Lecture 2: Register Allocation

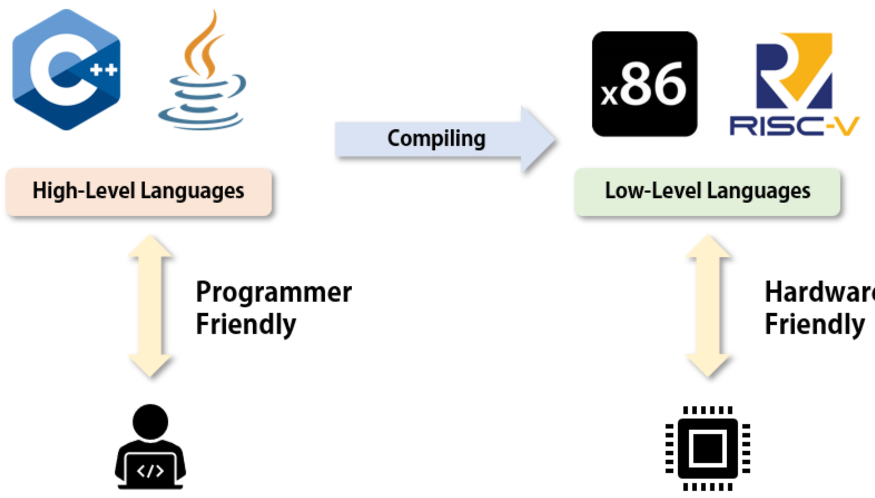
Chaofan Lin, TA of Advanced Compiler

Spring 2023

1 Background: Why RA?

1.1 Memory Abstractions from High Level to Low Level

Remembering that We are designing **compilers**, which translate program from high-level languages (cpp, Java) to low-level languages (x86, RISC-V).



Due to the different design goals, the **memory abstractions** they provide also differ a lot.

- High level languages assume that all variables we define live in **memory**. Disk (files), I/O and network are managed separately.
- Assembly languages explicitly have representations of **registers**, which is closer to the hardware architecture. In RISC, we use load/store instructions to access to memory.

And usually, registers in machines have the following characteristics:

- Fast to be accessed due to their locations in the hardware.
- Exist in small quantity. The number of registers are fixed and finite, determined in the hardware specification.

To guarantee the high-level abstraction, we need to put many variables into finitely many registers and try to make full use of them because of their high efficiency. In fact, this type of scenario is very common in computer system: we have a limited but fast memory A (registers, cache, memory, etc) and a unlimited (approximately) but slow memory B (memory, disk, etc). The target is to try make full use of A while still maintaining the functionality.

1.2 Challenges

Register allocation is hard not only because graph coloring is NP hard, but also because different hardware specifications are complicated and in order to implement a good allocator, you should be familiar with the architecture and some conventions. (Similar to the reason that writing CUDA is hard, because you need to be familiar with GPU architecture and make good use of resources such like shared memory.) The main challenges come from two parts:

- Commonly there are more variables in high-level than the registers. In order to allocate them to limited registers, **reusing** the registers is important.
- Specifications of registers are complicated. RISC-V is relatively simpler but there are still something like calling conventions (caller-saved / callee-saved registers) we need to pay attention to. For x86, some instructions require to store values in specific registers and they can't be allocated freely. And for most architecture, there are some registers which are reserved for assembler.

