Linear Scan Register Allocation

Massimiliano Poletto (MIT)
and
Vivek Sarkar (IBM Watson)

Introduction

- *Register Allocation*: The problem of mapping an unbounded number of virtual registers to physical ones
- Good register allocation is necessary for performance
  - Several SPEC benchmarks benefit an order of magnitude from good allocation
  - Core memory (and even caches) are slow relative to registers
- Register allocation is expensive
  - Most algorithms are variations on Graph Coloring
  - Non-trivial algorithms require liveness analysis
  - Allocators can be quadratic in the number of live intervals

Motivation

- On-line compilers need generate code quickly
  - Just-In-Time compilation
  - Dynamic code generation in language extensions (‘C)
  - Interactive environments (IDEs, etc.)
- Sacrifice code speed for a quicker compile.
  - Find a **faster allocation algorithm**
  - Compare it to the best allocation algorithms

Definitions

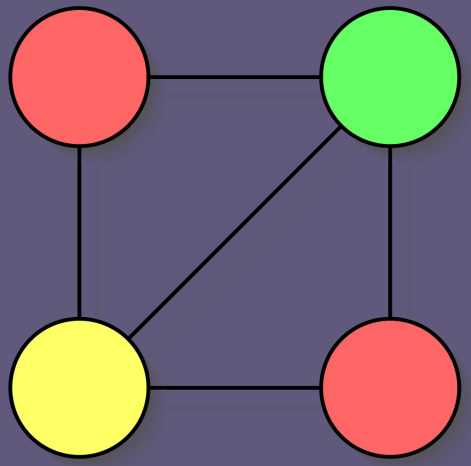
- *Live interval:* A sequence of instructions, outside of which a variable *v* is never live.

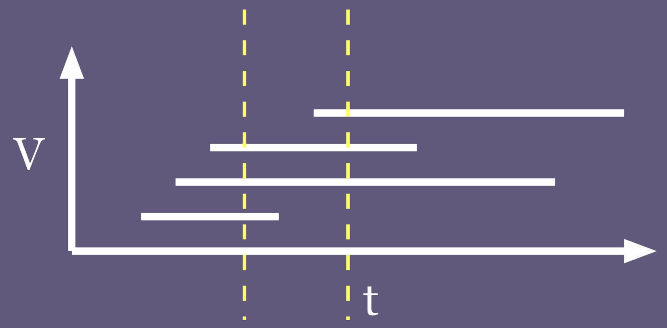
  (For this paper, intervals are assumed to be contiguous)
- *Spilling:* Variables are *spilled* when they are stored on the stack
- *Interference*: Two live ranges *interfere* if they are simultaneously live in a program.

Ye Olde Graph Coloring

- Model allocation as a graph coloring problem
- Nodes represent live ranges
- Edges represent interferences
- Colorings are safe allocations
- Order V^2 in live variables

- (See Chaitin82 on PLDI list)





Linear Scan Algorithm

- Compute live variable analysis
- Walk through intervals in order:
  - Throw away expired live intervals.
  - If there is contention, spill the interval that ends furthest in the future.
  - Allocate new interval to any free register

