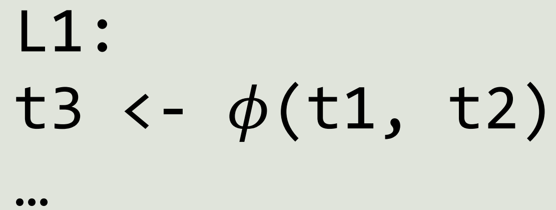
Register Allocation

PLUS IMPLEMENTING ϕ-FUNCTIONS AND METHOD INLINING

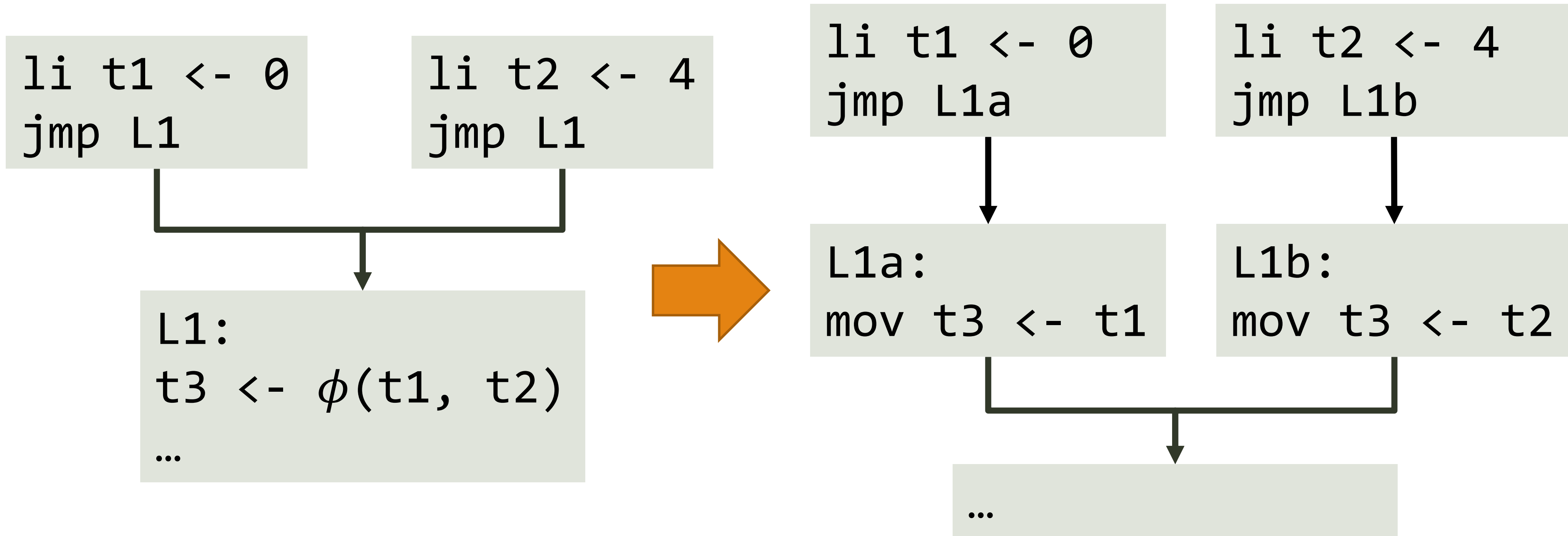
Implementing ϕ-Functions

```
li t1 <- 0
jmp L1
```

```
li t2 <- 4
jmp L1
```



Implementing ϕ-Functions



A Simple Algorithm

For each ϕ-function: `tx <- ` ϕ(ta, tb, …)

- For each inbound edge:
  - Walk the CFG from child to parent.
  - Let `ty` be the first assignment to one of `ta`, `tb`, …
  - Create a new node along the current edge containing `mov tx <- ty`

Why does this work?

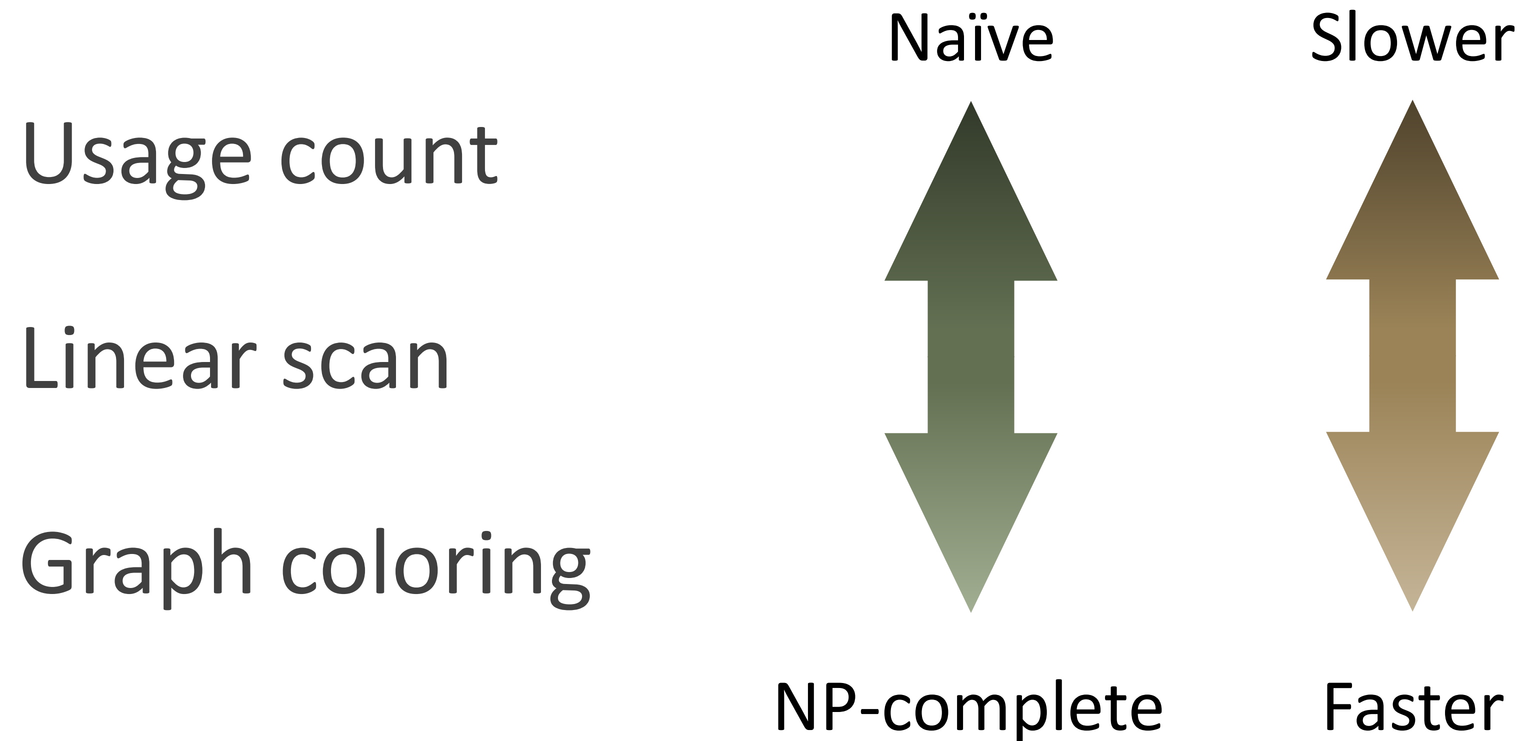
Register Allocation

The Register Allocation Problems

1. ***Register allocation***:
   - Which values to store in registers and which in memory?
2. ***Register assignment***:
   - Which registers to store them in?

