Liveness Analysis

Akim Demaille Étienne Renault Roland Levillain
first.last@lrde.epita.fr

EPITA — École Pour l'Informatique et les Techniques Avancées

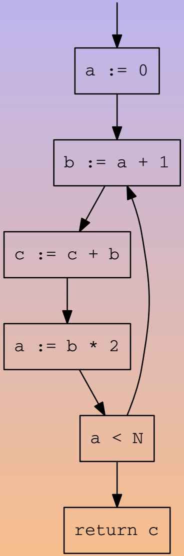
May 19, 2018

Control Flow Graph [Appel, 1998]

```
    a := 0
L1: b := a + 1
    c := c + b
    a := b * 2
    if a < N goto L1
    return c
```

Control Flow Graph [Appel, 1998]

```
    a := 0
L1: b := a + 1
    c := c + b
    a := b * 2
    if a < N goto L1
    return c
```



7.tig

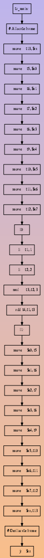
```
1 + 2 * 3
```

7's Pre-Assembly

```
tc_main:
# Allocate frame
        move    $x13, $ra
        move    $x5, $s0
        move    $x6, $s1
        move    $x7, $s2
        move    $x8, $s3
        move    $x9, $s4
        move    $x10, $s5
        move    $x11, $s6
        move    $x12, $s7
l0:
        li      $x1, 1
        li      $x2, 2
        mul     $x3, $x2, 3
        add     $x4, $x1, $x3
```

```
l1:
        move    $s0, $x5
        move    $s1, $x6
        move    $s2, $x7
        move    $s3, $x8
        move    $s4, $x9
        move    $s5, $x10
        move    $s6, $x11
        move    $s7, $x12
        move    $ra, $x13
# Deallocate frame
        jr      $ra
```

7's Flowgraph



7000.tig

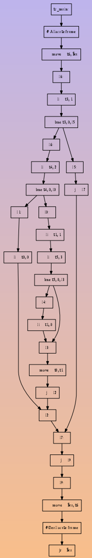
```
1 | 2 & 3
```

7000's Pre-Assembly

```
tc_main:
# Allocate frame
        move    $x6, $ra
l8:
        li      $x3, 1
        bne     $x3, 0, l5
l6:
        li      $x4, 2
        bne     $x4, 0, l0
l1:
        li      $x0, 0
l2:
l7:
        j       l9
```

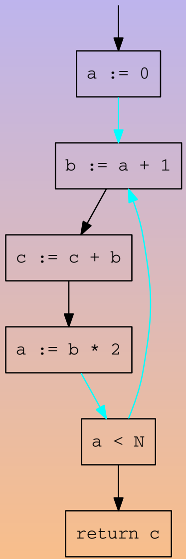
```
l0:
        li      $x1, 1
        li      $x5, 3
        bne     $x5, 0, l3
l4:
        li      $x1, 0
l3:
        move    $x0, $x1
        j       l2
l5:
        j       l7
l9:
        move    $ra, $x6
# Deallocate frame
        jr      $ra
```

