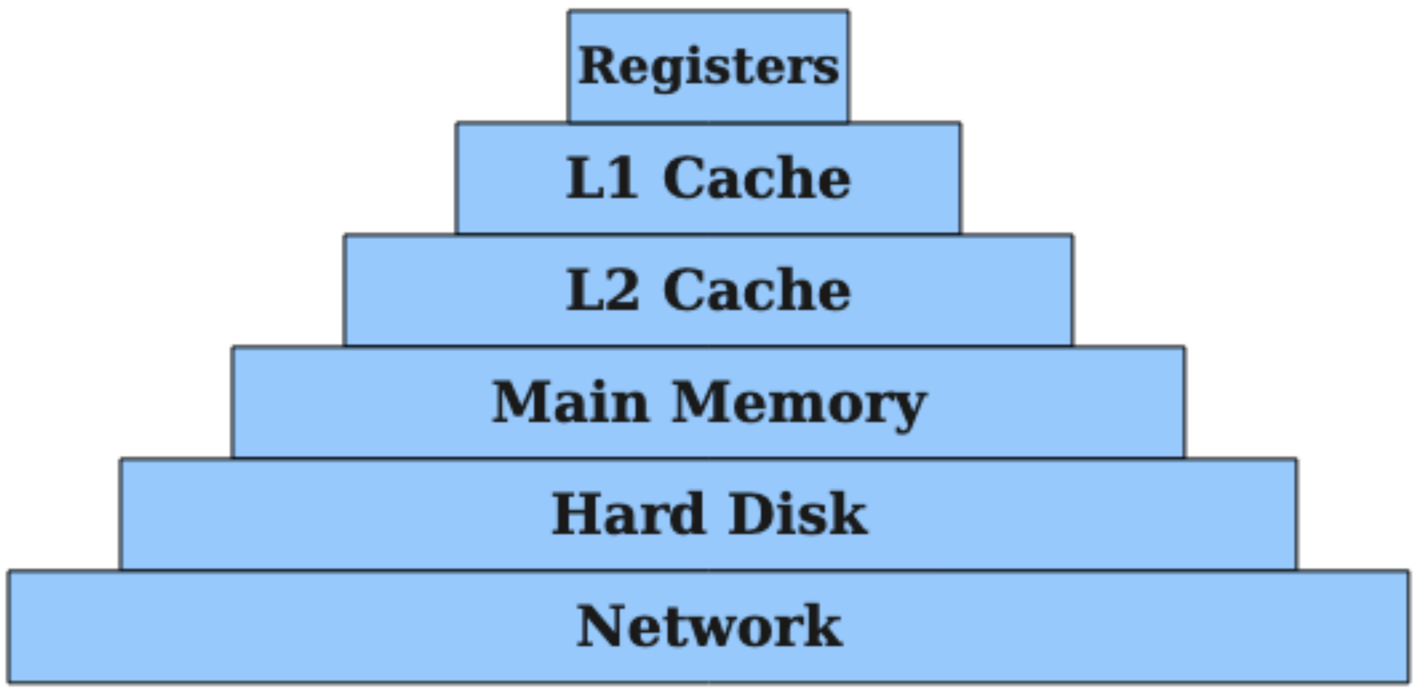
Register Allocation I

Oct 13, 2021

Memory Hierarchy

faster, smaller

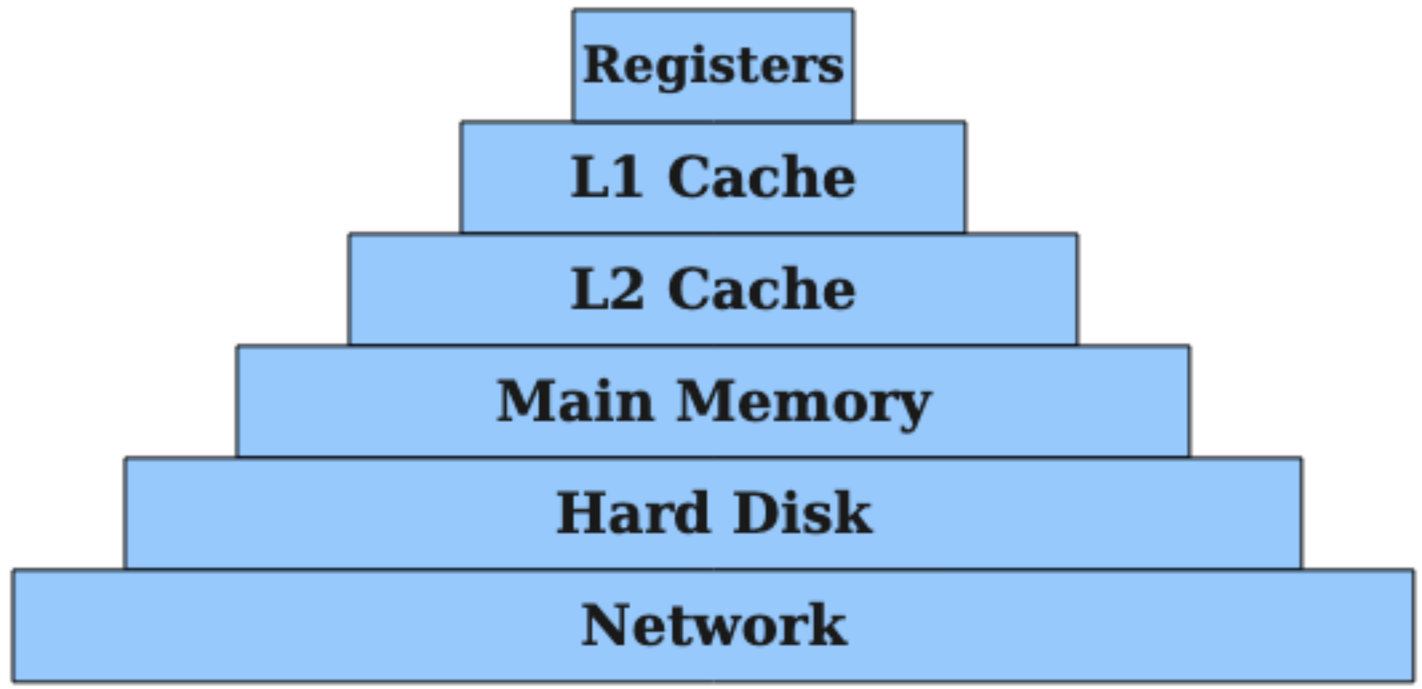


256B - 8KB	0.25 – 1ns
16KB – 64KB	1ns – 5ns
1MB - 4MB	5ns – 25ns
4GB – 256GB	25ns – 100ns
500GB+	3 – 10ms
HUGE	10 – 2000ms

slower, bigger

Memory Hierarchy

Systems view of memory:



Program view of memory

variables, arrays, structs

Register Allocation

The compiler needs to decide which variables to store in which registers, which registers to "spill" onto the stack

Register Allocation

The compiler needs to decide which variables to store in which registers, which registers to "spill" onto the stack

So far: all variables are stored on the stack

Easy, but slow, especially if we get cache misses.

Register Allocation

The compiler needs to decide which variables to store in which registers, which registers to "spill" onto the stack

So far: all variables are stored on the stack

Easy, but slow, especially if we get cache misses.

Performance gains:

- 3-10x+ faster variable accesses
- Pervasive speedup: variables are ubiquitous
- Most useful optimization in the compiler, also the most computationally expensive to perform

Examples

```
def f(a):
  let x = a * 2 in
  let y = x + 7 in
  y
end
```

