Register Allocation

Register Allocation

When we generated IR, we held temporary results in virtual registers that we "allocated" from a presumably infinite register file.

When we do code generation, we must face the reality of having a limited set of registers, including those that may have architecture-defined purposes (stack pointer, frame pointer, etc.).

Thus we have a mapping problem: How do we map the elements from the virtual register set into the real architectural registers?

First Approach: Do Nothing

There is a way to skip the issue.

Allocate every variable to memory:

- Locals in activation records
- Globals in data segment
- Member variables Heap allocated variables

Only bring a value from memory when you need it for a calculation.

Immediately store the result back to its memory location.

Naïve Approach Drawbacks

This approach is much too naïve for practical use.

Memory accesses are expensive, regardless of the cache structure.

We may have dozens of temporaries whose values are calculated once, used once, and then are never necessary again. This results in huge allocations.

However, this is easy to generate code for and to get right (our #1 priority in code generation).

Often this is what you will get from a compiler like gcc with no optimizations on (the default).

Better Approach

We may note that the idea of saving elements back to memory is unnecessary if we immediately use them again.

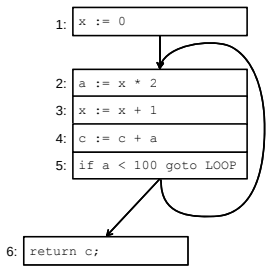
Can we identify when a value is useful versus when it does not need to be stored in a register?

Liveness Analysis can tell us the useful range of a value.

A value is **live** if its current value will be useful again at a point in the future.

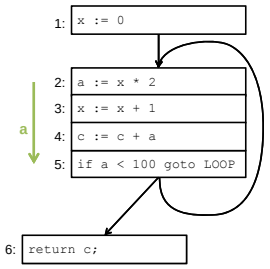
Liveness Analysis

```
        x := 0
LOOP:   a := x * 2
        x := x + 1
        c := c + a
        if a < 100 goto LOOP
        return c;
```



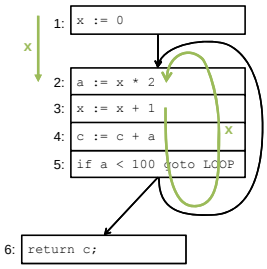
Liveness Analysis

```
        x := 0
LOOP:   a := x * 2
        x := x + 1
        c := c + a
        if a < 100 goto LOOP
        return c;
```



Liveness Analysis

```
        x := 0
LOOP:   a := x * 2
        x := x + 1
        c := c + a
        if a < 100 goto LOOP
        return c;
```



Liveness Analysis

The liveness of a value follows the control flow of the program.

We consider the set of problems where we follow data throughout the program a **dataflow** problem.

Control Flow

We have visited the problem of control flow and constructed a control flow graph (CFG) to determine the basic blocks' relation to each other.

Some definitions we will prefer to use:

A CFG node represents a basic block, and by our definition, a basic block has a single entry point and a single exit point.

For each incoming edge to the start of a basic block, we have a control-flow **predecessor**.

For each outgoing edge from the end of a basic block, we have a control-flow **successor**.

Defs and Uses

A definition (**def**) of a variable or temporary occurs when the symbol appears as the left hand side of an assignment.

A **use** of a variable is an occurrence of the symbol on the left hand side of an expression.

We can use these concepts then to describe the set of variable defs and uses that a node in the CFG defines.

Liveness can then be defined by saying:

A variable is **live** on a control-flow edge if there is a directed path from that edge to a *use* of that variable that does not go through a *def*.