7000's Flowgraph

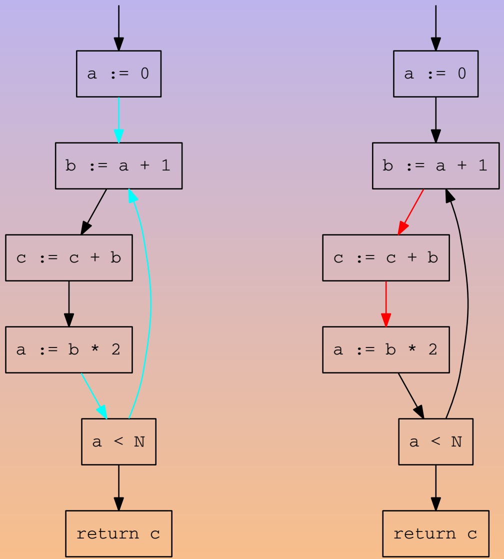


Liveness

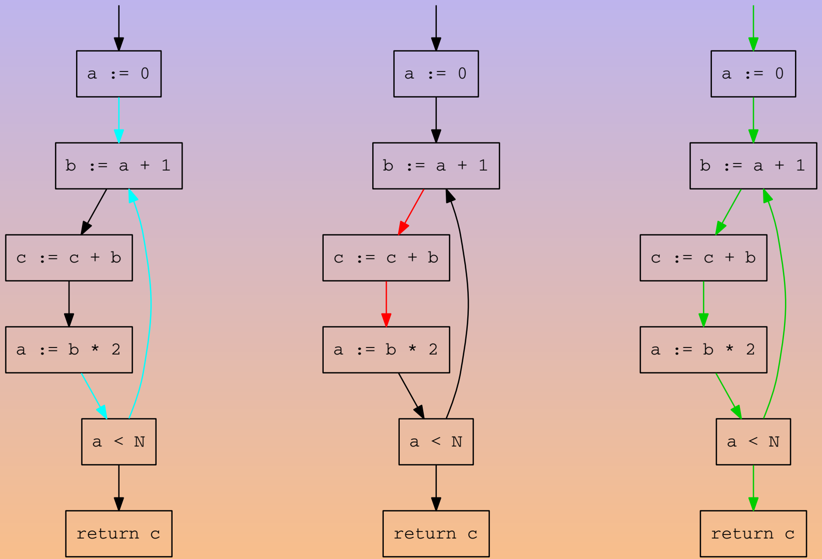
Liveness



Liveness



Liveness


a := 0
b := a + 1
c := c + b
a := b * 2
a < N
return c
a := 0
b := a + 1
c := c + b
a := b * 2
a < N
return c
a := 0
b := a + 1
c := c + b
a := b * 2
a < N
return c

Dataflow Equations for Liveness Analysis

$$
\begin{aligned}
\mathrm{in}[n] &= \mathrm{use}[n] \cup (\mathrm{out}[n] \setminus \mathrm{def}[n]) \\
\mathrm{out}[n] &= \bigcup_{s \in \mathrm{succ}[n]} \mathrm{in}[s]
\end{aligned}
$$

Liveness Calculation

	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a								
2	a	b								
3	bc	c								
4	b	a								
5	a									
6	c									

	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*
1		a						
2	a	b						
3	bc	c						
4	b	a						
5	a							
6	c							

$$\begin{aligned} \mathrm{in}[n] &= \mathrm{use}[n] \cup (\mathrm{out}[n] \setminus \mathrm{def}[n]) \\ \mathrm{out}[n] &= \bigcup_{s \in \mathrm{succ}[n]} \mathrm{in}[s] \end{aligned}$$

Liveness Calculation

	use	def	in (1st step)	out (1st step)	in	out	in	out	in	out
1		a								
2	a	b	a							
3	bc	c	bc							
4	b	a	b							
5	a		a	a						
6	c		c							

	use	def	in	out	in	out	in	out
1		a						
2	a	b						
3	bc	c						
4	b	a						
5	a							
6	c							

$$\mathrm{in}[n] = \mathrm{use}[n] \cup (\mathrm{out}[n] \setminus \mathrm{def}[n])$$

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

Liveness Calculation

	use	*def*	*in* (1st step)	*out* (1st step)	*in* (2nd step)	*out* (2nd step)	*in*	*out*	*in*	*out*
1		a				a				
2	a	b	a		a	bc				
3	bc	c	bc		bc	b				
4	b	a	b		b	a				
5	a		a	a	a	ac				
6	c		c		c					

	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*
1		a						
2	a	b						
3	bc	c						
4	b	a						
5	a							
6	c							

$$\mathrm{in}[n] = \mathrm{use}[n] \cup (\mathrm{out}[n] \setminus \mathrm{def}[n])$$

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

Liveness Calculation

			1st step		2nd step		3rd step			
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a				a		a		
2	a	b	a		a	bc	ac	bc		
3	bc	c	bc		bc	b	bc	b		
4	b	a	b		b	a	b	a		
5	a		a	a	a	ac	ac	ac		
6	c		c		c		c			

	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*
1		a						
2	a	b						
3	bc	c						
4	b	a						
5	a							
6	c							

$$\begin{aligned} \mathrm{in}[n] &= \mathrm{use}[n] \cup (\mathrm{out}[n] \setminus \mathrm{def}[n]) \\ \mathrm{out}[n] &= \bigcup_{s \in \mathrm{succ}[n]} \mathrm{in}[s] \end{aligned}$$

Liveness Calculation

			1st step		2nd step		3rd step		4th step	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a				a		a		ac
2	a	b	a		a	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	c
4	b	a	b		b	a	b	a	b	ac
5	a		a	a	a	ac	ac	ac	ac	ac
6	c		c		c		c		c	