Confusingly, “register allocation” may refer to doing both.

Register Allocation Algorithms



Usage Count Allocation

For each basic block:

1. Count uses for each variable.
2. Allocate registers to temporaries with highest usage.
3. Insert loads and stores at basic block boundaries.

Works best when IR is not in SSA form.

Live Intervals

Assume linear ordering of statements in IR.

A ***live interval*** $[i, j]$ for value v is an interval such that:

- For $0 < i' < i$, v is not live at i'.
- For $j < j' < n$, v is not live at j'.

Live Intervals

Assume linear ordering of statements in IR.

A ***live interval*** $[i, j]$ for value v is an interval such that:

- For $0 < i' < i$, v is not live at i'.
- For $j < j' < n$, v is not live at j'.

Not equivalent to [first use, last use]. Why?

In general, may contain subintervals in which v is not live.

A Previous Example (Before Optimization)

```
li t1 <- 1
boxi t2 <- t1
L1:
t19 <- φ(t2, t18)
t20 <- φ(r0, t12)
unboxi t3 <- t19
li t4 <- 10
boxi t5 <- t4
unboxi t6 <- t5
ble t6 t3 L2
```

```
unboxi t7 <- t20
li t8 <- 2
boxi t9 <- t8
unboxi t10 <- t9
mul t11 <- t7 t10
boxi t12 <- t11
unboxi t13 <- t2
li t14 <- 1
boxi t15 <- t14
unboxi t16 <- t15
```

```
add t17 <- t13 t16
boxi t18 <- t17
jmp L1
L2:
mov r1 <- t12
```

A Previous Example

```
  li t1 <- 1
L1:
  t19 <- ϕ(t1, t17)
  t20 <- ϕ(r0, t12)
  li t4 <- 10
  ble t4 t19 L2
  unboxi t7 <- t20
  li t8 <- 2
```

```
  mul t11 <- t7 t8
  boxi t12 <- t11
  li t14 <- 1
  add t17 <- t19 t14
  jmp L1
L2:
  mov r1 <- t20
```

A Previous Example

```
  li t1 <- 1
  mov t19 <- t1
  mov t20 <- r0
L1:
  li t4 <- 10
  ble t4 t19 L2
  ... t20
```

```
  mul t11 <- t7 t8
  boxi t12 <- t11
  li t14 <- 1
  add t17 <- t19 t14
  mov t19 <- t17
  mov t20 <- t12
  jmp L1
L2:
  mov r1 <- t20
```

Implement ϕ-functions

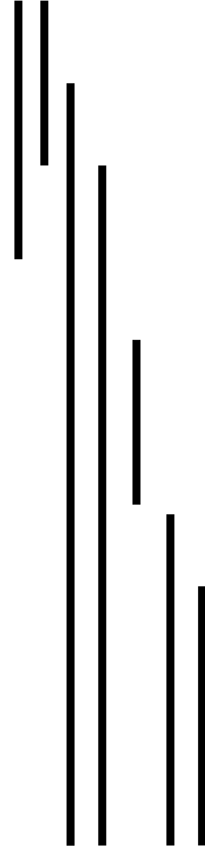
A Previous Example

```
  li t1 <- 1
  mov t19 <- t1
  mov t20 <- r0
L1:
  li t4 <- 10
  ble t4 t19 L2
  unboxi t7 <- t20
  li t8 <- 2
```

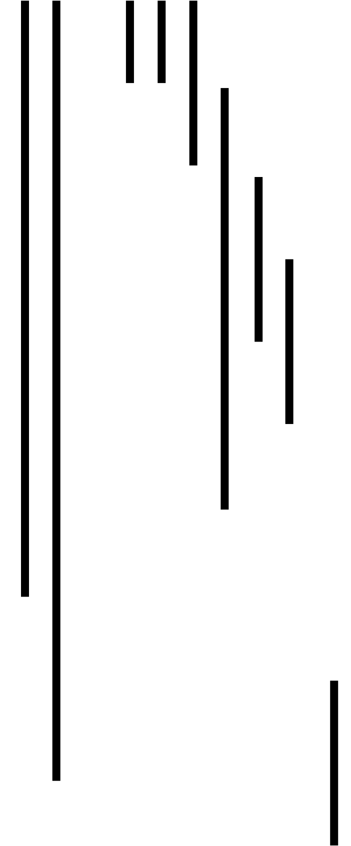
```
  mul t11 <- t7 t8
  boxi t12 <- t11
  li t14 <- 1
  add t17 <- t19 t14
  mov t19 <- t17
  mov t20 <- t12
  jmp L1
L2:
  mov r1 <- t20
```

Live Intervals Example

```
  li t1 <- 1
  mov t19 <- t1
  mov t20 <- r0
L1:
  li t4 <- 10
  ble t4 t19 L2
  unboxi t7 <- t20
  li t8 <- 2
```



```
  mul t11 <- t7 t8
  boxi t12 <- t11
  li t14 <- 1
  add t17 <- t19 t14
  mov t19 <- t17
  mov t20 <- t12
  jmp L1
L2:
  mov r1 <- t20
```



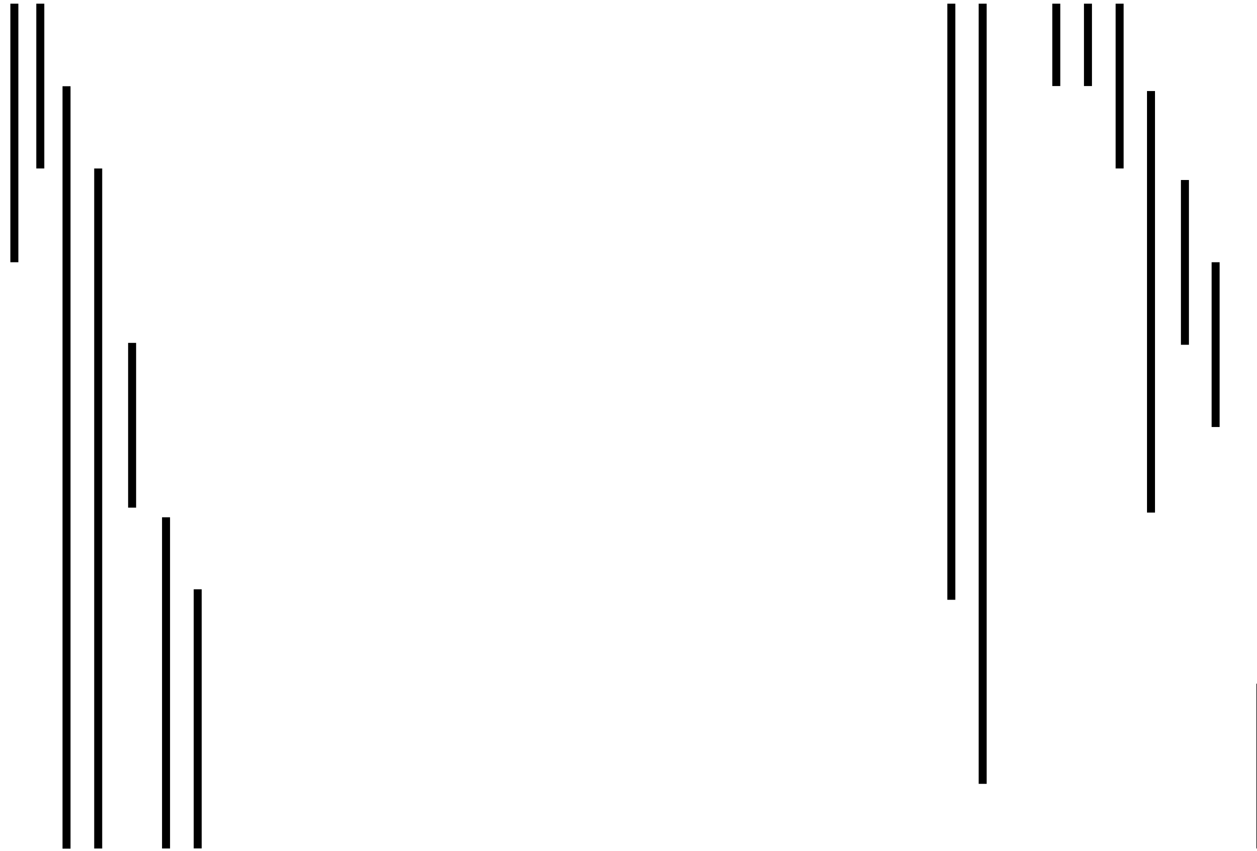
Interval Interference

Two intervals ***interfere*** if they overlap.