2 Naive Allocation

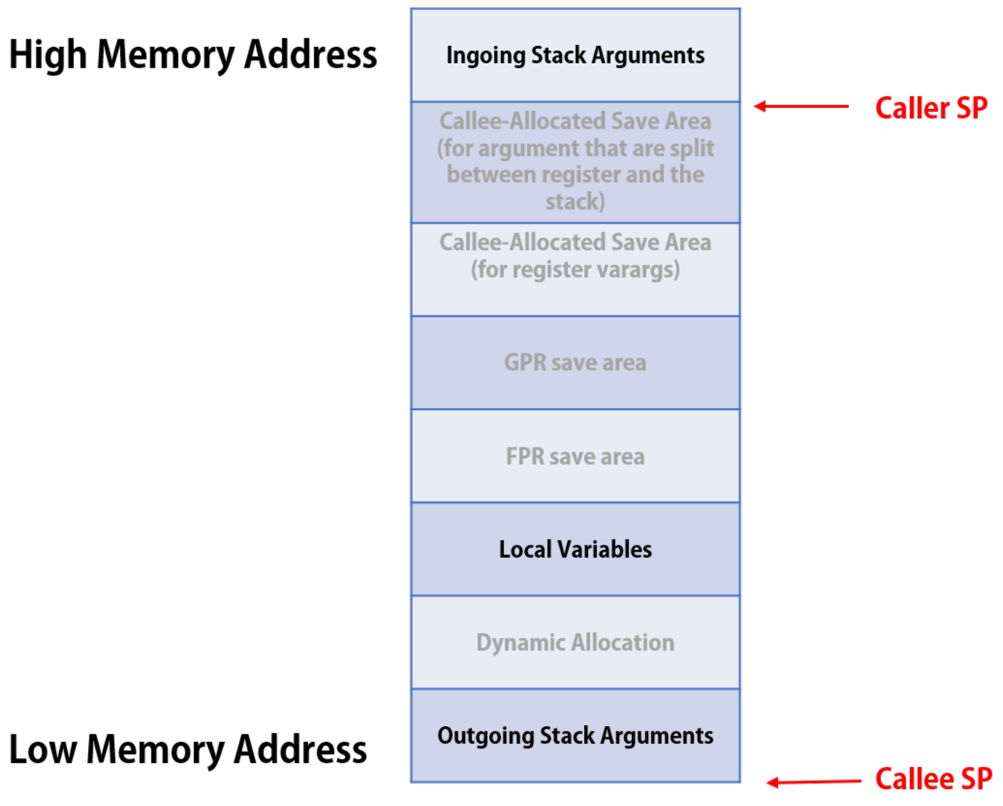
First let's look at a navie allocator: put all variables in the memory, load them to temporary registers, do the calculation and store them back. (This is the first allocator I wrote in my Compiler and it worked amazingly good, despite its gross inefficiency.)

For simplicity, we directly compile from a cpp-like language to a RISC-V like assembly, ignoring some unimportant technical details.

2.1 Stack Frame

Let's start from allocating registers for a function. Because we put all things in memory, we need first allocate a whole space, which we call it a **frame** or a **stack frame**. It contains local variables and some context information, which is specified by two pointers: **stack pointer (sp, low address)** and **frame pointer (fp, high address)**. In RISC-V fp(s0) is the x8 register and sp is the x2 register.

Fun fact: The frame pointer is **NOT** necessary indeed. Option '-fomit-frame-pointer' in gcc will try to optimize it.



As the figure shows, when a **caller** (outer function) calls a **callee** (inner function), a stack frame will be allocated to callee, which is the space between the callee's sp and the caller's sp. Here we focus on three segments:

- **Outgoing Stack Arguments**. When callee calls another function and it needs to pass the arguments to the "callee's callee", it uses this section to store the arguments. And the "callee's callee" will load them from this section.
- **Local Variables**. As its name, it's where we save local variables inside the callee function.
- **Ingoing Stack Arguments**. Indeed it's not the section of this frame but the caller's frame. It's the outgoing stack arguments section of the caller's frame and naturally becomes the ingoing stack arguments section of the callee, where callee loads its arguments.

So Local Variables section is where we put our variables on. Practically, we assign a unique address for every variable in this section. And when it comes to an instruction $c = a + b$, we load a and b from their address to registers like t0 and t1, perform the calculation and the result goes to t2, and store t2 back to the address of c.

2.2 Calling

After allocating the local variables, we need to handle the arguments passing problem when calling. Because we are creating a naive allocator, it will not use a0-a7. And even it uses, if we have more than 8 × 4 bytes arguments, we also need to put some data to the memory.

The place we put these arguments is the **outgoing stack arguments** as mentioned before.

Suppose we have a function called add and it takes two arguments:

```
    int add(int a, int b);
```

Then when calling add we need

```
    sw    t0, 0(sp)
    sw    t1, 4(sp)
    call  add
```

Another issue is how to return from a call. Specially, many architectures have a reserved register named **ra** to store the **return address** (i.e. the position in the assembly). And ret is just an alias of ”jump to ra”.

If there is a nested call, the ra register will be overwritten. So in the very beginning of every function, it will store ra to a specific space in the frame (The GPR save area).

And about the returned value, it will also be stored in specific registers, which are a0 and a1 in RISC-V (4-8 bytes).