Liveness Calculation

Liveness information of each node *n* of our CFG can be calculated as follows:

- If a node contains a use of a variable, the variable is live on the entry to the block (use[*n*] implies live-in[*n*])
- If a variable is live-in at node *n*, then it is live-out in all of the CFG predecessors of *n*
- If a variable is live-out at node *n*, and not in def[*n*], then the variable is also live-in at n

In equation form:

$$in[n] = use[n] \cup (out[n] - def[n])$$

$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

Liveness Calculation Algorithm

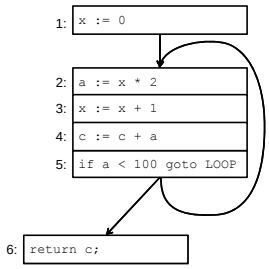
```
for each n
        in[n] ← {}
        out[n] ← {}
end for

repeat
        for each n                    //(in reverse DFS order)
                in'[n] ← in[n]
                out'[n] ← out[n]
                in[n] ← use[n] ∪ (out[n] − def [n])

                out[n] ← ∪ in[s]      (s ∈ succ[n])
        end for
until in'[n] = in[n] and out[n] = out'[n] for all n
```

Liveness Analysis

Block	Use	Def
6	c	
5	a	
4	c, a	c
3	x	x
2	x	a
1		x

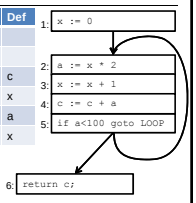


Liveness Analysis

Start with block 6:

```
in'[6] = in[6] = {}
out'[6] = out[6] ={}
in[6] = use[6] ∪ (out[6]-def[6])
in[6] = {c} ∪ ( {} − {} )
out[6] = Union over successors
No successors
Done.
```

Block	Use	Def
6	c	
5	a	
4	c, a	c
3	x	x
2	x	a
1		x



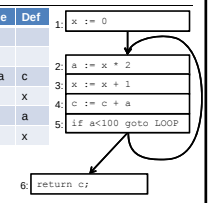
	In	Out
6	c	
5		
4		
3		
2		
1		

Liveness Analysis

Next go to block 5:

```
in'[5] = in[5] = {}
out'[5] = out[5] = {}
in[5] = use[5] ∪ (out[5]-def[5])
in[5] = {a} ∪ ( {} − {} )
out[5] = in[6] ∪ in[2] = {c}
```

Block	Use	Def
6	c	
5	a	
4	c, a	c
3	x	x
2	x	a
1		x



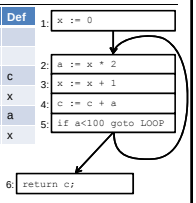
	In	Out
6	c	
5	a	c
4		
3		
2		
1		

Liveness Analysis

Next go to block 4:

```
in'[4] = in[4] = {}
out'[4] = out[4] = {}
in[4] = use[4] ∪ (out[4]-def[4])
in[4] = {c,a} ∪ ( {} − {c} )
out[4] = in[5] = {a}
```

Block	Use	Def
6	c	
5	a	
4	c, a	c
3	x	x
2	x	a
1		x



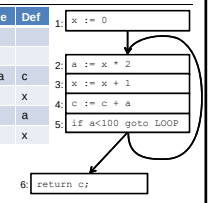
	In	Out
6	c	
5	a	c
4	c, a	a
3		
2		
1		

Liveness Analysis

Next go to block 3:

```
in'[3] = in[3] = {}
out'[3] = out[3] = {}
in[3] = use[3] ∪ (out[3]-def[3])
in[3] = {x} ∪ ( {} − {x} )
out[3] = in[4] = {c, a}
```

Block	Use	Def
6	c	
5	a	
4	c, a	c
3	x	x
2	x	a
1		x



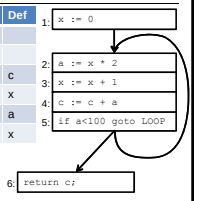
	In	Out
6	c	
5	a	c
4	c, a	a
3	x	c, a
2		
1		

Liveness Analysis

Next go to block 2:

```
in'[2] = in[2] = {}
out'[2] = out[2] = {}
in[2] = use[2] ∪ (out[2]-def[2])
in[2] = {x} ∪ ( {} – {a} )
out[2] = in[3] = {x}
```