- Complexity: O(V *log* R) for V vars and R registers

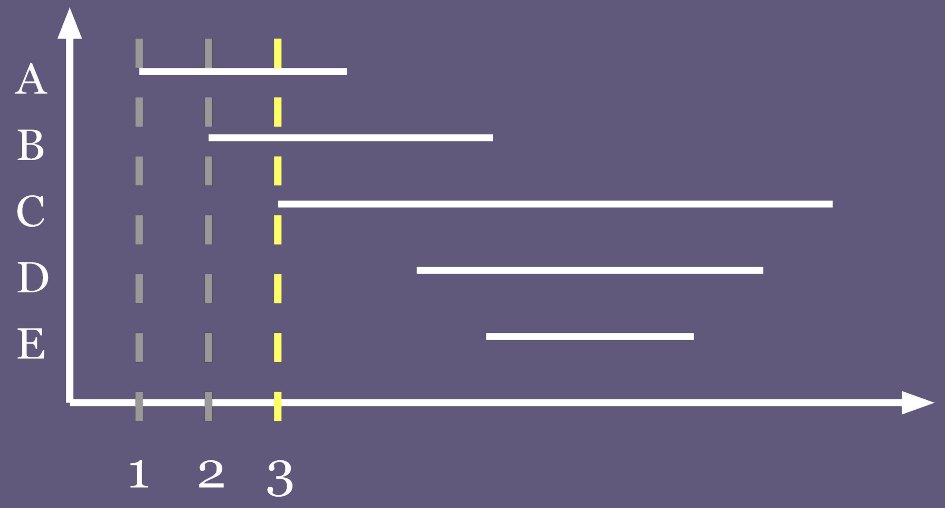
Example With Two Registers

- 1. Active = < A >

Example With Two Registers

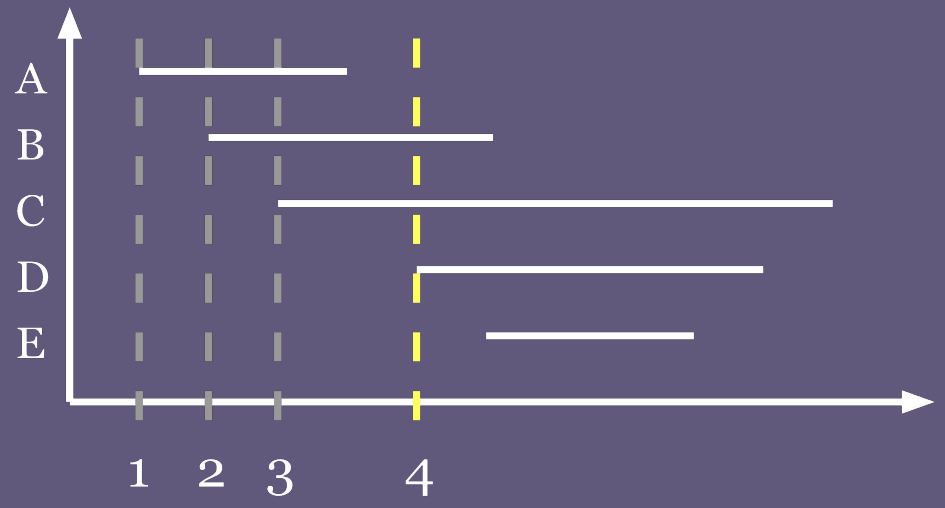
- 1. Active = < A >
- 2. Active = < A, B >

Example With Two Registers



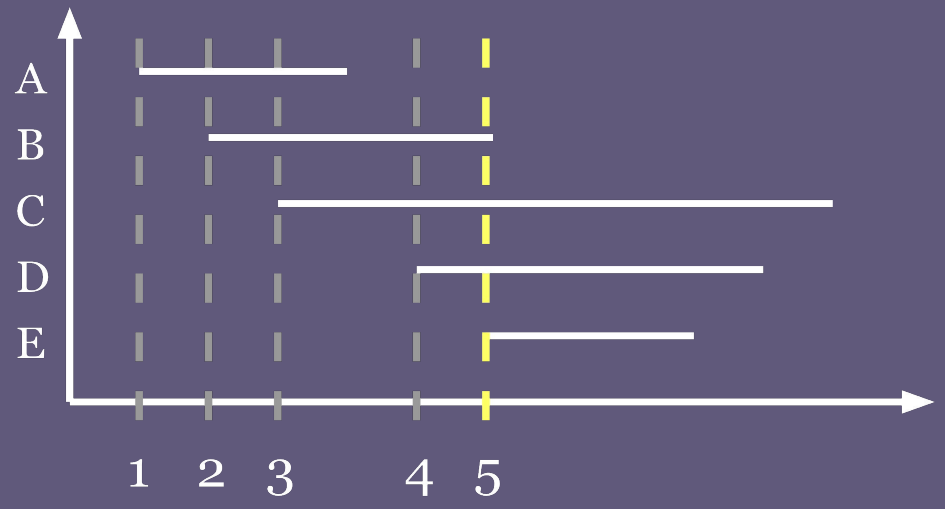
- 1. Active = < A >
- 2. Active = < A, B >
- 3. Active = < A, B > ; Spill = < C >

Example With Two Registers



- 1. Active = < A >
- 2. Active = < A, B >
- 3. Active = < A, B > ; Spill = < C >
- 4. Active = < D, B > ; Spill = < C >

