Register Allocation

Kristopher Micinski

Syracuse University

CIS531, Fall 2025

Register Allocation

- These slides borrow heavily from “Essentials of Compilation” book by Siek (Ch3)

- In most of our IRs, we deal in terms of “variables,” or “pseudo-registers”
  - Want to keep things in variables for as long as possible

- Can’t keep everything in registers:
  - Not many registers (16 in x86-64, 32 in RISC-V)
  - Have to reserve some for special purposes (rbp, rax, etc.), e.g., in patch-instructions pass
  - Baseline is what we’re doing now: put things on stack

- Want to *optimally* assign things to registers

Example: Register Allocation

- Consider example on the right
  - I annotated it with live variables at each point
- At a minimum, we'd like to keep everything alive in a register
  - If it's not alive, then I don't need to keep it in a register....

```
// {}
a := (read)
// {a}
b := (read)
// {a,b}
c := a + b
// {b,c}
d := (read)
// {b,c,d}
e := b + c
// {c,e,d}
f := c + e
// {d,f}
g := f + f
// {d,f,g}
h := f + g
// {d,f}
i := d + f
// {}
```

- On the right, I have performed an example allocation
  - When a variable is never used again, it can be overwritten
  - Especially common in the case of temporary variables
    - Introduced as part of other passes…
- In this case, I put the last two variables on the stack
  - Some implementations would simply *eliminate* dead writes to optimize them…

```
// {}
%rax := (read)
// {a=%rax}
%rbx := (read)
// {a,b=%rbx}
%rax := %rax + %rbx
// {b,c=%rax}
%rcx := (read)
// {b,c,d=%rcx}
%rbx := %rbx + %rax
// {c,e=%rbx,d}
%rax := %rax + %rbx
// {d,f=%rax}
%rbx := %rax + %rax
// {d,f,g=%rbx}
%rbp[0] := %rax + %rbx
// {d,f}
%rbp[1*8] := %rcx + %rax
// {}
```

- What I want is an **oracle** that tells me…
  - For each program location…
    - Map: variables → physical registers / stack locations

- This is the goal of a **register allocator**
  - Very beautiful “textbook” solutions (graph coloring)
  - Less-beautiful practical algorithms (greedy)
  - Even-less-pretty engineering issues (SSA, multi-stage,..)

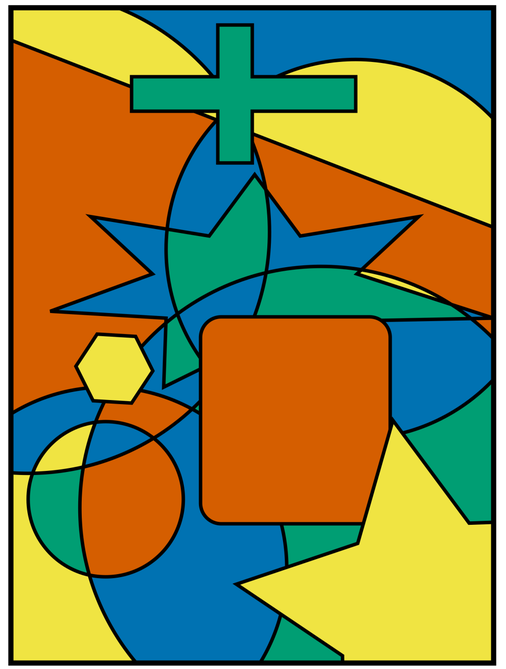
```
// {}
%rax := (read)
// {a=%rax}
%rbx := (read)
// {a,b=%rbx}
%rax := %rax + %rbx
// {b,c=%rax}
%rcx := (read)
// {b,c,d=%rcx}
%rbx := %rbx + %rax
// {c,e=%rbx,d}
%rax := %rax + %rbx
// {d,f=%rax}
%rbx := %rax + %rax
// {d,f,g=%rbx}
%rbp[0] := %rax + %rbx
// {d,f}
%rbp[1*8] := %rcx + %rax
// {}
```

Graph Coloring (famous NP-hard optimization problem)

- Early on, it was discovered that register allocation maps directly onto the famous *graph coloring* problem
  - Derive *interference graph* from live variables analysis

- For a **planar** graphs (one where you draw it as a 2D map)…
  - **Four Color Theorem**: any planar graph can be “colored” (no two regions with same color touch) with at most ***four*** distinct colors
  - Proved by Appel & Haken @ UIUC circa 1976: first novel proof done with significant computer assistance (controversial, thousands of small cases—how to know computer was correct?)

