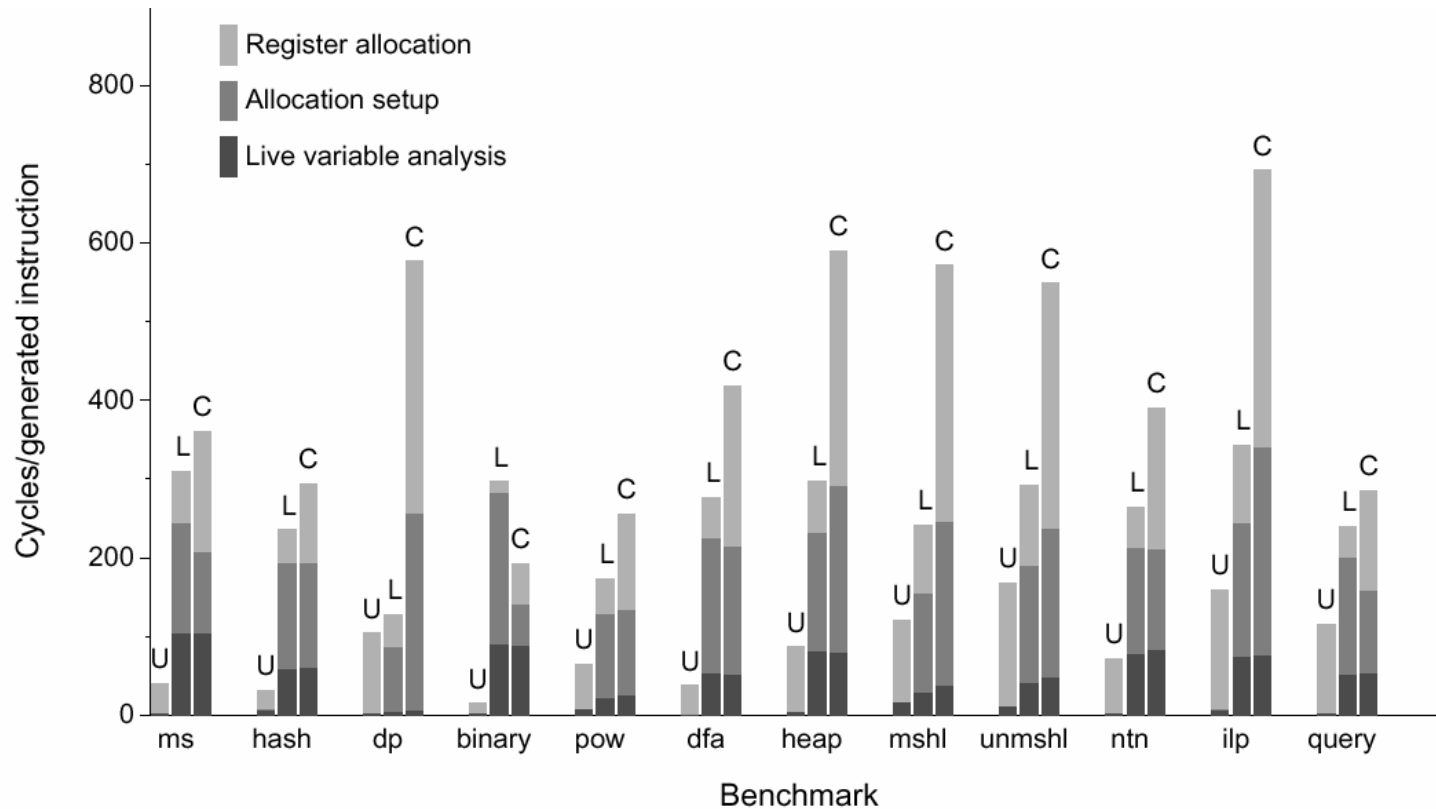


Fig. 3. Register allocation overhead for dynamic code ('C') kernels. U denotes a simple algorithm based on usage counts. L denotes linear scan. C denotes graph coloring.