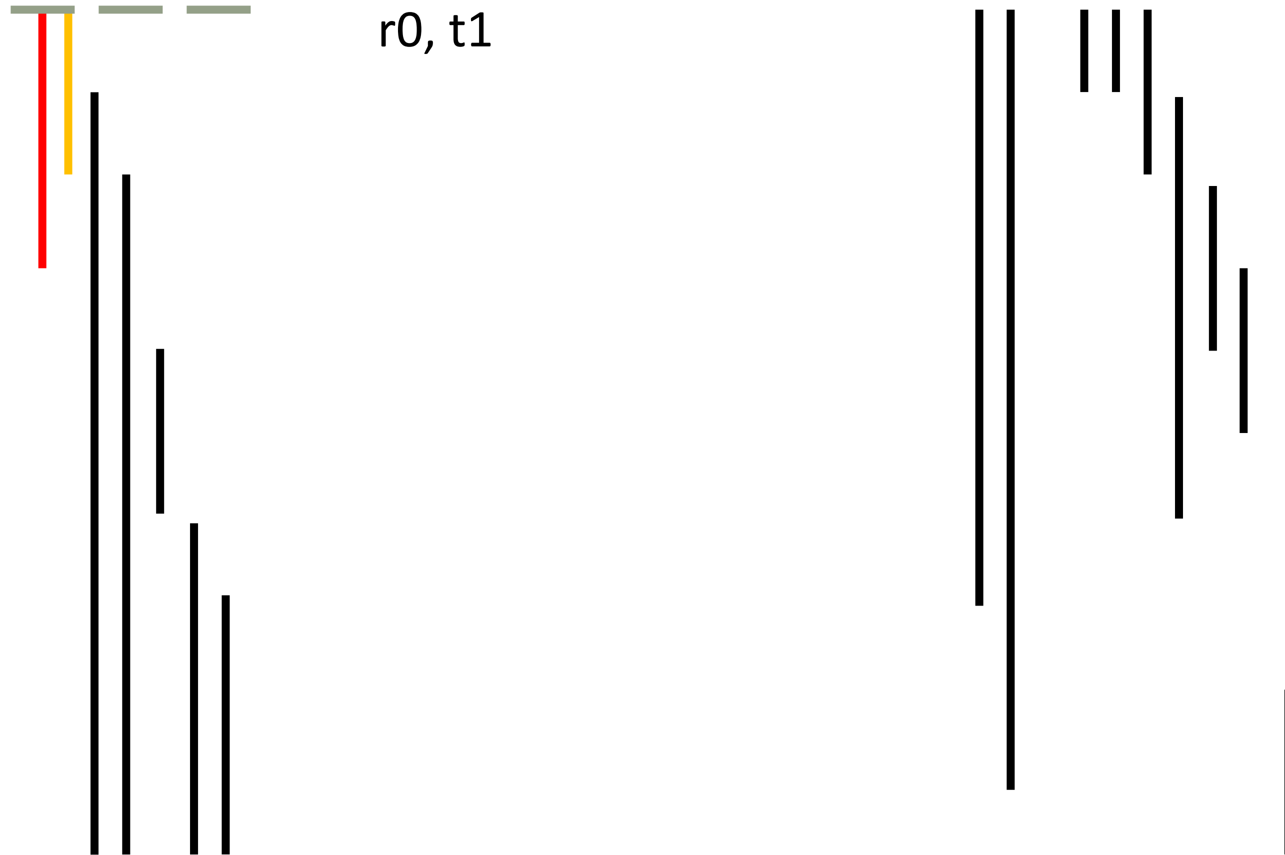
- Given $[i_1, j_1]$ and $[i_2, j_2]$, $i_1 \leq j_2$ and $i_2 \leq j_1$.

Interfering intervals may not be assigned the same register.