Examples

```
def f(a):
  let x = a * 2 in
  let y = x + 7 in
  y
end
```

```
Currently:
a: stack
x: stack (rbp - 8)
y: stack (rbp - 16)
```

Examples

```
def f(a):
  let x = a * 2 in
  let y = x + 7 in
  y
end
```

```
With register
alloc:
a: stack
x: rax
y: rax
```

Examples

```
def f(a):
  let x = a * 2 in
  let y = x + 7 in
  y
end

mov rax, [rbp + ..]
sar rax, 1
imul rax, 4
add rax, 14
```

```
With register
alloc:
a: stack
x: rax
y: rax
```

Examples

```
def f(a):
  let x = a * 2 in
  let y = x + 7 in
  g(x, y)
end
```

Examples

```
def f(a):
  let x = a * 2 in
  let y = x + 7 in
  g(x, y)
end
```

```
With register
alloc:
a: stack
x: rax
y: rbx
```

Examples

```
def f(a):
  let x = a * 2 in
  let y = x + 7 in
  g(x, y)
end
```

```
With register
alloc:
a: stack
x: rax
y: rbx
```

Can't put x and y in the same register because they need to hold different values at the same time

Register Allocation

4 Steps

Register Allocation

4 Steps

1. Liveness analysis: identify which variables are needed at every point in the program

Register Allocation

4 Steps

1. Liveness analysis: identify which variables are needed at every point in the program
2. Conflict analysis: based on liveness info, identify which variables *cannot* be assigned the same register