Block	Use	Def
6	c	
5	a	
4	c, a	c
3	x	x
2	x	a
1		x



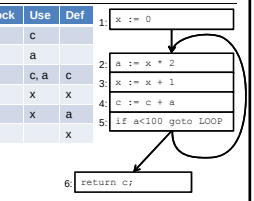
	In	Out
6	c	
5	a	c
4	c, a	a
3	x	c, a
2	x	x
1		

Liveness Analysis

Finally block 1:

```
in'[1] = in[1] = {}
out'[1] = out[1] = {}
in[1] = use[1] ∪ (out[1]-def[1])
in[1] = {} ∪ ( {} – {x} )
out[1] = in[2] = {x}
```

Block	Use	Def
6	c	
5	a	
4	c, a	c
3	x	x
2	x	a
1		x



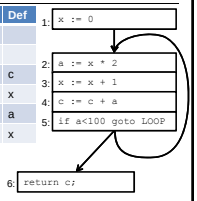
	In	Out
6	c	
5	a	c
4	c, a	a
3	x	c, a
2	x	x
1		x

Liveness Analysis

Since the sets changed in the first go through, we must do it all again, starting with Block 6:

```
in'[6] = in[6] = {c}
out'[6] = out[6] = {}
in[6] = use[6] ∪ (out[6]-def[6])
in[6] = {c} ∪ ( {} – {} )
out[6] =
```

Block	Use	Def
6	c	
5	a	
4	c, a	c
3	x	x
2	x	a
1		x



	In	Out	In	Out
6	c		c	
5	a	c		
4	c, a	a		
3	x	c, a		
2	x	x		
1		x		

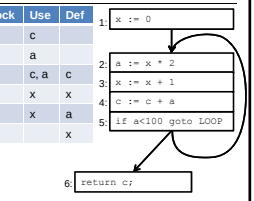
No Change

Liveness Analysis

Block 5:

```
in'[5] = in[5] = {a}
out'[5] = out[5] = {c}
in[5] = use[5] ∪ (out[5]-def[5])
in[5] = {a} ∪ ( {c} – {} )
out[5] = in[6] ∪ in[2] = {c,x}
```

Block	Use	Def
6	c	
5	a	
4	c, a	c
3	x	x
2	x	a
1		x



	In	Out	In	Out
6	c		c	
5	a	c	a, c	c, x
4	c, a	a		
3	x	c, a		
2	x	x		
1		x		

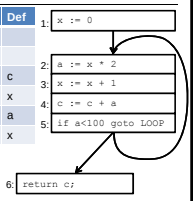
Changed

Liveness Analysis

Block 4:

```
in'[4] = in[4] = {a}
out'[4] = out[4] = {a}
in[4] = use[4] ∪ (out[4]-def[4])
in[4] = {c, a} ∪ ( {a} – {c} )
out[4] = in[5] = {a,c}
```

Block	Use	Def
6	c	
5	a	
4	c, a	c
3	x	x
2	x	a
1		x



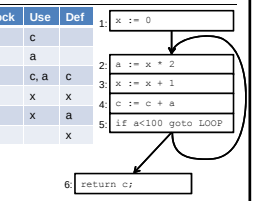
	In	Out	In	Out
6	c		c	
5	a	c	a, c	c, x
4	c, a	a	a, c	a, c
3	x	c, a		
2	x	x		
1		x		

Liveness Analysis

Block 3:

```
in'[3] = in[3] = {x}
out'[3] = out[3] = {c, a}
in[3] = use[3] ∪ (out[3]-def[3])
in[3] = {x} ∪ ( {c,a} – {x} )
out[3] = in[4] = {a,c}
```

Block	Use	Def
6	c	
5	a	
4	c, a	c
3	x	x
2	x	a
1		x



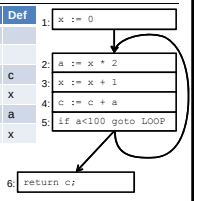
	In	Out	In	Out
6	c		c	
5	a	c	a, c	c, x
4	c, a	a	a, c	a, c
3	x	c, a	a, c, x	a, c
2	x	x		
1		x		