Example With Two Registers

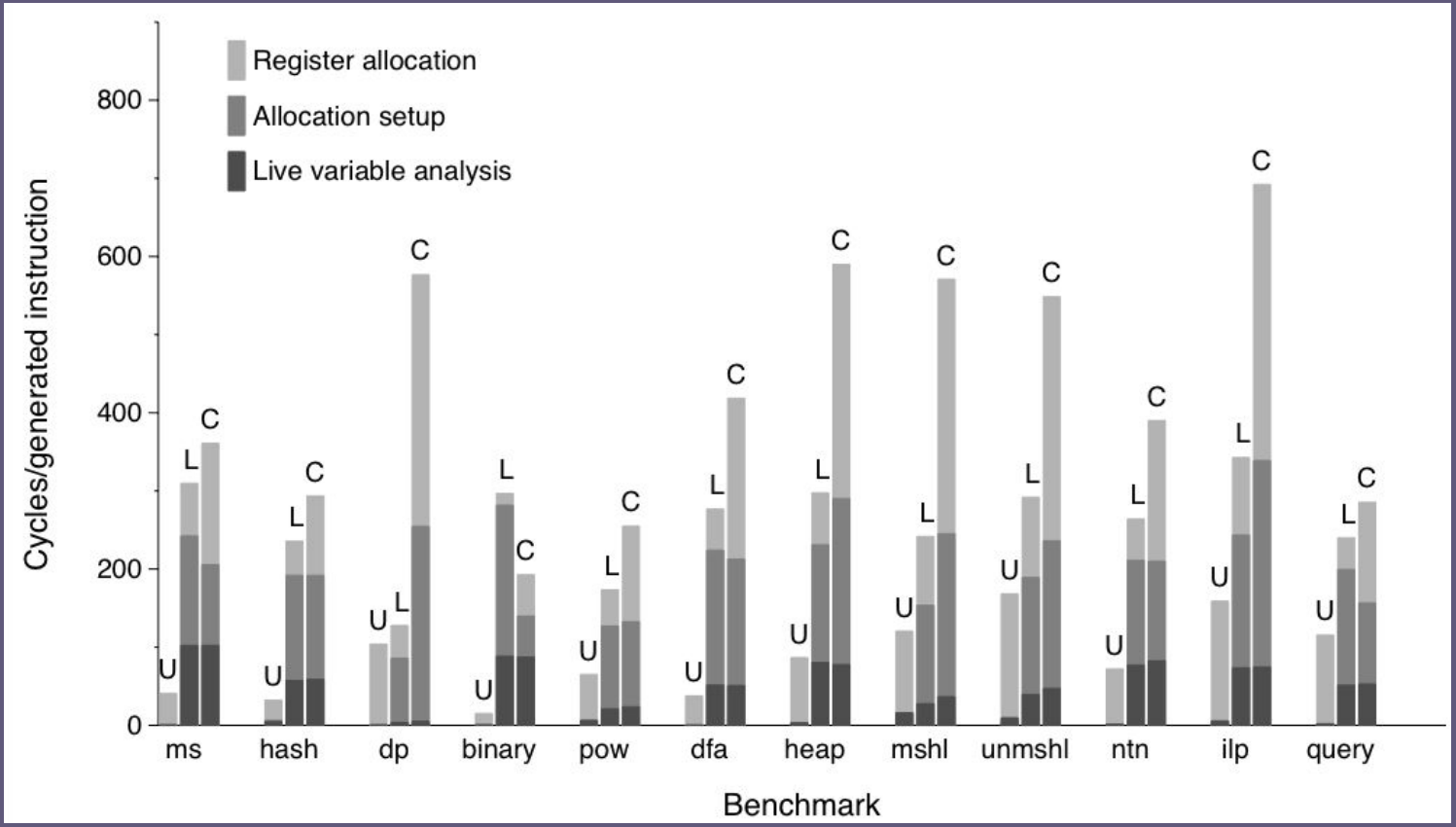


- 1. Active = < A >
- 2. Active = < A, B >
- 3. Active = < A, B > ; Spill = < C >
- 4. Active = < D, B > ; Spill = < C >
- 5. Active = < D, E > ; Spill = < C >

Evaluation Overview

- Evaluate both compile-time and run-time performance
- Two Implementations
  - ICODE dynamic ‘C compiler; (already had efficient allocators)
    - Benchmarks from the previously used ICODE suite (all small)
    - Compare against tuned graph-coloring and usage counts
    - Also evaluate a few pathological program examples
  - Machine SUIF
    - Selected benchmarks from SPEC92 and SPEC95
    - Compare against graph-coloring, usage counts, and second-chance binpacking
- Compare both metrics on both implementations