- For non-planar (general) graphs, **no** bound on # of needed colors



The Interference Graph

- Input to graph coloring: undirected graph that needs to be colored
- The key is to realize the equivalence of the two following statements…
  - In graph coloring: all n_0 and n_1 s.t., $n_0 \rightarrow n_1$ must have different colors
  - In register allocation terms: two variables live at same time must be in different registers
- Thus we do the following:
  - Run live variables analysis on the function (register allocation is per-function)
  - Build an interference graph based on the results
  - Consult the coloring to assign variables to registers as we emit code
    - Graph coloring yields minimal number of registers

Register Spilling

- Sometimes need more list than registers—in general, may need to ***spill***
- Register allocation algorithms account for this
  - Obviously, we want minimal spilling

- If we have N registers and need K > N colors to color the graph, then we interpret
  - N of the colors as the registers
  - K - N of the colors as the stack locations (minimized, because colors are minimized)

Building the Interference Graph, Naively…

- Siek discusses this very well in the book (Sec 3.3 in 153-page version)…
  - “The most obvious way to compute the interference graph is to look at the set of live variables between each statement in the program, and add an edge to the graph for every pair of variables in the same set. This approach is less than ideal for two reasons. First, it can be rather expensive because it takes $O(n^2)$ time to look at every pair in a set of n live variables. Second, there is a special case in which two variables that are live at the same time do not actually interfere with each other: when they both contain the same value because we have assigned one to the other.”

A better approach...

- Siek's book sketches a very nice, practical approach...

- Instead of considering all pairs, focus only on **writes**
  - Basic arithmetic operations, e.g., addq %src, %dst: %dst interferes with *all live variables*
  - Call instructions i add edge between all caller-saved registers and every live var v $\in$ Out(i)
  - If the instruction is a movq %src, %dst, add an edge between %dst and every live variable in the Out set of i, unless that variable is %src or %dst (can't add self edges, ask yourself why)
  - Other instructions follow the same pattern: add edges between %dst and each live variable

- Why does this work?

1	`(block ()`	$\{\}$
2	`  (movq (int 1) (var v))`	$\{v\}$
3	`  (movq (int 46) (var w))`	$\{v, w\}$
4	`  (movq (var v) (var x))`	$\{w, x\}$
5	`  (addq (int 7) (var x))`	$\{w, x\}$
6	`  (movq (var x) (var y))`	$\{w, x, y\}$
7	`  (addq (int 4) (var y))`	$\{w, x, y\}$
8	`  (movq (var x) (var z))`	$\{w, y, z\}$
9	`  (addq (var w) (var z))`	$\{y, z\}$
10	`  (movq (var y) (var t.1))`	$\{z, t.1\}$
11	`  (negq (var t.1))`	$\{z, t.1\}$
12	`  (movq (var z) (reg rax))`	$\{t.1\}$
13	`  (addq (var t.1) (reg rax))`	$\{\}$
14	`  (jmp conclusion))`	$\{\}$

Figure 3.2: An example block annotated with live-after sets

Line 2: no interference,
Line 3: w interferes with v,
Line 4: x interferes with w,
Line 5: x interferes with w,
Line 6: y interferes with w,
Line 7: y interferes with w and x,
Line 8: z interferes with w and y,
Line 9: z interferes with y,
Line 10: $t.1$ interferes with z,
Line 11: $t.1$ interferes with z,
Line 12: no interference,
Line 13: no interference.
Line 14: no interference.

The resulting interference graph is shown in Figure 3.3.