2.3 Example

Try it by yourself in Compile Explorer! As an example, we compile the following cpp code with -O0 option. We add 8 dummy arguments to let the truly used arguments a, b spilled to the memory to see what happen:

```
int add(int d1, int d2, int d3, int d4, int d5, int d6, int d7, int d8, int a, int b) {
  int c;
  c = a + b;
  return c;
}

int main() { add(0, 0, 0, 0, 0, 0, 0, 0, 233, 666); }
```

The following is the important code piece:

```
    main:
        addi    sp,sp,-32
        sw      ra,28(sp)
        sw      s0,24(sp)
        addi    s0,sp,32
        li      a5,666
        sw      a5,4(sp)
        li      a5,233
        sw      a5,0(sp)
        ...
        call    add
        ...
    add:
        addi    sp,sp,-64
        sw      s0,60(sp)
        addi    s0,sp,64
        ...
        lw      a4,0(s0)
        lw      a5,4(s0)
        add     a5,a4,a5
        sw      a5,-20(s0)
        lw      a5,-20(s0)
        mv      a0,a5
        lw      s0,60(sp)
        addi    sp,sp,64
        jr      ra
```

To be summarized, when entering a new function, a stack frame is created as follows:

- Lower the stack pointer (sp).
- Save the outer context information (s0, ra) to memory because they will be overwritten.
- Define new frame pointer by s0 = sp + [frame size].
- Loading the arguments and execute instructions in the function body.
- Saving the return value to a0 and a1.
- Load the outer context information (s0, ra) back.
- Jump to ra (Ret).

Recall the two challenges in the register allocation. Indeed the naive allocator escapes from the first problem (Since it claimes to be naive) but can't get around with the second problem. We still need to dive into the specifications and get our hands dirty.

3 Linear Scan Register Allocation

3.1 Liveness

Since load/store are slow, a better solution is to put variables in the registers. So specifically, we need to answer the following two questions:

- Which variables should be put in the registers and how do we reuse them? (Since the registers are limited, **choices** are important to some extent)
- What do we do when we **run out of** registers?

A key observation here is that all variables have their **live range**, which means the same register can store different variables in different time. (Imaging that if all variables are always live, then we can't reuse any register. So the optimal solution is that: choosing top-K most frequently accessed variables and allocating registers for them.)

We have the following definitions to build our liveness analysis theory:

Definition 1. A variable is **live** if it may be read in the later before it is written.

Commonly, we use words "define (def)" for writing a variable, and "use" for reading a variable. Indeed if a variable is written, it can be seen as a newly created variable (Treat old variable as "dead"!).

Definition 2. The **live range** of a variable is the set of points (i.e. lines in assembly code) at which that variable is live.

By definition, a live range consists of several disjoint intervals. For example, the live range of a variable may look like $[0, 2]$, $[4, 7]$, $[9, 15]$. Every interval is started by a def and ends with another def.

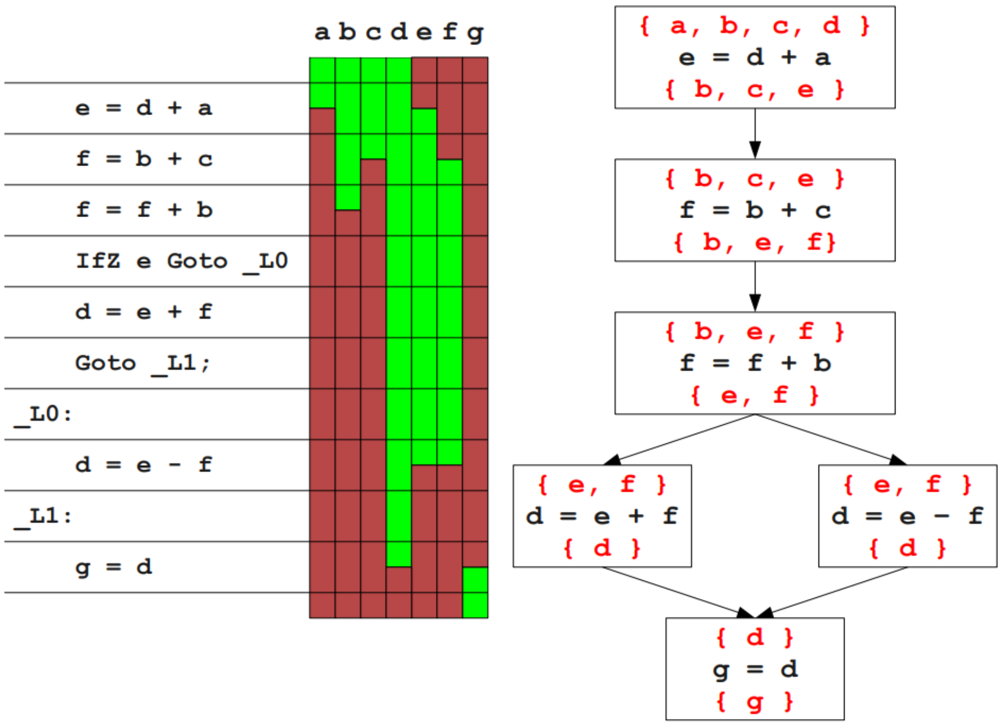
Definition 3. The **live interval** of a variable is the smallest interval containing the live range.