	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*
1		a						
2	a	b						
3	bc	c						
4	b	a						
5	a							
6	c							

$$\mathrm{in}[n] = \mathrm{use}[n] \cup (\mathrm{out}[n] \setminus \mathrm{def}[n])$$

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

Liveness Calculation

			1st step		2nd step		3rd step		4th step	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a				a		a		ac
2	a	b	a		a	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	c
4	b	a	b		b	a	b	a	b	ac
5	a		a	a	a	ac	ac	ac	ac	ac
6	c		c		c		c		c	

			5th step					
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*
1		a	c	ac				
2	a	b	ac	bc				
3	bc	c	bc	b				
4	b	a	bc	ac				
5	a		ac	ac				
6	c		c					

$$\begin{aligned} \mathrm{in}[n] &= \mathrm{use}[n] \cup (\mathrm{out}[n] \setminus \mathrm{def}[n]) \\ \mathrm{out}[n] &= \bigcup_{s \in \mathrm{succ}[n]} \mathrm{in}[s] \end{aligned}$$

Liveness Calculation

			1st step		2nd step		3rd step		4th step	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a				a		a		ac
2	a	b	a		a	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	c
4	b	a	b		b	a	b	a	b	ac
5	a		a	a	a	ac	ac	ac	ac	ac
6	c		c		c		c		c	

			5th step		6th step			
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*
1		a	c	ac	c	ac		
2	a	b	ac	bc	ac	bc		
3	bc	c	bc	b	bc	bc		
4	b	a	bc	ac	bc	ac		
5	a		ac	ac	ac	ac		
6	c		c		c			

$$\mathrm{in}[n] = \mathrm{use}[n] \cup (\mathrm{out}[n] \setminus \mathrm{def}[n])$$

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

Liveness Calculation

	use	def	1st step in	1st step out	2nd step in	2nd step out	3rd step in	3rd step out	4th step in	4th step out
1		a				a		a		ac
2	a	b	a		a	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	c
4	b	a	b		b	a	b	a	b	ac
5	a		a	a	a	ac	ac	ac	ac	ac
6	c		c		c		c		c	

	use	def	5th step in	5th step out	6th step in	6th step out	7th step in	7th step out
1		a	c	ac	c	ac	c	ac
2	a	b	ac	bc	ac	bc	ac	bc
3	bc	c	bc	b	bc	bc	bc	bc
4	b	a	bc	ac	bc	ac	bc	ac
5	a		ac	ac	ac	ac	ac	ac
6	c		c		c		c	

$$\text{in}[n] = \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n])$$

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

Liveness Calculation (Forward)

			1st step		2nd step		3rd step		4th step	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1		a				a		a		ac
2	a	b	a		a	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	c
4	b	a	b		b	a	b	a	b	ac
5	a		a	a	a	ac	ac	ac	ac	ac
6	c		c		c		c		c	

			5th step		6th step		7th step	
	use	*def*	*in*	*out*	*in*	*out*	*in*	*out*
1		a	c	ac	c	ac	c	ac
2	a	b	ac	bc	ac	bc	ac	bc
3	bc	c	bc	b	bc	bc	bc	bc
4	b	a	bc	ac	bc	ac	bc	ac
5	a		ac	ac	ac	ac	ac	ac
6	c		c		c		c	

$$\begin{aligned} \mathrm{in}[n] &= \mathrm{use}[n] \cup (\mathrm{out}[n] \setminus \mathrm{def}[n]) \\ \mathrm{out}[n] &= \bigcup_{s \in \mathrm{succ}[n]} \mathrm{in}[s] \end{aligned}$$

Calculation done following forward control-flow edges.

Liveness Calculation (Backward)

	use	def	out	in	out	in	out	in
6	c							
5	a							
4	b	a						
3	bc	c						
2	a	b						
1		a						

$$\begin{aligned} \text{in}[n] &= \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n]) \\ \text{out}[n] &= \bigcup_{s \in \text{succ}[n]} \text{in}[s] \end{aligned}$$

Calculation done following *reverse* control-flow edges.

Liveness Calculation (Backward)

			1st step					
	use	*def*	*out*	*in*	*out*	*in*	*out*	*in*
6	c			c				
5	a		c	ac				
4	b	a	ac	bc				
3	bc	c	bc	bc				
2	a	b	bc	ac				
1		a	ac	c				

$$\text{in}[n] = \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n])$$

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

Calculation done following *reverse* control-flow edges.