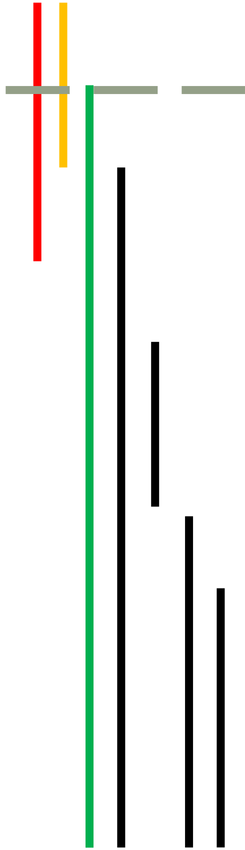
Linear Scan



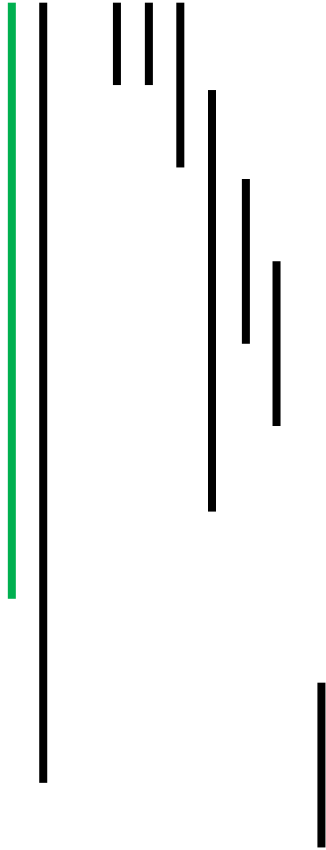
Linear Scan



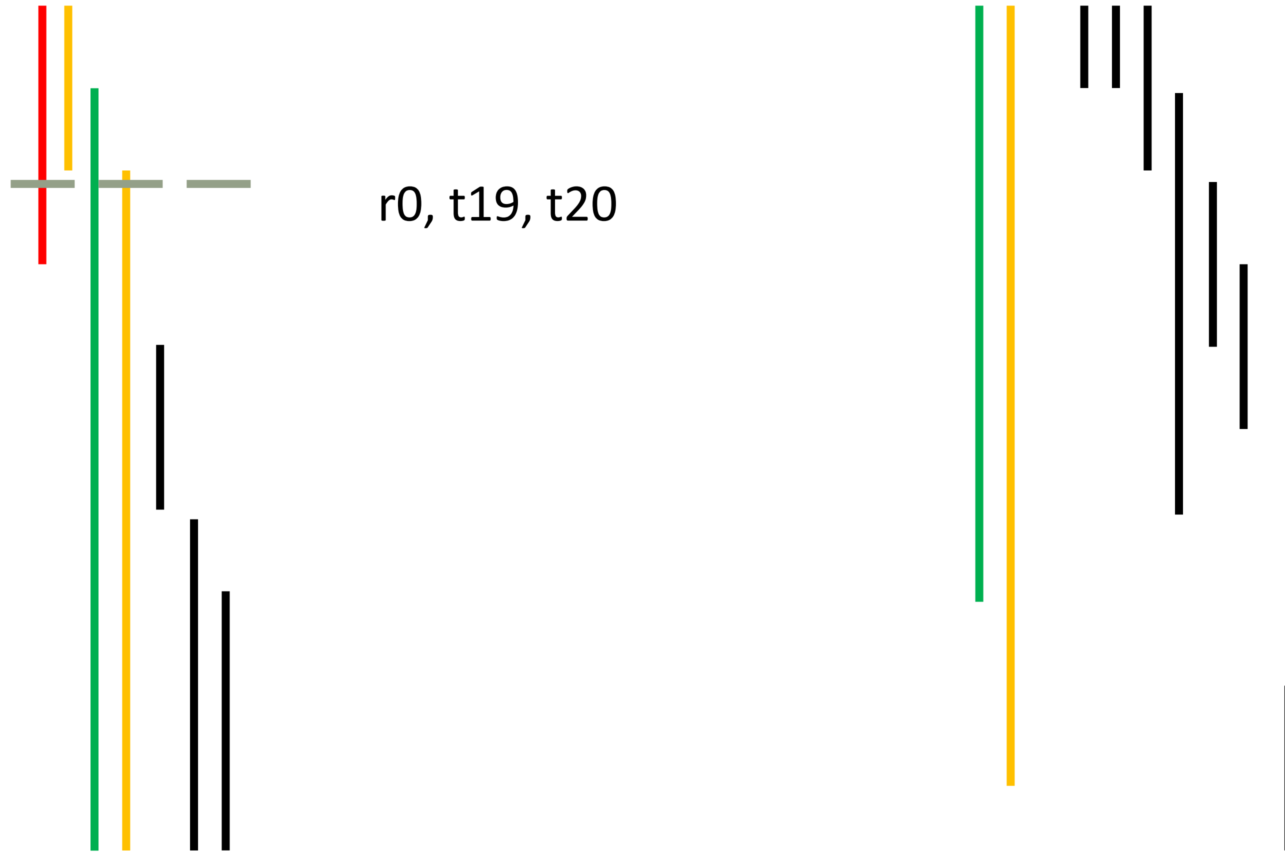
Linear Scan



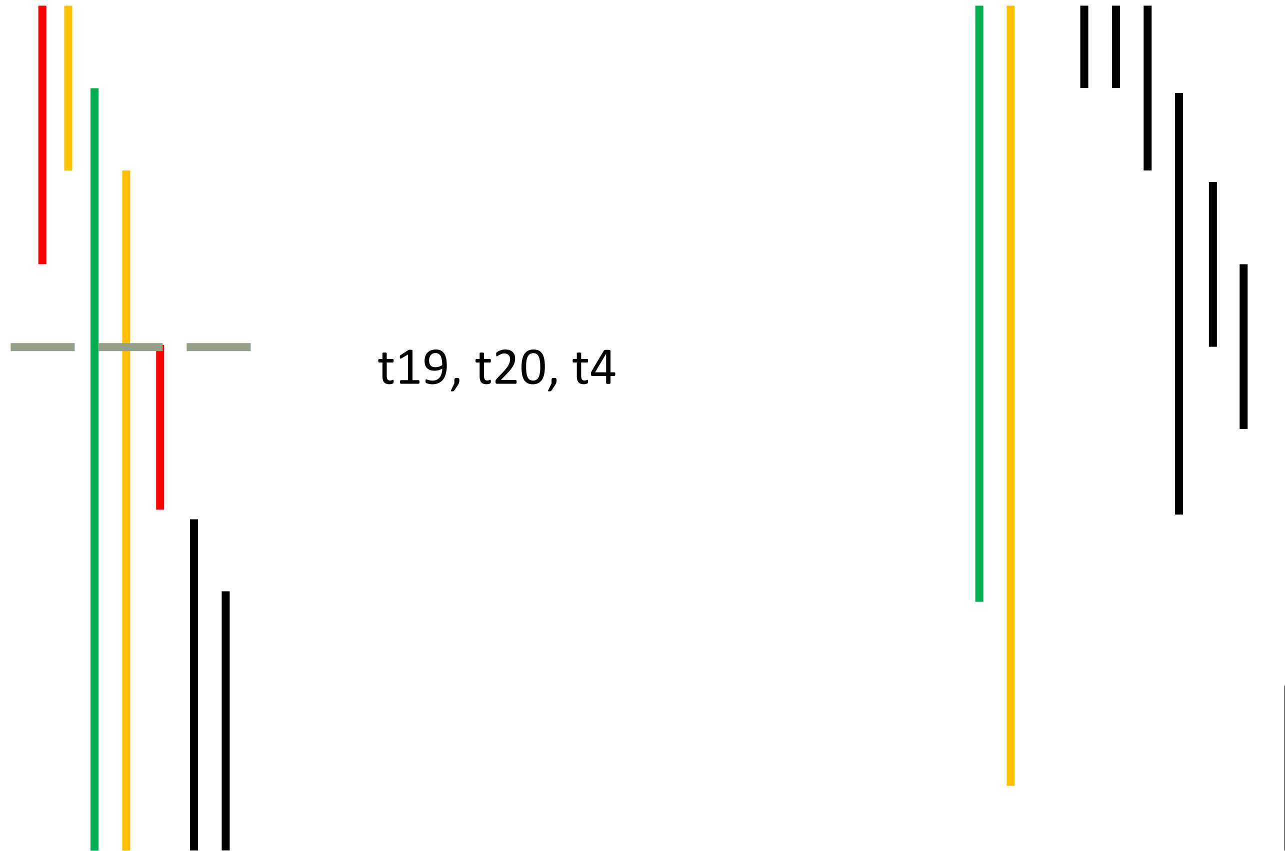
r0, t1, t19



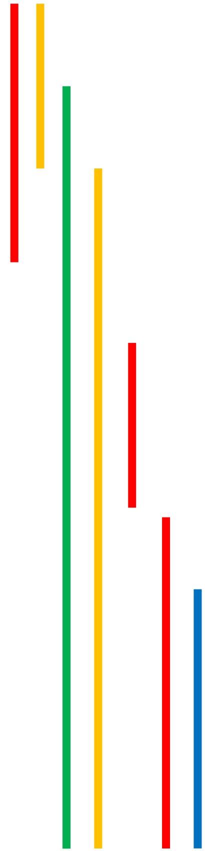
Linear Scan

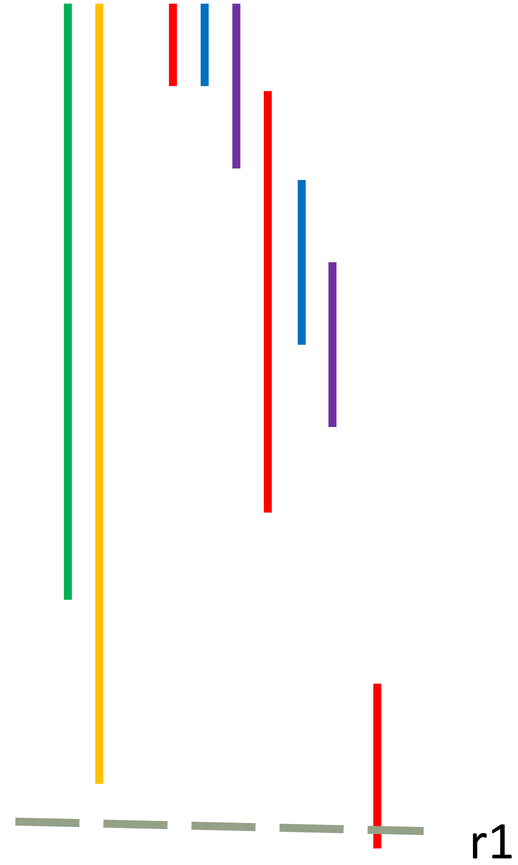