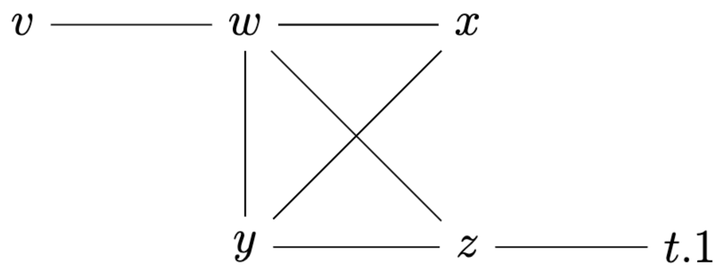


Figure 3.3: The interference graph of the example program.

Graph Coloring Algorithms…

- Graph coloring is NP-hard, and thus no *good* algorithms exist
  - Best we can hope for is reasonable approximation algorithms

- In practice, register coloring not easy—serious impediment to real-time compilation
  - Time/space trade-off: doing just a bit better might take a lot longer
  - But making a **bad** decision (particularly about what / when to spill) can be *very* bad

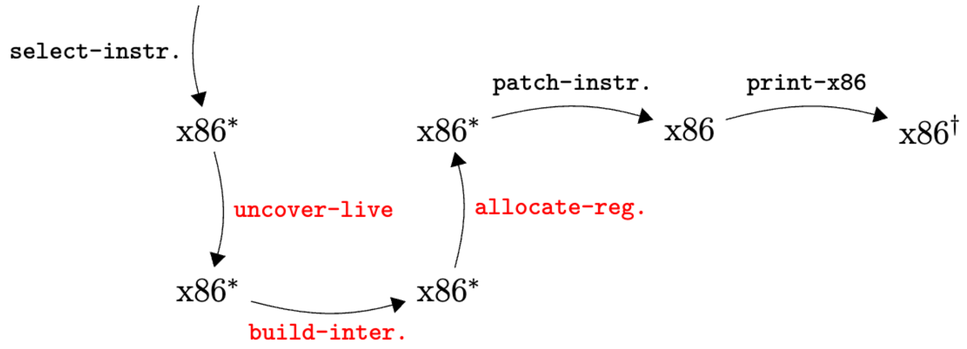
- Siek's book shows a relatively simple greedy algorithm called D-SATUR

D-SATUR Algorithm

- Colors are ordered: first N are available physical registers, N+1 represent the stack.

- While the graph contains an uncolored vertex
  - Pick v, the vertex with the highest ***saturation***
    - The virtual register whose neighbors already use the most distinct physical registers
    - Break times randomly / etc.
  - Find the lowest color c distinct from all the colors of neighbor nodes
  - Color v with the color c

- The outcome of the coloring is an assignment of each virtual register to all of the available physical registers, along with (when needed) -8(%rbp), -16(%rbp), ...
- We coded things up to use ***allocate-homes*** in our projects
  - You can replace it with these three passes
    - Do these really need to be three passes? Or can they just be one?
    - Why do live variables only on x86? Why not C2? Or R3? Or all of them?
      - Answer: only need live variables for x86, but might need to do analysis if you optimize those other IRs too...