Liveness Calculation (Backward)

			1st step		2nd step			
	use	*def*	*out*	*in*	*out*	*in*	*out*	*in*
6	c			c		c		
5	a		c	ac	ac	ac		
4	b	a	ac	bc	ac	bc		
3	bc	c	bc	bc	bc	bc		
2	a	b	bc	ac	bc	ac		
1		a	ac	c	ac	c		

$$\begin{aligned} \mathrm{in}[n] &= \mathrm{use}[n] \cup (\mathrm{out}[n] \setminus \mathrm{def}[n]) \\ \mathrm{out}[n] &= \bigcup_{s \in \mathrm{succ}[n]} \mathrm{in}[s] \end{aligned}$$

Calculation done following *reverse* control-flow edges.

Liveness Calculation (Backward)

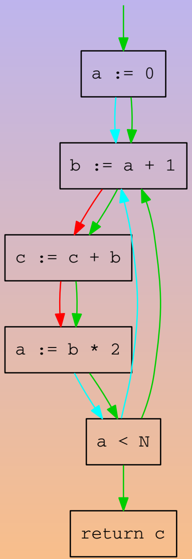
			1st step		2nd step		3rd step	
	use	*def*	*out*	*in*	*out*	*in*	*out*	*in*
6	c			c		c		c
5	a		c	ac	ac	ac	ac	ac
4	b	a	ac	bc	ac	bc	ac	bc
3	bc	c	bc	bc	bc	bc	bc	bc
2	a	b	bc	ac	bc	ac	bc	ac
1		a	ac	c	ac	c	ac	c

$$\mathrm{in}[n] = \mathrm{use}[n] \cup (\mathrm{out}[n] \setminus \mathrm{def}[n])$$
$$\mathrm{out}[n] = \bigcup_{s \in \mathrm{succ}[n]} \mathrm{in}[s]$$

Calculation done following *reverse* control-flow edges.

Control Flow Graph [Appel, 1998]

```
    a := 0
L1: b := a + 1
    c := c + b
    a := b * 2
    if a < N goto L1
    return c
```



Conservative Approximation

Suppose d a variable not used in the fragment of code

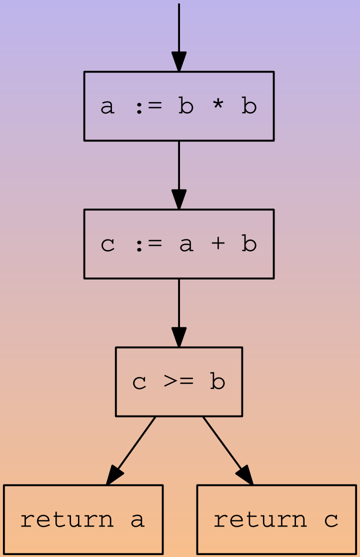
			Another Solution	
	use	*def*	*out*	*in*
1		a		
2	a	b		
3	bc	c		
4	b	a		
5	a			
6	c			

Conservative Approximation

Suppose d a variable not used in the fragment of code

			Another Solution	
	use	*def*	*out*	*in*
1		a	cd	acd
2	a	b	acd	bcd
3	bc	c	bcd	bcd
4	b	a	bcd	acd
5	a		acd	acd
6	c		c	

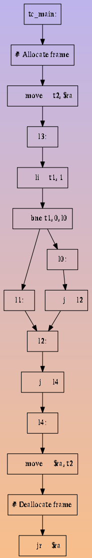
Conservative Approximation

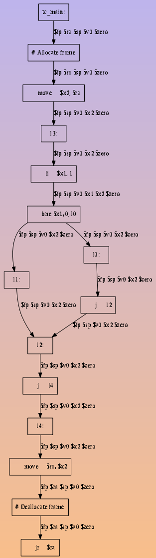


ors.tig

```
1 | 2
```

ors' Flowgraph




tc_main:
$fp $ra $sp $v0 $zero
Allocate frame
$fp $ra $sp $v0 $zero
move $x2, $ra
$fp $sp $v0 $x2 $zero
l3:
$fp $sp $v0 $x2 $zero
li $x1, 1
$fp $sp $v0 $x1 $x2 $zero
bne $x1, 0, l0
$fp $sp $v0 $x2 $zero
$fp $sp $v0 $x2 $zero
l0:
l1:
$fp $sp $v0 $x2 $zero
$fp $sp $v0 $x2 $zero
j l2
$fp $sp $v0 $x2 $zero
l2:
$fp $sp $v0 $x2 $zero
j l4
$fp $sp $v0 $x2 $zero
l4:
$fp $sp $v0 $x2 $zero
move $ra, $x2
$fp $ra $sp $v0 $zero
Deallocate frame
$fp $ra $sp $v0 $zero
jr $ra