Linear Scan



Linear Scan



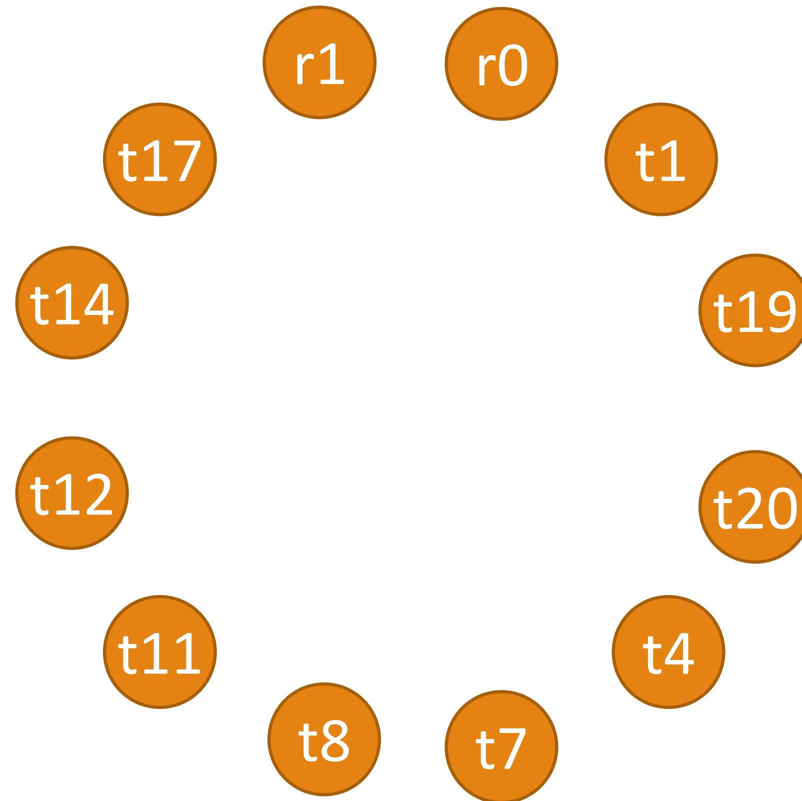


Linear Scan Register Allocation

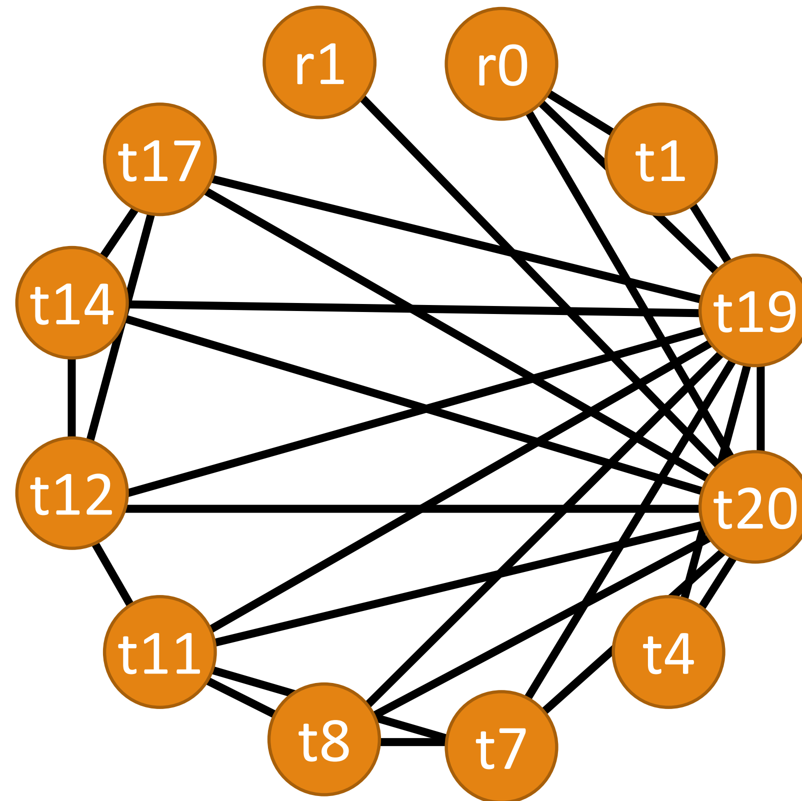
Scan IR from top to bottom.

- Keep track of currently active live intervals.
- When interval becomes inactive free its register.
- When interval becomes active assign it a free register.