Linear Scan - Run times

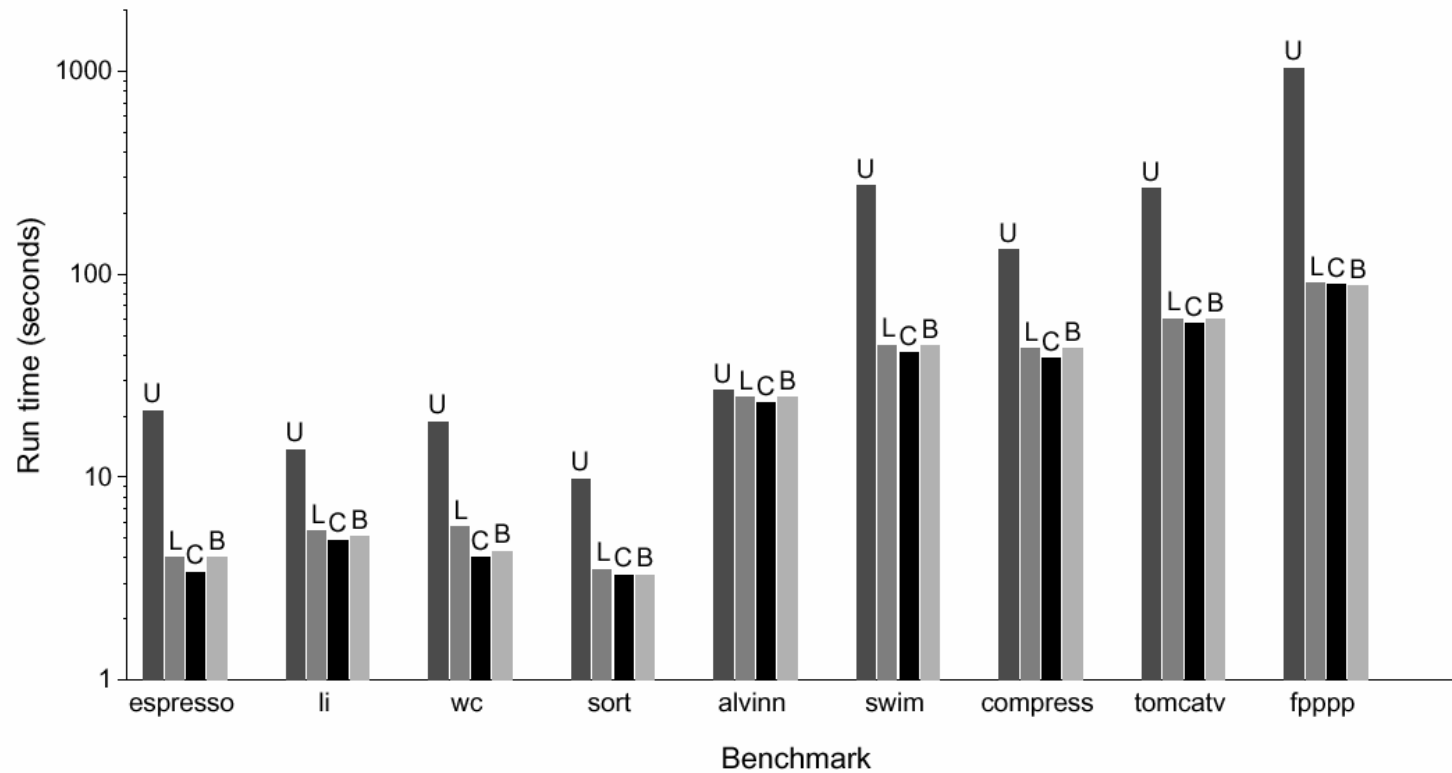


Fig. 8. Run times of static C benchmarks compiled with different register allocation algorithms. U, L, and C are as before. B denotes second-chance binpacking.

A Variation of Linear Scan Register Allocation

- In linear scan, liveness is a *lifetime interval*
- first definition to last use
- May have intervals when temporary holds
no useful value - *lifetime holes*
- *Second chance binpacking* makes use of
these holes