Liveness Analysis

Block 2:

```
in'[2] = in[2] = {x}
out'[2] = out[2] = {x}
in[2] = use[2] ∪ (out[2]-def[2])
in[2] = {x} ∪ ( {x} - {a} )
out[2] = in[3] = {a,c,x}
```

Block	Use	Def
6	c	
5	a	
4	c, a	c
3	x	x
2	x	a
1		x



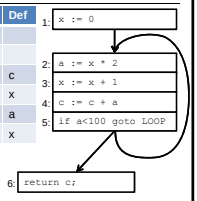
	In	Out	In	Out
6	c		c	
5	a	c	a, c	c, x
4	c, a	a	a, c	a, c
3	x	c, a	a, c, x	a, c
2	x	x	x	a, c, x
1		x		

Liveness Analysis

Block 1:

```
in'[1] = in[1] = {}
out'[1] = out[1] = {x}
in[1] = use[1] ∪ (out[1]-def[1])
in[1] = {} ∪ ( {x} - {x} )
out[1] = in[2] = {x}
```

Block	Use	Def
6	c	
5	a	
4	c, a	c
3	x	x
2	x	a
1		x



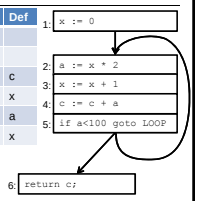
	In	Out	In	Out
6	c		c	
5	a	c	a, c	c, x
4	c, a	a	a, c	a, c
3	x	c, a	a, c, x	a, c
2	x	x	x	a, c, x
1		x		x

Liveness Analysis

Since the sets changed, we must do it all again, starting with Block 6:

```
in'[6] = in[6] = {c}
out'[6] = out[6] = {}
in[6] = use[6] ∪ (out[6]-def[6])
in[6] = {c} ∪ ( {} - {} )
out[6] =
```

Block	Use	Def
6	c	
5	a	
4	c, a	c
3	x	x
2	x	a
1		x



	In	Out	In	Out	In	Out
6	c		c		c	
5	a	c	a, c	c, x		
4	c, a	a	a, c	a, c		
3	x	c, a	a, c, x	a, c		
2	x	x	x	a, c, x		
1		x		x		

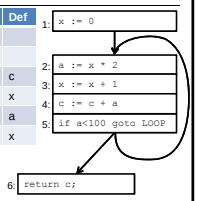
No Change

Liveness Analysis

Block 5:

```
in'[5] = in[5] = {a, c}
out'[5] = out[5] = {c, x}
in[5] = use[5] ∪ (out[5]-def[5])
in[5] = {a} ∪ ( {c,x} - {} )
out[5] = in[6] ∪ in[2] = {c,x}
```

Block	Use	Def
6	c	
5	a	
4	c, a	c
3	x	x
2	x	a
1		x



	In	Out	In	Out	In	Out
6	c		c		c	
5	a	c	a, c	c, x	a, c, x	c, x
4	c, a	a	a, c	a, c		
3	x	c, a	a, c, x	a, c		
2	x	x	x	a, c, x		
1		x		x		

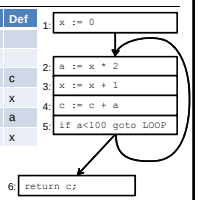
Changed

Liveness Analysis

Block 4:

```
in'[4] = in[4] = {a, c}
out'[4] = out[4] = {a, c}
in[4] = use[4] ∪ (out[4]-def[4])
in[4] = {a, c} ∪ ( {a,c} - {c} )
out[4] = in[5] = {a,c,x}
```