Chaitin's Algorithm

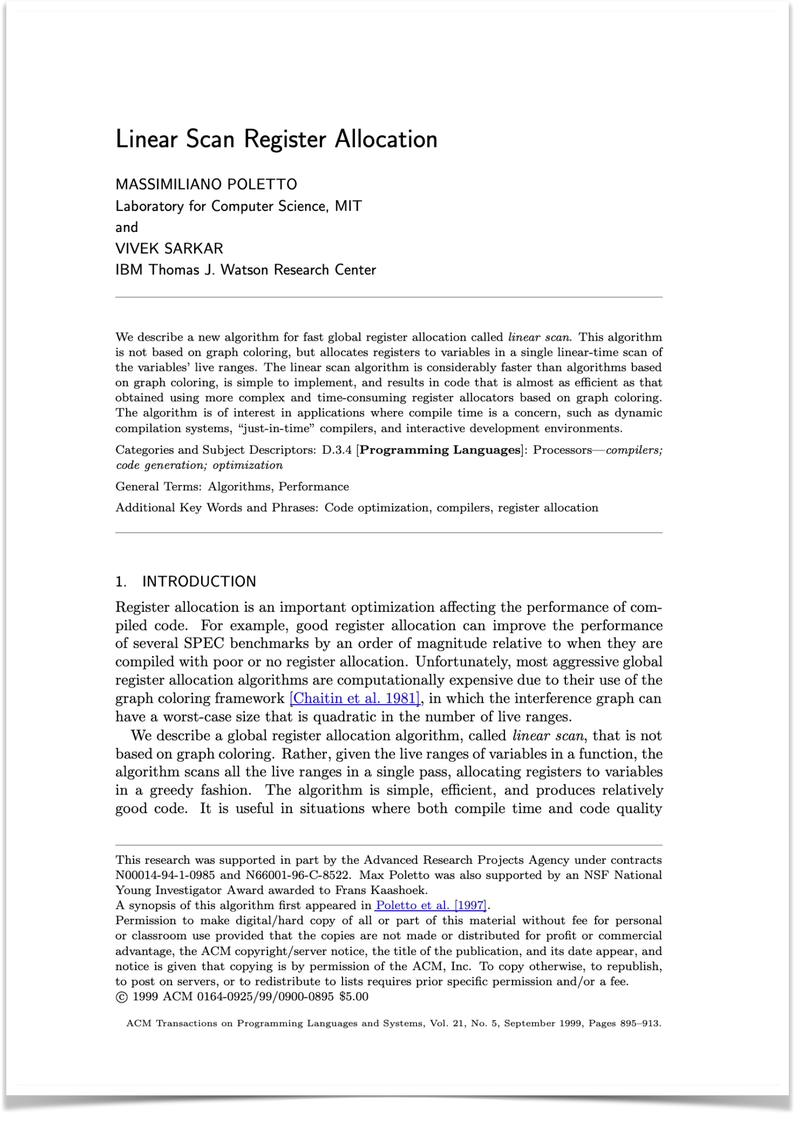
- Gregory Chaitin's register allocation algorithm: first to consider properties of usages / definitions to identify a ***spill metric***: not all spills are equal! Analyze how variable used

> These cost estimates are made as follows. The cost of spilling a node is defined to be the increase in execution time if it is spilled, which is approximately equal to the number of definition points plus the number of uses of that computation, where each definition and use is weighted by its estimated execution frequency. The cost estimates also take into account the fact that some computations can be redone instead of spilling and reloading them, and that if the source or target of a register copy operation is spilled then the copy operation is no longer necessary. In fact spilling a computation that can be recomputed and which is used as the source of a register copy operation can have negative cost!

Linear Scan Register Allocation

- Very, very simple idea: O(n) time!
- Desirable for real-time system, JIT compilers, etc.
- V8 and similar JavaScript runtimes typically use linear scan
  - Simple to implement: no O(n^2) performance as typical in coloring-based algorithms
  - O(n) in both theory and practice!

- Idea is this: order the “live ranges” of variables
  - Then traverse them in order, assigning the next available color to the range
  - When a variable’s live range ends, make it available again
    - It will be reassigned next



Linear Scan Register Allocation

MASSIMILIANO POLETTO
Laboratory for Computer Science, MIT
and
VIVEK SARKAR
IBM Thomas J. Watson Research Center

We describe a new algorithm for fast global register allocation called *linear scan*. This algorithm is not based on graph coloring, but allocates registers to variables in a single linear-time scan of the variables' live ranges. The linear scan algorithm is considerably faster than algorithms based on graph coloring, is simple to implement, and results in code that is almost as efficient as that obtained using more complex and time-consuming register allocators based on graph coloring. The algorithm is of interest in applications where compile time is a concern, such as dynamic compilation systems, "just-in-time" compilers, and interactive development environments.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers; code generation; optimization*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Code optimization, compilers, register allocation

1. INTRODUCTION

Register allocation is an important optimization affecting the performance of compiled code. For example, good register allocation can improve the performance of several SPEC benchmarks by an order of magnitude relative to when they are compiled with poor or no register allocation. Unfortunately, most aggressive global register allocation algorithms are computationally expensive due to their use of the graph coloring framework [Chaitin et al. 1981], in which the interference graph can have a worst-case size that is quadratic in the number of live ranges.