The corresponding live interval of the above example is $[0, 15]$. As we can see, the live interval is a **conservative estimate** of the live range since it ignores holes (Those areas where the variable is not live). But the live interval is easy to manipulate so it serves as the unit in linear scan allocation.

Question 1: Can we treat a written variable as a new variable and then every interval in the live range becomes a live interval?

The answer is certainly YES. We can further optimize it by utilizing lifetime holes, which will soon be introduced in 3.3. And a more direct way is do this in SSA form! This approach indeed shortens the length of live interval. Check this paper [?].

In the following example, each instruction contains two timestamp units: before and after the instruction. And we add a extra time unit at the beginning and end of the program. A live interval begins in the first def (except some pre-declared variables like a, b, c, d) and ends in the last use.



If two intervals are overlapping, we say they are **interfered** mutually. Obviously, two interfered variables can't be allocated with the same register.

3.2 Basic Linear Scan Algorithm

Linear Scan Register Allocation (LSRA) proceeds the live intervals sorted by their start positions. and allocate registers greedily:

- If a live interval begins, it selects a free register and allocate.
- If a live interval ends, the allocated register is marked as free again.

- If a live interval begins but **no free registers**, spill it.

To implement it, the algorithm maintains a **active** list which contains all intervals that overlap with the current position and have registers assigned.

The liveness information is collected by a dataflow analysis algorithm. If there is no conditions or loops in the program, the liveness is easy to compute. To take conditions and loops into account, a CFG-based **liveness analysis** is needed because the logic is no longer linear.

LSRA is simple so the allocation algorithm **runs very fast**, which is why it is usually used in JIT compilers (such as Hotspot). And the allocated result is also not bad.

However, there are also some shortages. Although it's "not bad", it's not good enough and there are many existing algorithms which is superior to it. The main reason is that it uses live interval to estimate the live range conservatively.

3.3 Second Chance Binpacking

It's a more aggresive version of LSRA proposed in [**?**]. As we said in **Question 1**, the basic LSRA ignore holes in the live intervals. To further improve the allocation, a **binpacking** model is proposed to address this problem in a finer grain size. And when handling spill, it adopts a **second change** allocation.

3.3.1 Binpacking Model

This method views each register as a "bin" and we pack variables' lifetime into these bins. We can try to reuse our bins in two aspects:

- Put two non-overlapping lifetimes into the same bin.
- Assign two variables in the same bin if one's lifetime is **entirely contained in a lifetime hole** of the other.

Practically, when processing a variable, the binpacking allocator finds a free register for it. In this context, the "free register" refers to not only those non-occupied registers but those registers containing in lifetime holes. We can also view the former as registers containing in some virtual lifetime holes. And the allocator heuristically chooses the register **with the smallest hole that is larger than the variable's lifetime**.

3.3.2 Second Chance Allocation

When we allocate a register to a variable, we may **spill another variable** to create a free register. This spilling strategy is a heuristic one that compares the distance to **each variable's next**

reference and weighted by **the depth of the loop** it occurs in. The allocator evicts the one with the lowest priority. What this method does is trying to **split variables' life intervals**.

When we next meet this spilled variable, we will allocate a new register r for it and load it into r. The allocation process is the same with the above (Eviction based on some strategies). This gives it a **second chance** in allocating.

And in the end, a CFG-based dataflow analysis called the **resolution** pass should run on the program since the allocation is in a linear ordering of blocks, which may affect **register consistency**. This pass is similar with the PhiResolve pass we have written in our Advanced Compile Design, which adds some extra move instructions.