Block	Use	Def
6	c	
5	a	
4	c, a	c
3	x	x
2	x	a
1		x



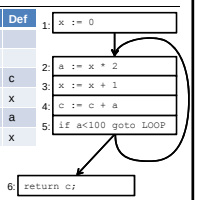
	In	Out	In	Out	In	Out
6	c		c		c	
5	a	c	a, c	c, x	a, c, x	c, x
4	c, a	a	a, c	a, c	a, c	a, c, x
3	x	c, a	a, c, x	a, c		
2	x	x	x	a, c, x		
1		x		x		

Liveness Analysis

Block 3:

```
in'[3] = in[3] = {a, c, x}
out'[3] = out[3] = {a, c}
in[3] = use[3] ∪ (out[3]-def[3])
in[3] = {x} ∪ ( {a,c} - {x} )
out[3] = in[4] = {a,c}
```

Block	Use	Def
6	c	
5	a	
4	c, a	c
3	x	x
2	x	a
1		x



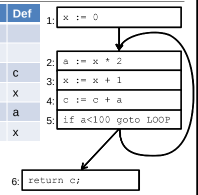
	In	Out	In	Out	In	Out
6	c		c		c	
5	a	c	a, c	c, x	a, c, x	c, x
4	c, a	a	a, c	a, c	a, c	a, c, x
3	x	c, a	a, c, x	a, c	a, c, x	a, c
2	x	x	x	a, c, x		
1		x		x		

Liveness Analysis

Block 2:

```
in'[2] = in[2] = {x}
out'[2] = out[2] = {a, c, x}
in[2] = use[2] ∪ (out[2]-def[2])
in[2] = {x} ∪ ( {a,c,x} - {a} )
out[2] = in[3] = {a,c,x}
```

Block	Use	Def
6	c	
5	a	
4	c, a	c
3	x	x
2	x	a
1		x



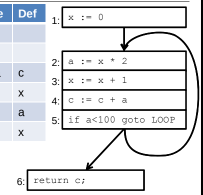
	In	Out	In	Out	In	Out
6	c		c		c	
5	a	c	a, c	c, x	a, c, x	c, x
4	c, a	a	a, c	a, c	a, c	a, c, x
3	x	c, a	a, c, x	a, c	a, c, x	a, c
2	x	x	x	a, c, x	a, c, x	a, c, x
1		x		x		

Liveness Analysis

Block 1:

```
in'[1] = in[1] = {}
out'[1] = out[1] = {x}
in[1] = use[1] ∪ (out[1]-def[1])
in[1] = {} ∪ ( {x} - {x} )
out[1] = in[2] = {a,c,x}
```

Block	Use	Def
6	c	
5	a	
4	c, a	c
3	x	x
2	x	a
1		x



	In	Out	In	Out	In	Out
6	c		c		c	
5	a	c	a, c	c, x	a, c, x	c, x
4	c, a	a	a, c	a, c	a, c	a, c, x
3	x	c, a	a, c, x	a, c	a, c, x	a, c
2	x	x	x	a, c, x	a, c, x	a, c, x
1		x		x		a, c, x

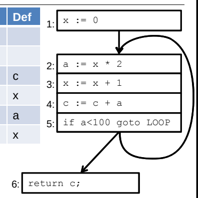
Liveness Analysis

It will take one more iteration to converge.

```
in[4] = use[4] ∪ (out[4]-def[4])
in[4] = {a, c} ∪ ( {a,c,x} - {c} )

out[3] = in[4] = {a,c,x}
```

Block	Use	Def
6	c	
5	a	
4	c, a	c
3	x	x
2	x	a
1		x



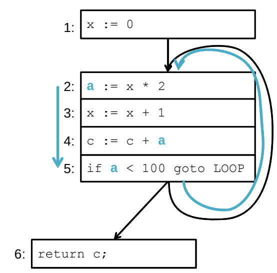
	In	Out	In	Out	In	Out
6	c		c		c	
5	a	c	a, c	c, x	a, c, x	c, x
4	c, a	a	a, c	a, c	a, c	a, c, x
3	x	c, a	a, c, x	a, c	a, c, x	a, c
2	x	x	x	a, c, x	a, c, x	a, c, x
1		x		x		a, c, x

use[4] ∪ (out[4]-def[4])

in[4]