Various Dataflow Analysis

Optimizing Compiler

- First step toward optimizing compilers
- How definitions and uses are related to each other
- What value a variable may have at a given point
- Constant propagation
- Common sub-expression elimination
- Copy propagation
- Dead Code Elimination

Constant propagation

An ambiguous definition is a statement that might or not assign a temporary *t*. For instance, a call may sometimes modifies *t* and sometimes not.

We don't have this problem for tiger due to excaping variables.

Don't loose optimisation! Consider every definiton as ambiguous

We need to define the set of definitions that reach the begining and the end of each node.

- *gen*: when enter this statement, we know that we will reach the end of it
- kills: any statement that invalidates a *gen*
- begin[n]: which statements can reach the begining of statement *n*
- end[n]: which statements can reach the end of statement *n*

Reaching definition [Appel, 1998]

```
    a := 5
    c := 1
L1: if c > a goto L2
    c := c + c
    goto L1
L2: a := c - a
    c := 0
```

	gen	*kills*	*begin*	*end*	*begin*	*end*	*begin*	*end*
1	1	6						
2	2	4,7						
3								
4	4	2,7						
5								
6	6	1						
7	7	2,4						

$$\mathrm{end}[n] = \mathrm{gen}[n] \cup (\mathrm{begin}[n] \setminus \mathrm{kills}[n])$$

$$\mathrm{begin}[n] = \bigcup_{p \in \mathrm{pred}[n]} \mathrm{end}[p]$$

			1st step					
	gen	*kills*	*begin*	*end*	*begin*	*end*	*begin*	*end*
1	1	6		1				
2	2	4,7	1	1,2				
3			1,2	1,2				
4	4	2,7	1,2	1,4				
5			1,4	1,4				
6	6	1	1,2	2,6				
7	7	2,4	2,6	6,7				

$$\mathrm{end}[n] = \mathrm{gen}[n] \cup (\mathrm{begin}[n] \setminus \mathrm{kills}[n])$$

$$\mathrm{begin}[n] = \bigcup_{p \in \mathrm{pred}[n]} \mathrm{end}[p]$$

	gen	kills	1st step begin	1st step end	2nd step begin	2nd step end	begin	end
1	1	6		1		1		
2	2	4,7	1	1,2	1	1,2		
3			1,2	1,2	1,2,4	1,2,4		
4	4	2,7	1,2	1,4	1,2,4	1,4		
5			1,4	1,4	1,4	1,4		
6	6	1	1,2	2,6	1,2,4	2,4,6		
7	7	2,4	2,6	6,7	2,4,6	6,7		

$$\text{end}[n] = \text{gen}[n] \cup (\text{begin}[n] \setminus \text{kills}[n])$$

$$\text{begin}[n] = \bigcup_{p \in \text{pred}[n]} \text{end}[p]$$

			1st step		2nd step		3rd step	
	gen	*kills*	*begin*	*end*	*begin*	*end*	*begin*	*end*
1	1	6		1		1		1
2	2	4,7	1	1,2	1	1,2	1	1,2
3			1,2	1,2	1,2,4	1,2,4	1,2,4	1,2,4
4	4	2,7	1,2	1,4	1,2,4	1,4	1,2,4	1,4
5			1,4	1,4	1,4	1,4	1,4	1,4
6	6	1	1,2	2,6	1,2,4	2,4,6	1,2,4	2,4,6
7	7	2,4	2,6	6,7	2,4,6	6,7	2,4,6	6,7

$$\mathrm{end}[n] = \mathrm{gen}[n] \cup (\mathrm{begin}[n] \setminus \mathrm{kills}[n])$$

$$\mathrm{begin}[n] = \bigcup_{p \in \mathrm{pred}[n]} \mathrm{end}[p]$$

Constant Propagation

- If we have a statement $d_1 : t := c$, with c constant, and another statement d_2 that uses t.
- t is constant
- if d_1 reaches d_2 **and** no other definition of t reaches d_2
- then we can rewrite d_2

In the previous example, only one definition of a reaches statement 3 so we can replace $c > a$ by $c > 5$.