Register Allocation

4 Steps

1. Liveness analysis: identify which variables are needed at every point in the program
2. Conflict analysis: based on liveness info, identify which variables *cannot* be assigned the same register
3. Graph Coloring: based on conflict information, assign registers to variables so that conflicting vars get different registers

Register Allocation

4 Steps

1. Liveness analysis: identify which variables are needed at every point in the program
2. Conflict analysis: based on liveness info, identify which variables *cannot* be assigned the same register
3. Graph Coloring: based on conflict information, assign registers to variables so that conflicting vars get different registers
4. Spilling: if graph coloring fails, pick a variable to put on the stack and retry

Shadowing

```
def f(a):
  let x = a * 2 in
  let y = let x = 14 in x + 7 in
  f(x, y)
end
```

Shadowing

```
def f(a):
  let x = a * 2 in
  let y = let x = 14 in x + 7 in
  f(x, y)
end
```

two different "x" are in conflict here

Shadowing

```
def f(a):
    let x0 = a * 2 in
    let y = let x1 = 14 in x1 + 7 in
    f(x0, y)
end
```

Before reg allocation:
make all variable names unique.

Limitation: Computability

Rice's Theorem

Any non-trivial semantic property of programs in a Turing-complete language is undecidable.

Limitation: Computability

Rice's Theorem

Any non-trivial semantic property of programs in a Turing-complete language is undecidable.

Our takeaway: any interesting program analyses must be an approximation

Liveness Analysis

Goal: determine at each point in the program which variables are "alive"

Liveness Analysis

Goal: determine at each point in the program which variables are “alive”

- i.e., their values will be needed later in the program
- overapproximate on the side of too many variables are

Liveness Analysis

Goal: determine at each point in the program which variables are "alive"

- i.e., their values will be needed later in the program
- overapproximate on the side of too many variables are

Basic idea: start at the end and work backwards

Liveness Analysis

1. At the end of a function, no variables are live
2. If a variable is used in an expression, it is alive immediately preceding that expression
3. If a variable is assigned to, it is dead immediately preceding the let
4. In an if, we run the analysis on both branches and take the union of the results

Liveness Analysis

```
def f(a, b):
  let x = a + b in
  let y = g(x)  in
  let z = x * y in
  h(x, z)
end
```

Liveness Analysis

```
def f(a, b):
  let x = a + b in
  let y = g(b) in
  let z = if b:
            x + 1
          else:
            y
  in h(z)
end
```

Conflict Analysis

Once we know when we need the value of each variable, we determine which variables cannot be assigned the same register

Conflict Analysis

Once we know when we need the value of each variable, we determine which variables cannot be assigned the same register

2 variables truly conflict when

- They are live at the same time

Conflict Analysis

Once we know when we need the value of each variable, we determine which variables cannot be assigned the same register

2 variables truly conflict when

- They are live at the same time
- with different values

Conflict Analysis

Once we know when we need the value of each variable, we determine which variables cannot be assigned the same register

2 variables truly conflict when

- They are live at the same time
- with different values

Algorithm overview: look at every **let** and add an associated conflict

Conflict Analysis

For each let in the program, check the variables that are live before the **body** of the let.

For each variable in that set, add a conflict with the let bound variable unless

1. It's the same variable
2. The two variables **must** have the same value

Conflict Analysis

```
def f(a, b):
  let x = a + b in
  let y = g(x)  in
  let z = x * y in
  h(x, z)
end
```

Conflict Analysis

```
def f(a, b):
  let x = a + b in
  let y = x in
  h(x, y)
end
```

Conflict Analysis

```
def f(a, b):
  let x = a + b in
  let y = let z = print(10) in x in
  h(x, y)
end
```

Conflict Analysis

```
def f(a, b):
  let x = a + b in
  let y = print(x) in
  h(x, y)
end
```

Conflict Analysis

```
def f(a, b):
  let x = a + b in
  let y = if b: print(x) else: x in
  h(x, y)
end
```

Conflict Analysis

```
def f(a, b):
  let x = a + b in
  let y = if b: print(x) else: x in
  h(x, y)
end
```

Start with something simple and iterate from there

Summary so Far

For each function in the program

1. Liveness Analysis annotates each expression with which variables are live immediately **before** the expression runs.
2. Conflict Analysis produces a **conflict graph** whose nodes are variables and edges are conflicts (the variables cannot share a register)
3. Next time: Use this conflict graph to assign registers to variables