3.4 Greedy Register Allocator in LLVM

LLVM leverages an improved version of LSRA called **Greedy Register Allocation**. It uses a priority queue since it's a greedy algorithm.

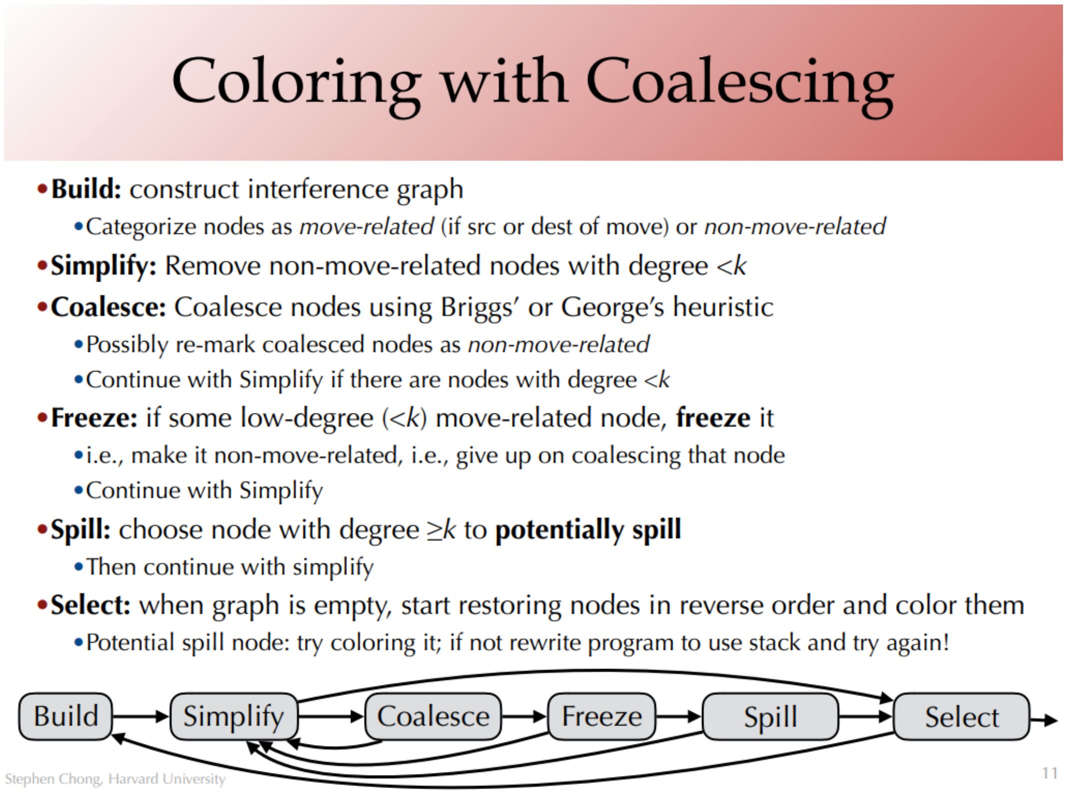
- It has 7 stages to describe the state of a live range.
- It calculates weights as the comparison basis of the priority queue. Approximately, it chooses live ranges with longer length to assign. And when it comes to spill, it performs **eviction** (The anti-operation of the register allocation, i.e. unbind a pair of virtual register and physical register) according to the spill weight.

More details can be found in: Greedy Register Allocation in LLVM 3.0.

4 Graph Coloring Register Allocation

Given that we use graph coloring in our Compiler Project, I believe you guys have been already familiar with it. It builds a **register interference graph (RIG)** and reduce the allocation problem to the coloring problem in this graph. Now the most widely known version is the one proposed by Chaitin [**?**].

The algorithm described in Tiger Book is a relatively well-established version containing **move coalesce** and some heuristic spilling strategies (Briggs' criterion and George's criterion).