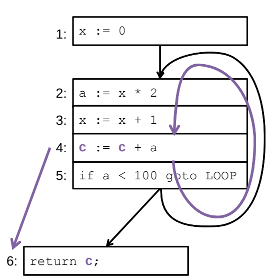
Liveness Analysis

	In	Out
1		a, c, x
2	a, c, x	a, c, x
3	a, c, x	a, c, x
4	a, c, x	a, c, x
5	a, c, x	c, x
6	c	



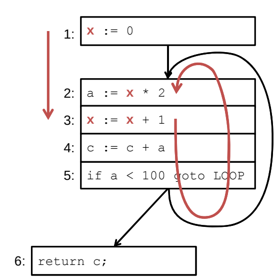
Liveness Analysis

	In	Out
1		a, c, x
2	a, c, x	a, c, x
3	a, c, x	a, c, x
4	a, c, x	a, c, x
5	a, c, x	c, x
6	c	



Liveness Analysis

	In	Out
1		a, c, x
2	a, c, x	a, c, x
3	a, c, x	a, c, x
4	a, c, x	a, c, x
5	a, c, x	c, x
6	c	



Notes on Liveness

The liveness algorithm we have presented will converge regardless of the order that the blocks are visited in, however since out depends on the control flow successors, working in the reverse flow order will generally converge faster.

The algorithm we have presented is conservative: it may overestimate the liveness.

To do this perfectly, we'd have to solve the halting problem.

Since we cannot, we have two choices:

1. Potentially underestimate and have our wrong solution influence our decisions and ultimately produce wrong code
2. Overestimate and sometimes produce code that is not as good as if we had perfect knowledge

Notes on Liveness (2)

We are much more likely to apply this algorithm at the basic block level rather than the statement level. This also reduces the number of nodes in the graph, something that affects the performance of the algorithm.

The performance of the algorithm depends upon the number of variables, the number of blocks, and how much work the algorithm does each step. In the worst case, the algorithm is $O(n^4)$ but in practice, using the reverse CFG order, it is usually $O(n^2)$ or less.

Data Structures for Implementation

We need a set data structure. We can implement one using a linked list or, if our data is dense, use a bit vector.

A bit vector allows us to assign a bit in a word to represent something like the presence of a particular variable in a given block's def, use, in, or out set.

For example:
In block 5, we have an out set of {c, x}. We can represent this as:

a	c	x
0	1	1

java.util.BitSet

```
BitSet()
BitSet(int nbits)
```

`and(BitSet set)` – set intersection
`or(BitSet set)` – set union
`andNot(BitSet set)` – set difference

set/get/clear/flip, etc.

Interference Graphs

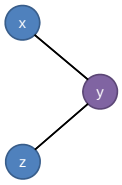
The liveness information we have computed can be used for a variety of optimizations in the compiler.

The most immediate and beneficial one to us on typical machine architectures will be for register allocation.

Imagine we have three variables: x, y, and z and two registers $r1 and $r2 to use.

We can build an interference graph like the one on the right to express overlapping live ranges for the variables.

From this we can discover that x and z do not overlap and can be assigned to the same register by the allocator.



Move (Copy) Instructions

If we have a statement of the form:

$t_2 := t_1$

And subsequent uses of both t_1 and t_2 later in the program, their live ranges overlap and an edge (t_1, t_2) would be added to the interference graph.

However, since it is the same value in each, we do not need to keep them in separate registers.

Building the Interference Graph

Considering this, there are two rules for when to add an edge to the interference graph:

1. A non-move definition of a variable a with live-out variables $b_1, \ldots, b_n$
   - Add edges $(a, b_1) \ldots (a, b_n)$
2. A move $a := c$ with live-out variables $b_1, \ldots, b_n$
   - Add edges $(a, b_1) \ldots (a, b_n)$ for all $b_i \neq c$