Compile-Time on ICODE 'C



- ***U***sage Counts, ***L***inear Scan, and Graph ***C***oloring shown
- Linear Scan allocation is always faster than Graph Coloring

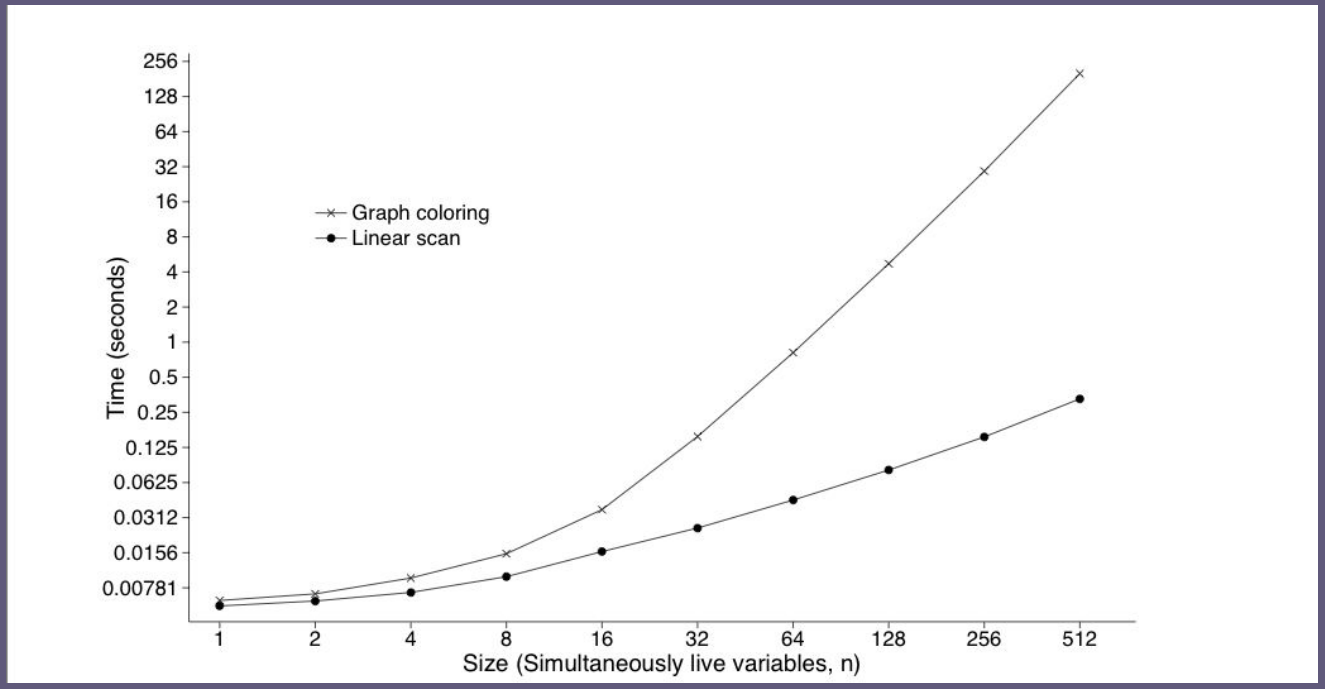
Compile-Time on SUIF

Table I. Allocation Times for Linear Scan and Binpacking

File (Benchmark)	Time in seconds		Ratio
	Linear scan	Binpacking	(Binpacking / linear scan)
swim.f (swim)	0.42	1.07	2.55
xllist.c (li)	0.31	0.60	1.94
xleval.c (li)	0.14	0.29	2.07
tomcatv.f (tomcatv)	0.19	0.48	2.53
compress.c (compress)	0.14	0.32	2.29
cvrin.c (espresso)	0.61	1.14	1.87
backprop.c (alvinn)	0.07	0.19	2.71
fpppp.f (fpppp)	3.35	4.26	1.27
twldrv.f (fpppp)	1.70	3.49	2.05

- Linear Scan allocation is around twice as fast than Binpacking
  - (Binpacking is known to be slower than Graph Coloring)