Coloring with Coalescing

- **Build:** construct interference graph
  - Categorize nodes as *move-related* (if src or dest of move) or *non-move-related*
- **Simplify:** Remove non-move-related nodes with degree $<k$
- **Coalesce:** Coalesce nodes using Briggs' or George's heuristic
  - Possibly re-mark coalesced nodes as *non-move-related*
  - Continue with Simplify if there are nodes with degree $<k$
- **Freeze:** if some low-degree ($<k$) move-related node, **freeze** it
  - i.e., make it non-move-related, i.e., give up on coalescing that node
  - Continue with Simplify
- **Spill:** choose node with degree $\geq k$ to **potentially spill**
  - Then continue with simplify
- **Select:** when graph is empty, start restoring nodes in reverse order and color them
  - Potential spill node: try coloring it; if not rewrite program to use stack and try again!

$$\begin{tikzcd}
\text{Build} \arrow[r] & \text{Simplify} \arrow[r] \arrow[rrrr, bend left] & \text{Coalesce} \arrow[r] \arrow[l, bend left] & \text{Freeze} \arrow[r] \arrow[ll, bend left] & \text{Spill} \arrow[r] \arrow[lll, bend left] & \text{Select} \arrow[r] \arrow[lllll, bend left] & {}
\end{tikzcd}$$

Stephen Chong, Harvard University

11

5 Integer Linear Programming Register Allocation

0-1 Integer Programming (IP) (Yuhao must talked about this topic!) can also be used in solving register allocation problems. The core idea is to use the integer programming constraints to model the constraints of register allocation.

The actual modeling is complex because it needs to take many factors into account. Here we give a simple formalization. Let $x_{ij} \in \{0, 1\}$ be a binary decision variable that indicates whether variable i is assigned to register j or not.

The objective function is to minimize the number of spills. Here we simply let c_{ij} be the possible cost of spilling and reloading variable i if it is assigned to register j. Then the objective function can be expressed as:

$$\min \sum_i \sum_j c_{ij} x_{ij}$$

And there are a few constraints:

- Each variable must be assigned to exactly one register (Here, we have already splitted and renamed these variables so that we only need to consider allocating one register for each variable):

$$\forall i,\ \sum_j x_{ij} \leq 1$$

- Two interfering variables can't be assigned to the same register:

  $$\forall (i,k) \in E(G),\ \forall j,\ x_{ij} + x_{kj} \leq 1$$

  where E(G) is the set of edges in the interference graph.

This formalization is not such direct. It's more like you first convert it to the graph coloring problem and then reduce to 0-1 IP program. Anyway, that's one way to think about it.

6 Partitioned Quadratic Programming Register Allocation

Partitioned Boolean Quadratic Problem (PBQP) is another NP-complete problem which also has a good ability of modeling. The problem is defined as follows.

Definition 4. We have $n \times n$ matrices $C_{ij},\ 1 \leq i \leq n, 1 \leq j \leq n$. The shapes of the matrices are not the same. The solution we want to find is a set of vectors $\vec{x_i},\ 1 \leq i \leq n$. More formally, PBQP is defined as follows:

$$\min \sum_{1 \leq i \leq n} \sum_{1 \leq j \leq n} \vec{x_i}^T C_{ij} \vec{x_j}$$

$$s.t.\ \vec{x_i} \in \{0,1\}^{|\vec{c_i}|},\ \vec{x_i}^T \mathbf{1} = \mathbf{1}$$

To apply this problem in register allocation, we use $\vec{x_i}$ to represent the register selected for the variable v_i. For example, $\{0,1,0,0\}$ indicates we allocate the second physics register to it.

We only allow two possible values in the cost matrix C_{ij}: inf and 0. The value $C_{ij}[a,b]$ means the cost of allocating register a to variable v_i and allocating register b to variable v_j. Then inf means there is a interference in this allocation behaviour while 0 means it is available.

You can also find the PBQP allocator in the LLVM codebase.

Acknowledgement

Many thanks to the following resources which help me write this note greatly:

- CS143 Course Slide17, Standford: Link

- RISC-V Function Frame by @lhtin: Link
- RISC-V Calling Convetions: Link
- Linear Scan Register Allocation for the Java HotSpot Client Compiler: Link
- A Survey on Register Allocation: Link

References

[1] C. Wimmer and M. Franz, "Linear scan register allocation on ssa form," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, (New York, NY, USA), p. 170–179, Association for Computing Machinery, 2010.

[2] O. Traub, G. Holloway, and M. D. Smith, "Quality and speed in linear-scan register allocation," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, (New York, NY, USA), p. 142–151, Association for Computing Machinery, 1998.

[3] G. Chaitin, "Register allocation and spilling via graph coloring," *SIGPLAN Not.*, vol. 39, p. 66–74, apr 2004.