Lifetimes and holes

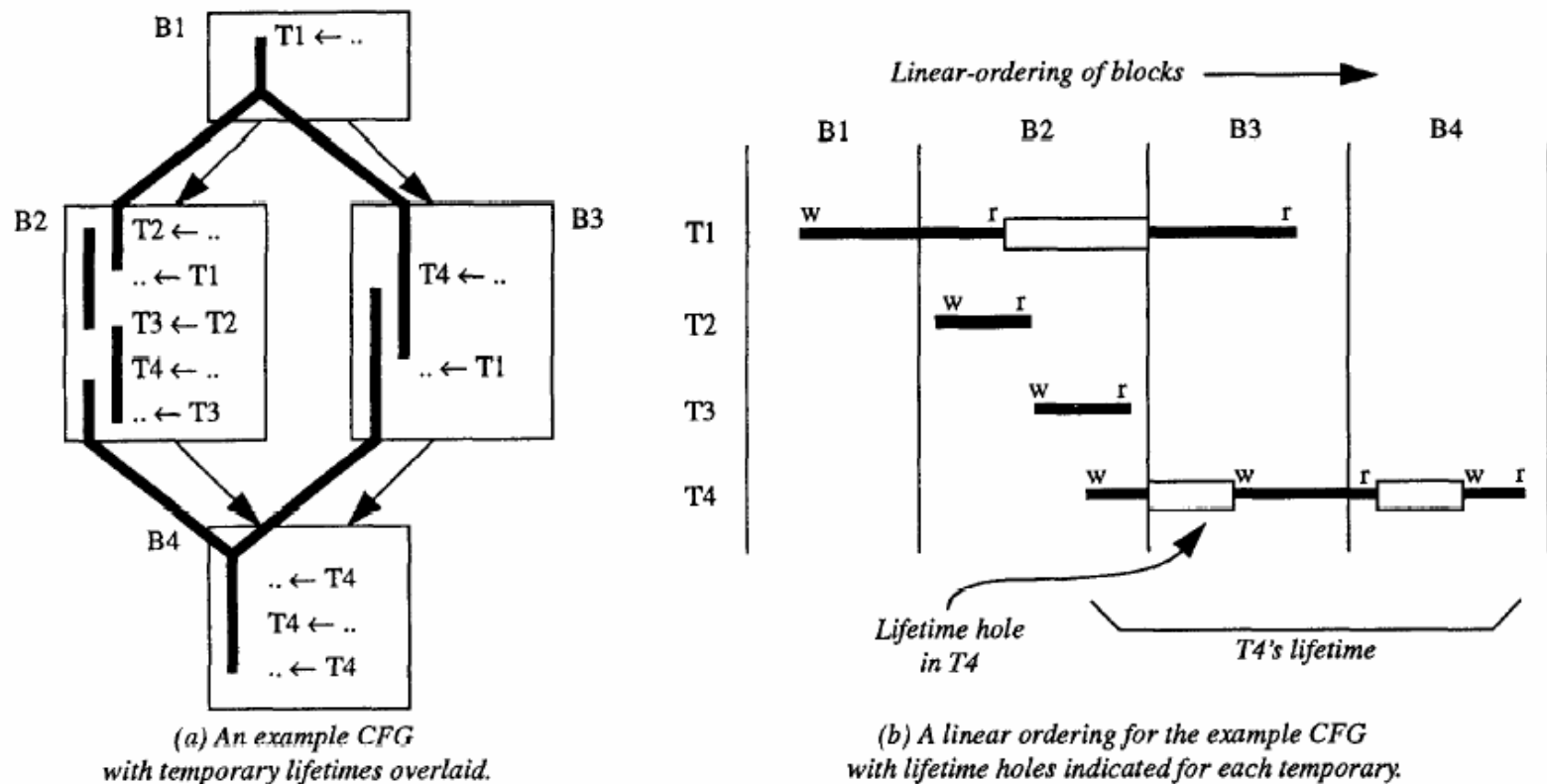


Figure 1. Example illustrating the concept of a linear ordering of a procedure's basic blocks, and the lifetimes and lifetime holes for the temporaries in this procedure. Notice that a block boundary can cause a hole to begin or end in the linear view of the program.

Code rewriting

- Linear Scan - allocate registers, then go back and rewrite operands with allocated register
- Second chance binpacking - allocation and rewriting in a *single scan*

Second chance binpacking

- If see t for first time, allocate register
 - may need to spill another temporary u from register r , using heuristic
 - Heuristic : distance to next reference, weighed by depth of loop it occurs in
 - all references to u already use r

Second chance binpacking

- See u again
 - read
 - find a register r (possibly spilling something else), and let u remain in r - optimistic about future references
 - write
 - allocate u to register r (possibly spilling)
 - postpone store of u into memory until u is spilled
 - if reach end of lifetime of u - don't need to store!