Copy Propagation

- If we have a statement $d_1 : t := z$, with z variable, and another statement d_2 that uses t.
- t is constant
- if d_1 reaches d_2 **and** no other definition of t reaches d_2 **and** the is no definition of z in all pathes between d_1 and d_2
- then we can rewrite d_2

Good register allocator will automatically detects some such cases.

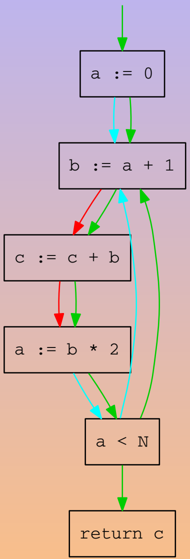
Optimizing compiler

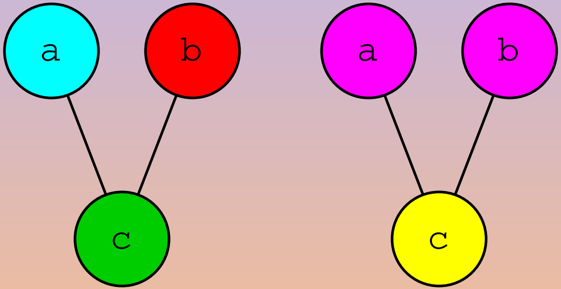
The removal of dead statements (or other optimizations) might introduce new dead statements.

To avoid the need for repeated global calculation, several strategies exist:

- Cutoff: perform no more than k round
- Cascading analysis: predict the cascade of effects of an optimization. Value numbering is a typical case of cascading analysis
- Incremental dataflow analysis: patch the dataflow after applying an optimization.

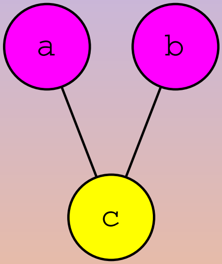
Interference Graph





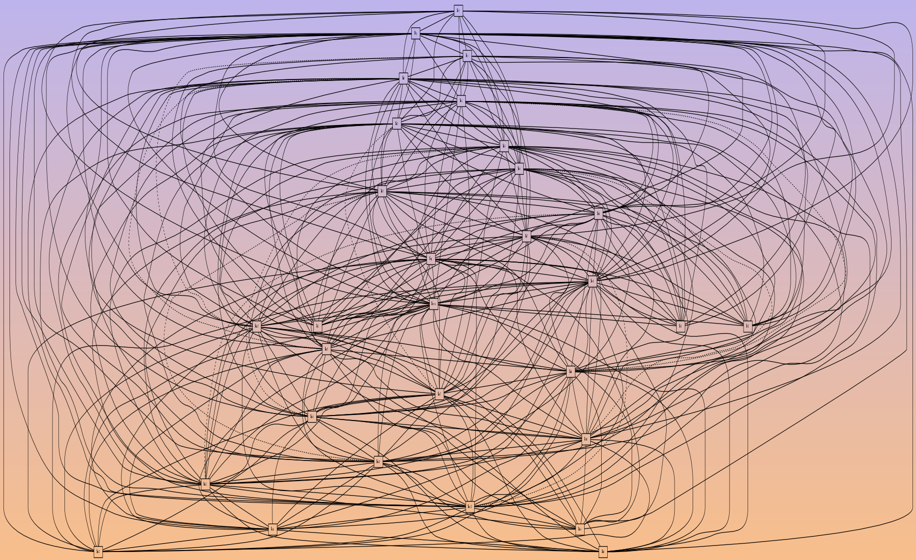
Register Allocation

```
    a := 0
L1: b := a + 1
    c := c + b
    a := b * 2
    if a < N goto L1
    return c
```