Graph Coloring

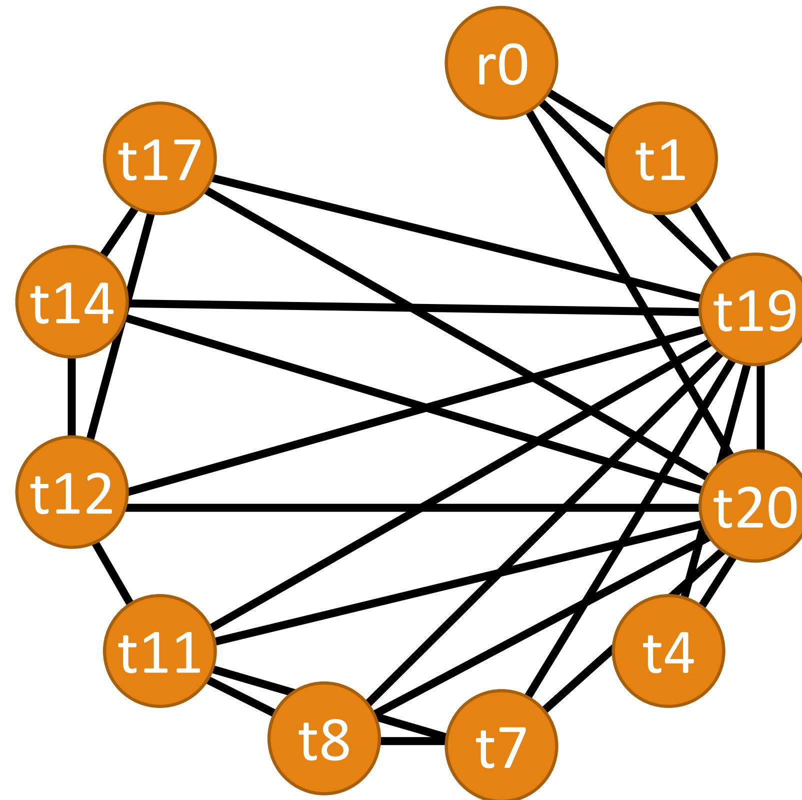


Graph Coloring



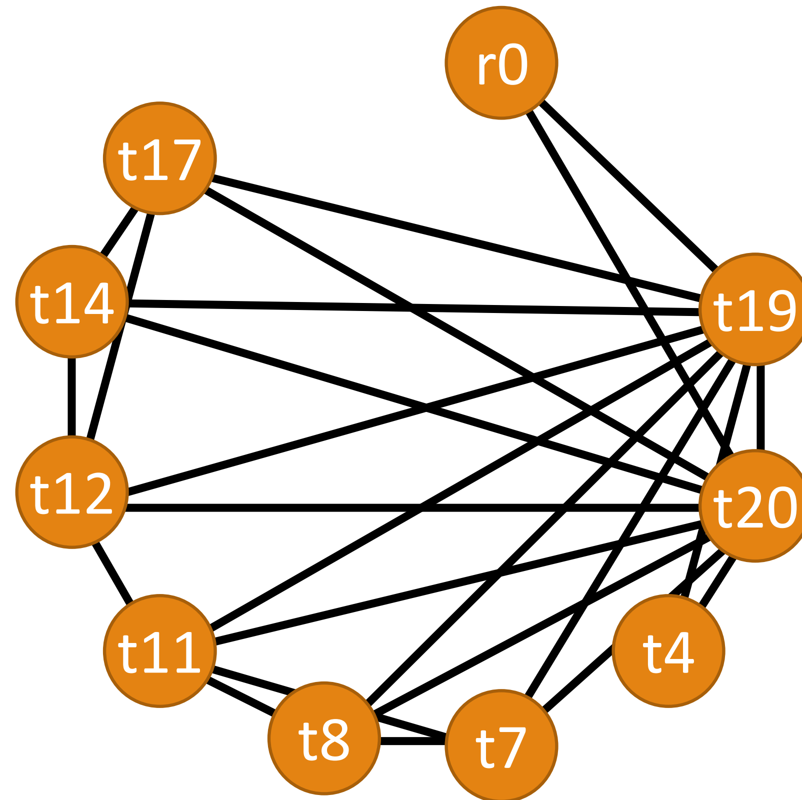
Graph Coloring

r1



Graph Coloring

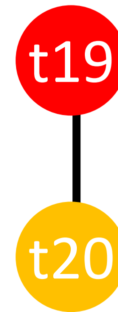
r1 t1



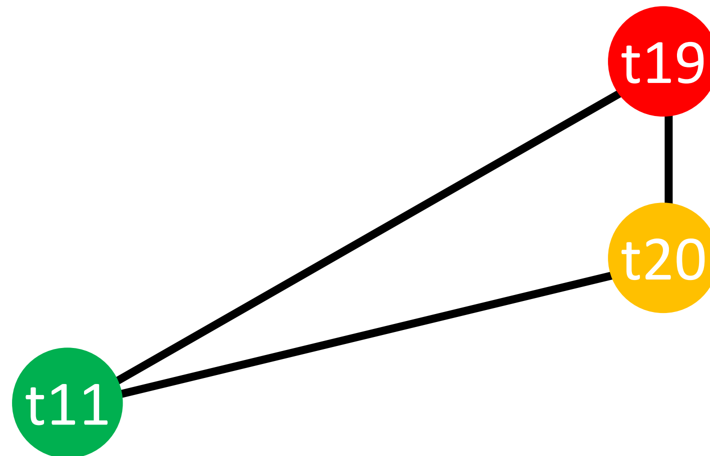
Graph Coloring

Graph Coloring

Graph Coloring

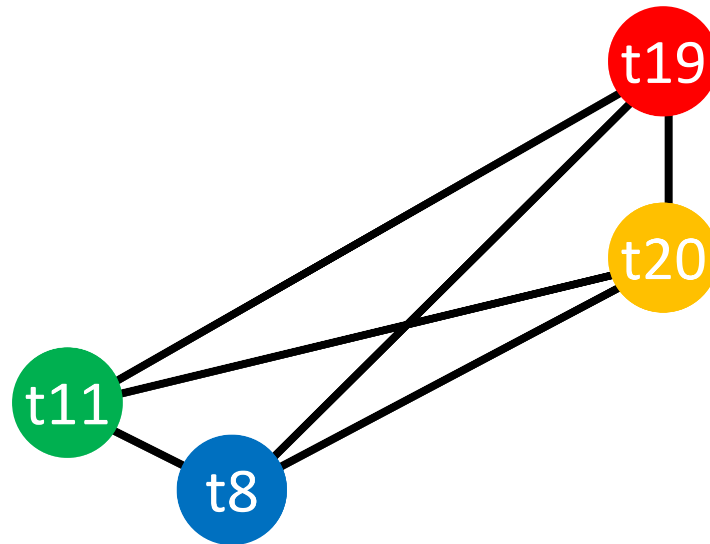


Graph Coloring



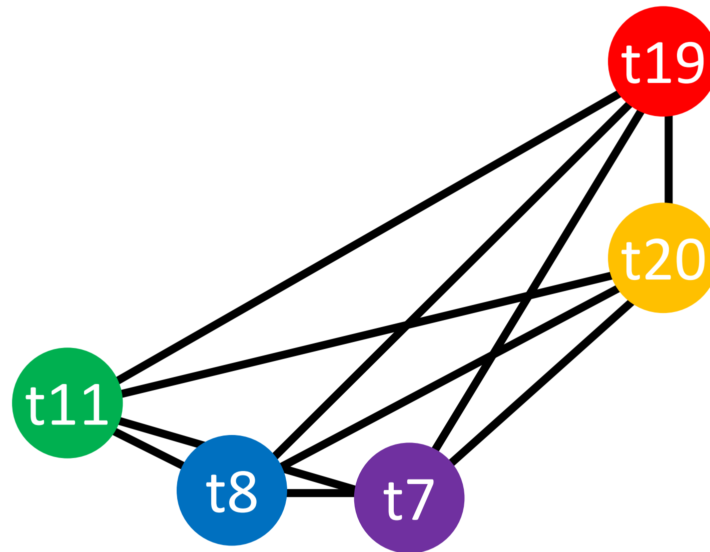
Graph Coloring

r1 t1 r0 t4 t17 t14 t12 t7



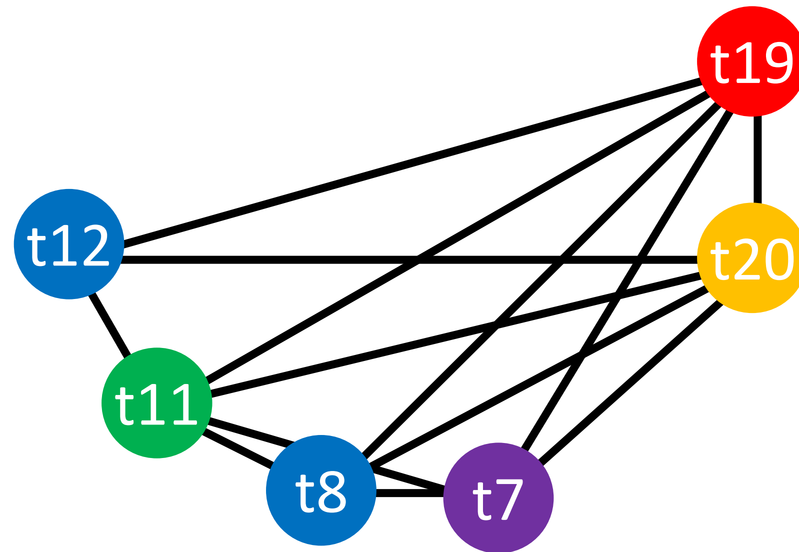
Graph Coloring

r1 t1 r0 t4 t17 t14 t12



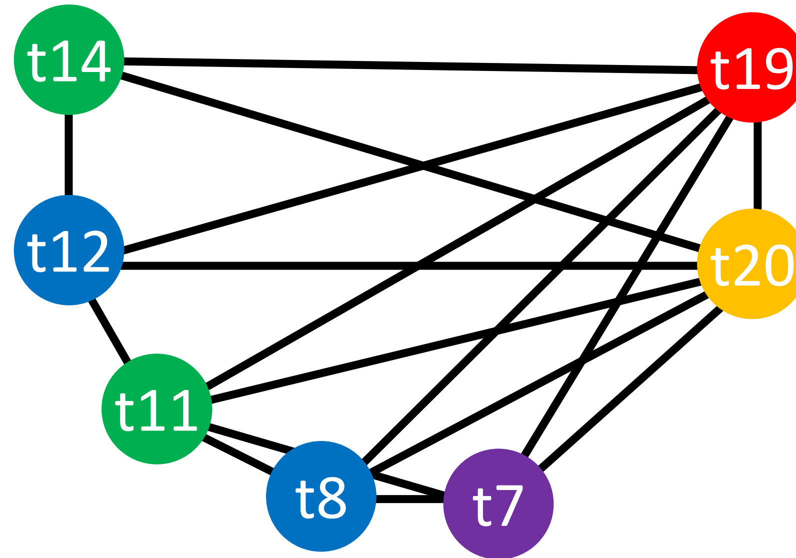
Graph Coloring

r1 t1 r0 t4 t17 t14



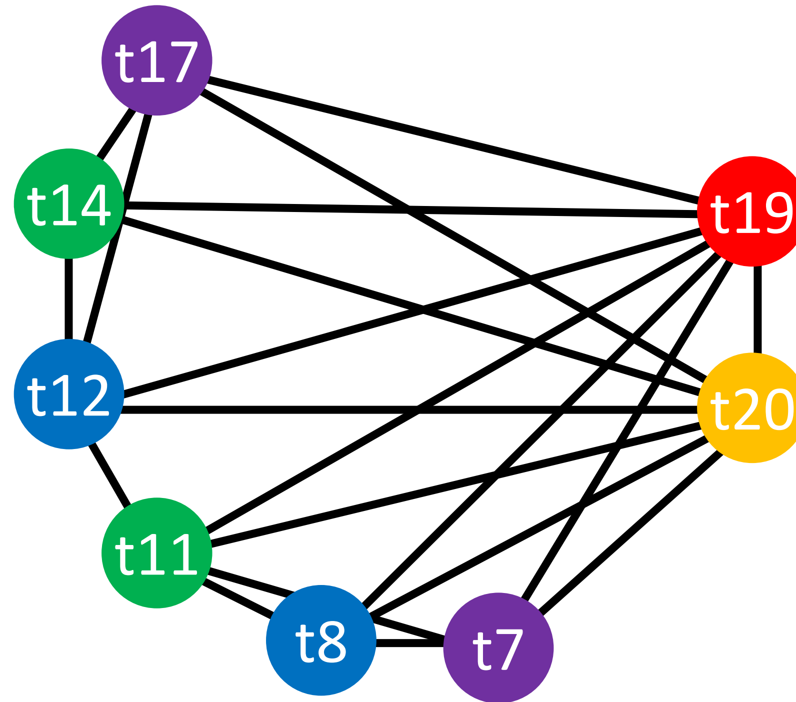
Graph Coloring

r1 t1 r0 t4 t17



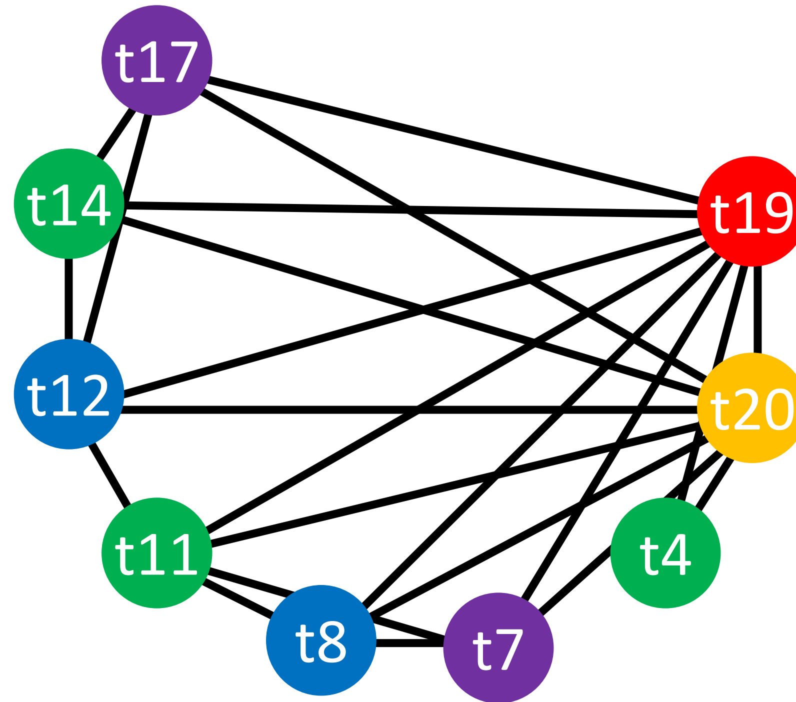
Graph Coloring

r1 t1 r0 t4



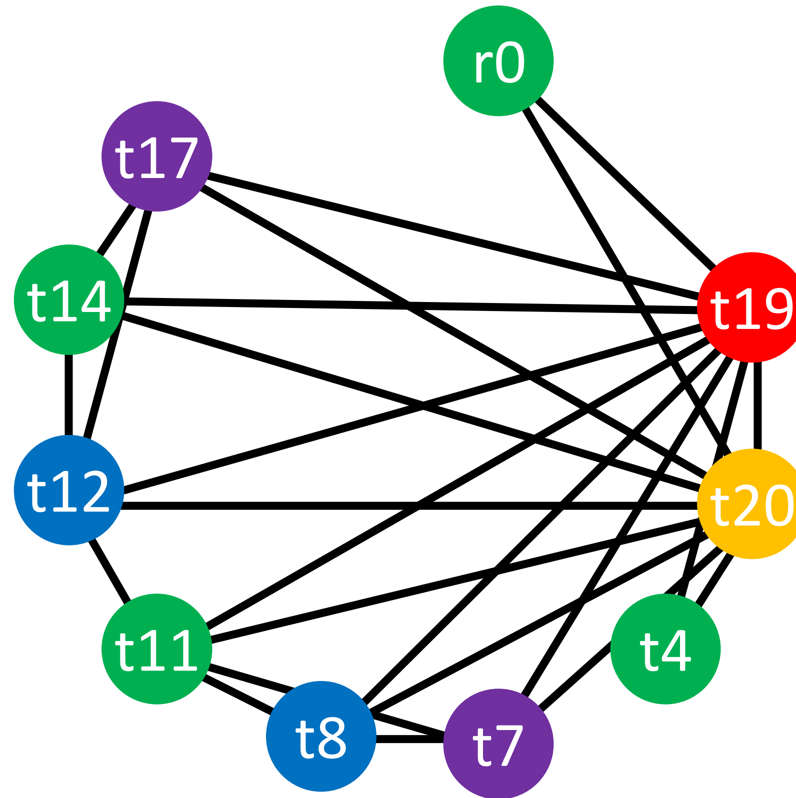
Graph Coloring

r1 t1 r0



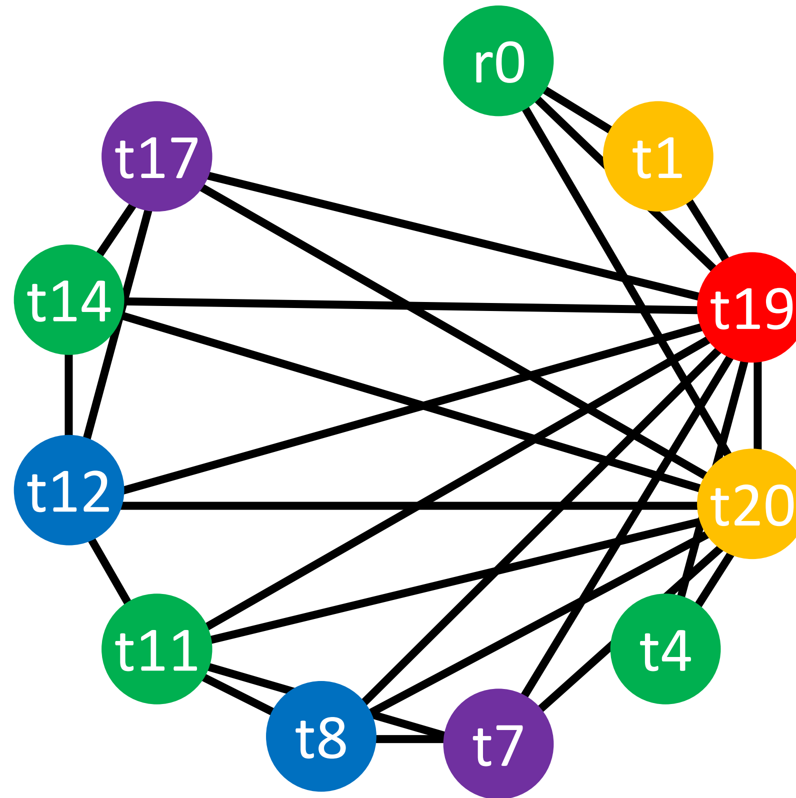
Graph Coloring

r1 t1

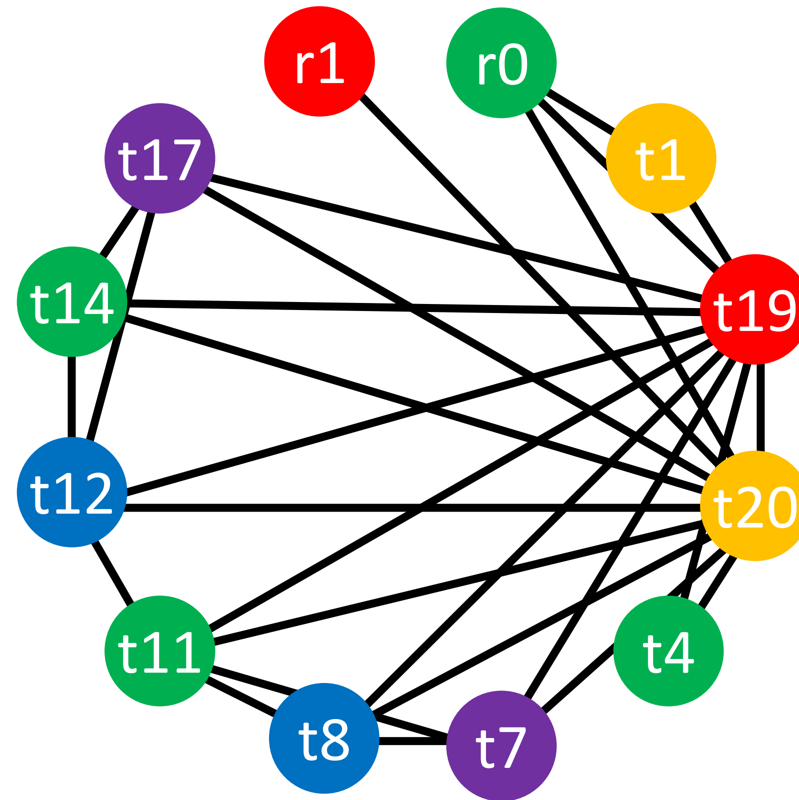


Graph Coloring

r1



Graph Coloring



Graph Coloring Register Allocation

Construct graph:

- Vertex for each location
- Edge between simultaneously live locations

While graph is not empty:

- Remove node with fewest edges and push on stack.

While stack is not empty:

- Pop node, reinsert into graph, and assign color.