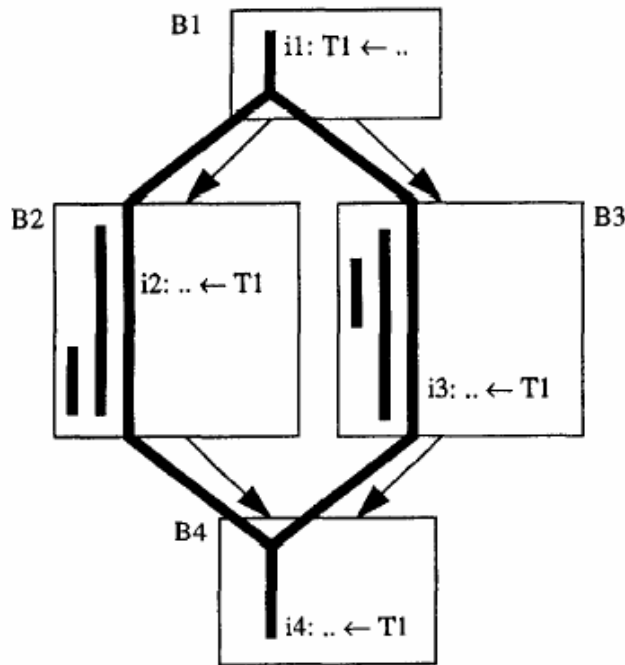
Second chance binpacking

- All following references to u are rewritten to use r
- ***Lazy storing*** : When spilling u , don't generate store if value in r is same as in memory
 - need to maintain *dirty bit*

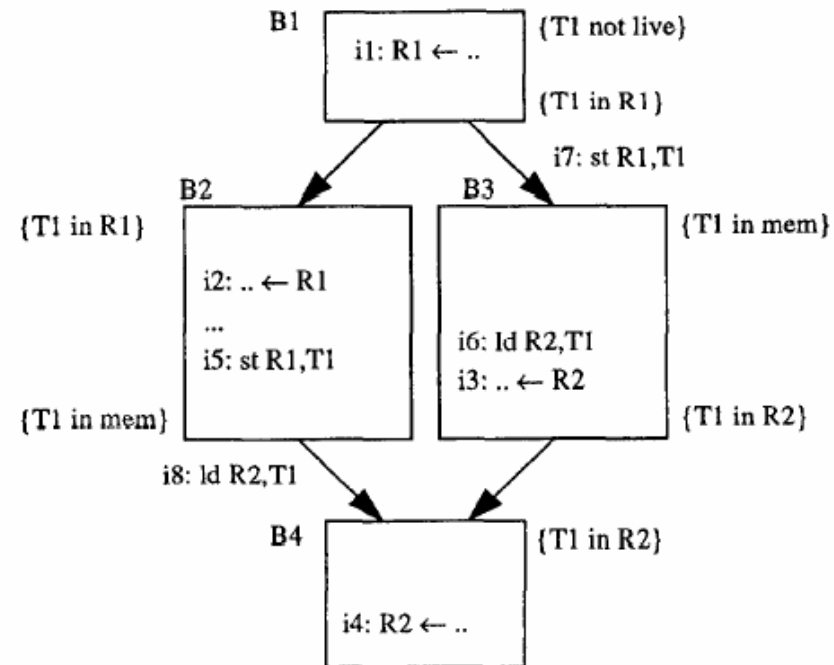
Second chance binpacking - Resolution

- Linear processing of allocation and rewrite incompletely models program control flow
- May have mismatches at CFG edges
 - may need to generate loads and stores along CFG edges
 - register - memory : store
 - memory - register : load
 - register i - register j : move

Second chance binpacking - Resolution



(a) An example CFG before allocation. The CFG contains 5 temporary lifetimes, but only T1's references are shown.



(b) The CFG after allocation. Only instructions associated with T1 are shown. The linear allocation order is B1-B2-B3-B4. The allocation assumptions for T1 before resolution are shown as sets at the top and bottom of each block.

Second chance binpacking - Resolution

- When doing lazy storing, when is a register consistent with memory? When do we need to generate spill stores?
 - Need to determine this along all paths
 - Need to do data flow analysis

Second chance binpacking - Complexity

- During data flow analysis, worst case of $O(N^2)$ bit vector operations
 - if size of bit vectors = number of temporaries, then $O(N^3)$
- Experience : terminates in 2 or 3 iterations - so $O(N)$ in practice
- Time spent in data flow : $< 1\%$ overall time

Second chance binpacking - Results

- Run times: no worse than approx. 10% more than graph coloring
- Compile times :
 - coloring faster for small programs
 - binpacking much faster for larger programs
 - coloring slows down drastically as complexity of interference graph increases

Summary

- Linear scan register allocation
 - much faster than graph coloring
 - but code quality comparable
- Second chance binpacking
 - code quality similar to linear scan
 - does more work