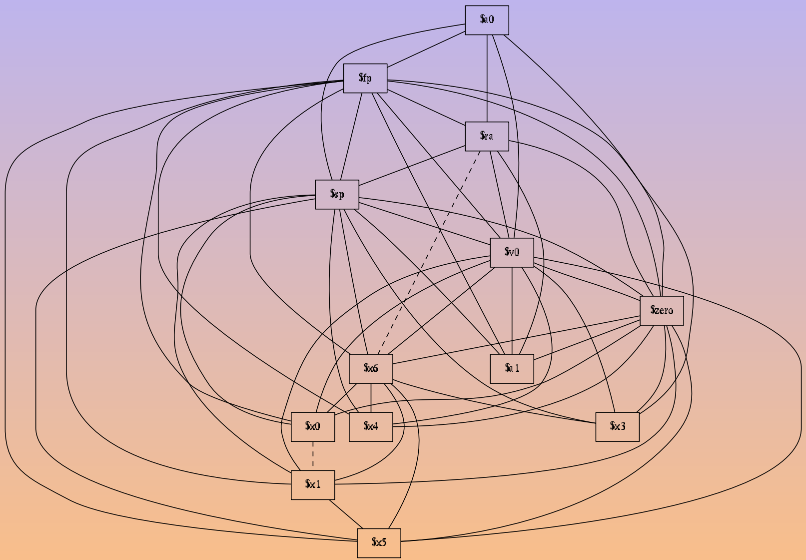



```
    r1 := 0
L1: r1 := r1 + 1
    r2 := r2 + r1
    r1 := r1 * 2
    if r1 < N goto L1
    return r2
```

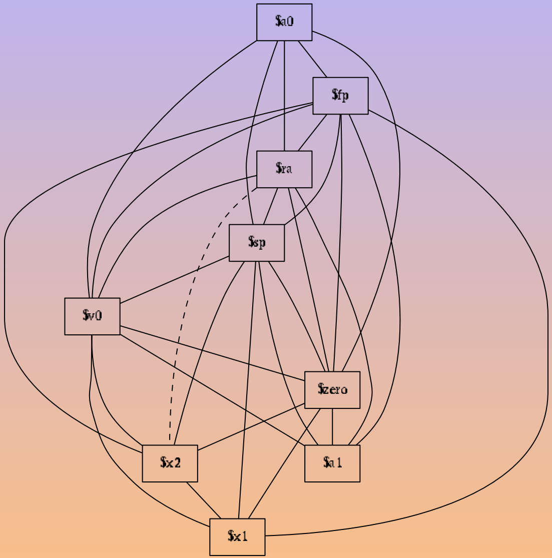
7's Interference Graph



7000's Interference Graph



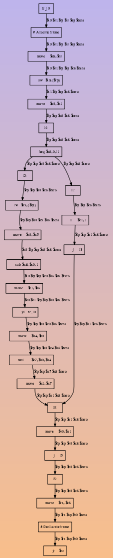
ors' Interference Graph



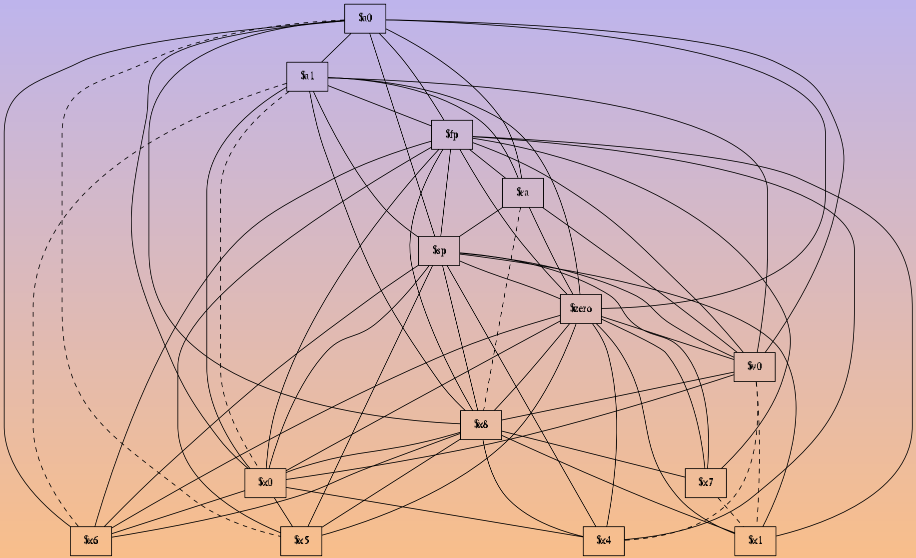
fact.tig

```
let function fact (n : int) : int =
      if n = 0 then
        1
      else
        n * fact (n - 1)
in
  fact (12)
end
```

fact's Liveness Graph



fact's Interference Graph



Bibliography I

Appel, A. W. (1998).
Modern Compiler Implementation in C, Java, ML.
Cambridge University Press.