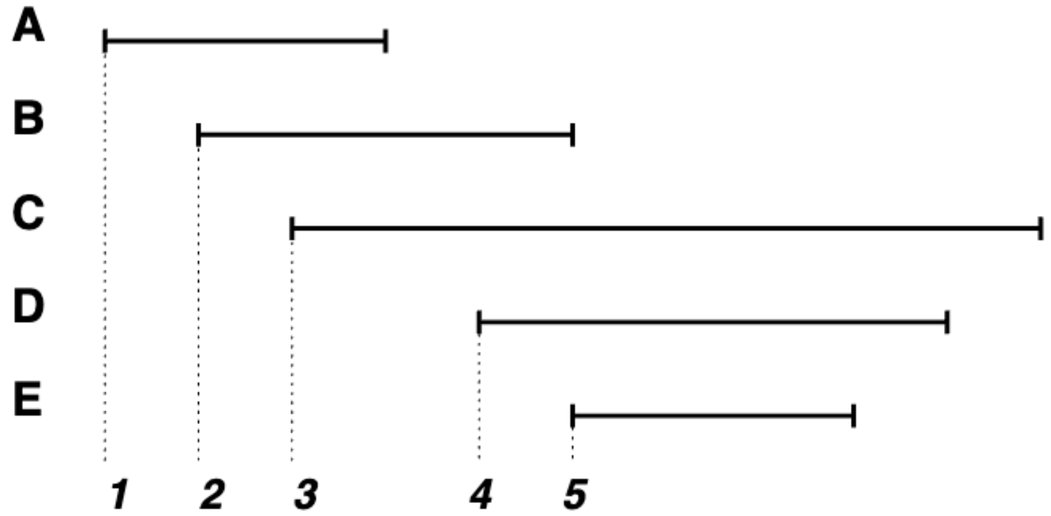
We describe a global register allocation algorithm, called *linear scan*, that is not based on graph coloring. Rather, given the live ranges of variables in a function, the algorithm scans all the live ranges in a single pass, allocating registers to variables in a greedy fashion. The algorithm is simple, efficient, and produces relatively good code. It is useful in situations where both compile time and code quality

This research was supported in part by the Advanced Research Projects Agency under contracts N00014-94-1-0985 and N66001-96-C-8522. Max Poletto was also supported by an NSF National Young Investigator Award awarded to Frans Kaashoek.
A synopsis of this algorithm first appeared in Poletto et al. [1997].

```
LinearScanRegisterAllocation
    active ← {}
    foreach live interval i, in order of increasing start point
        ExpireOldIntervals(i)
        if length(active) = R then
            SpillAtInterval(i)
        else
            register[i] ← a register removed from pool of free registers
            add i to active, sorted by increasing end point

ExpireOldIntervals(i)
    foreach interval j in active, in order of increasing end point
        if endpoint[j] ≥ startpoint[i] then
            return
        remove j from active
        add register[j] to pool of free registers

SpillAtInterval(i)
    spill ← last interval in active
    if endpoint[spill] > endpoint[i] then
        register[i] ← register[spill]
        location[spill] ← new stack location
        remove spill from active
        add i to active, sorted by increasing end point
    else
        location[i] ← new stack location
```



Important note: screenshots from the paper “Linear Scan Register Allocation” by Massimiliano Poletto and Vivek Sarkar

Register Allocation

- Generally challenging: requires at least live variables analysis
- Coloring-based approaches are typically slow; greedy algorithms do reasonably well
- At -O0, clang uses a linear scan allocator
- At -O1/2/3, clang uses a greedy allocator, extended with…
  - Live-range splitting, coalescing, rematerialization, and spill-weight heuristics
  - Want to reduce spills / copies, use variety of advanced heuristics to get close
- Always a time/space trade-off
  - Global/per-function register allocation can be costly
  - Linear scan allocation is O(n), very easy to implement, and works well