Pathological Cases



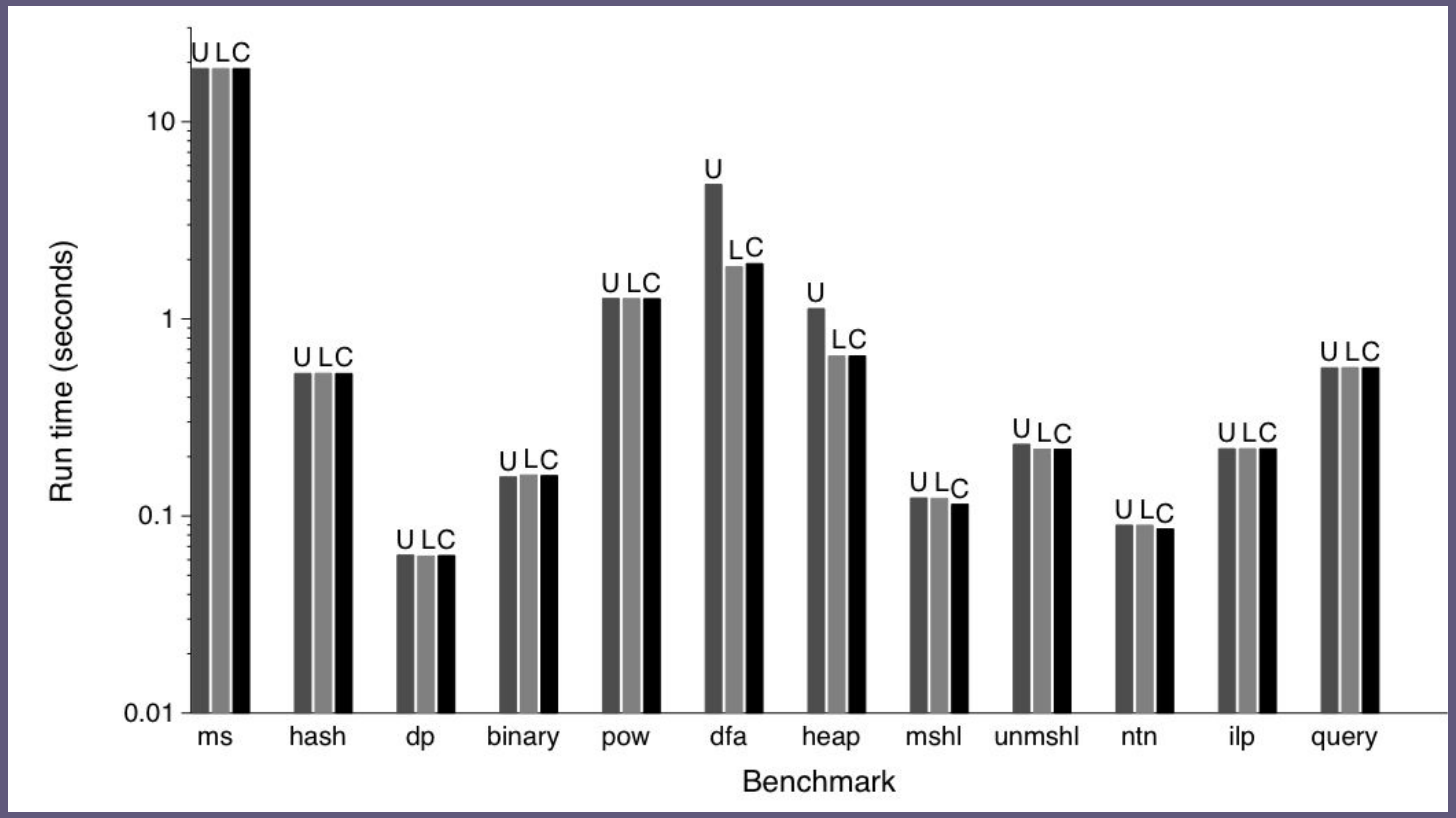
- *N* live variable ranges interfering over the entire program execution
- Other pathological cases omitted for brevity; see Figure 6.

Compile-Time Bottom Line

- Linear Scan
  - is faster than Binpacking and Graph Coloring
  - works in dynamic code generation (ICODE)
  - scales more gracefully than Graph Coloring

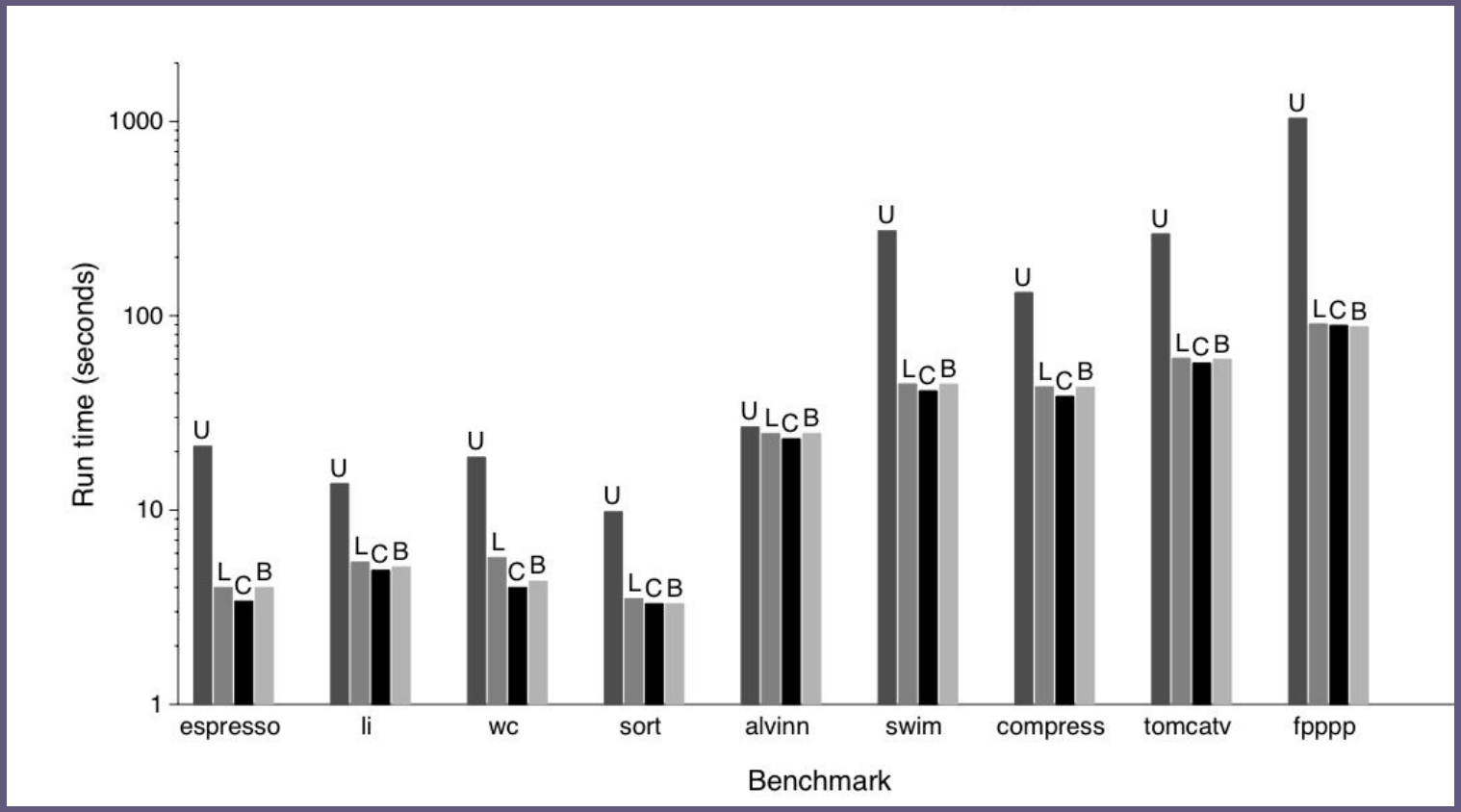
- … but does it generate good code?

Run-Time on ICODE 'C



- ***U***sage Counts, ***L***inear Scan, and Graph ***C***oloring shown
- Dynamic kernels do not have enough register pressure to illustrate differences

Run-Time on SUIF / SPEC



- ***U***sage Counts, ***L***inear Scan, Graph ***C***oloring and ***B***inpacking shown
- Linear Scan makes a fair performance trade-off (5% - 10% slower than G.C.)

Evaluation Summary

- Linear Scan
  - is faster than Binpacking and Graph Coloring
  - works in dynamic code generation (ICODE)
  - scales more gracefully than Graph Coloring
  - generates code within 5-10% of Graph Coloring
- Implementation alternatives evaluated in paper
  - Fast Live Variable Analysis
  - Spilling Hueristics

Conclusions

- Linear Scan is a faster alternative to Graph Coloring for register allocation
- Linear Scan generates faster code than similar algorithms (Binpacking, Usage Counts)
- Where can we go from here?
  - Reduce register interference with *live range splitting*
  - Use *register move coalescing* to free up extra registers

Questions?