What if we need more colors than we have registers?

“Spilling” Values: Linear Scan

More active live intervals than registers.

Spill value in interval with latest end-point.

- Insert store after definition and loads before every access.
  - Note: Live interval of new load ends after first use.
- Remove value from active list and continue.

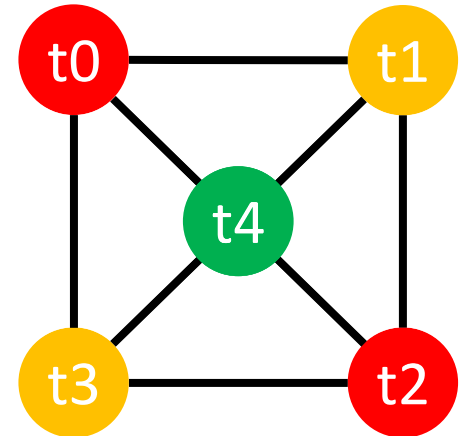
Alternatively, spill value in interval with fewest uses.

“Spilling” Values: Graph Coloring

Adding a node with no available color.

- Not same as node with at least *N* edges.

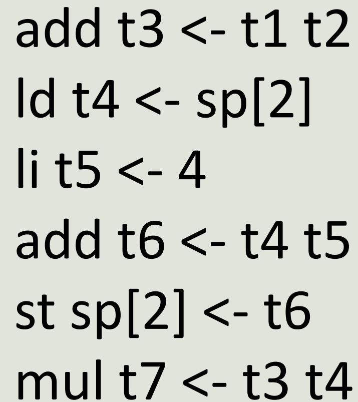
Spill node and ***restart*** graph coloring.

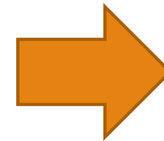


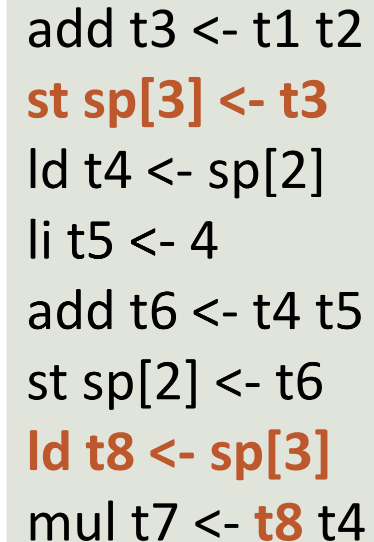
“Spilling” Values: Graph Coloring

Alternatively, attempt to “split” node:

- Find interfering range contained in this one such that this value is not used in the other range.
- Store this value before the other range and load it after.







“Spilling” Values: Graph Coloring

Alternatively, attempt to “split” node:

- Find interfering range contained in this one such that this value is not used in the other range.
- Store this value before the other range and load it after.

Caution: Split must be CFG-aware.

- Value must be stored and loaded on ***all*** paths.

Registers in Calling Conventions

If you pass parameters or return values by register:

- Pre-color intervals / nodes with appropriate register.

Callee-saved registers:

- Treat method entry as definition of all callee-saved registers.
- Treat method exit as use of all callee-saved registers.

Caller-saved registers:

- Insert store/load for all live caller-saved registers.

Performance Considerations

For small functions:

- Graph coloring and linear scan perform similarly.

For large functions:

- Graph coloring produces faster code (~10%, YMMV).
- Linear scan runs faster during compilation – ideal for JIT.

For human developers:

- Linear scan is simpler to write and debug.

Method Inlining

Inlining Intuition

Replace `foo(x,y,z)` with the body of `foo`.

- No function call overhead.
  - Stack updates, register saving…
- Optimize inlined body as part of caller.
  - Unboxing, register allocation, common sub-expressions,…

Inlining Limitations

Code bloat:

- `foo(1); foo(2)` becomes two complete copies of foo.
- More instructions = greater chance of I-cache miss.
- Typically, limit size of methods to inline.

Recursive calls:

- How deep do you go?
- May require ***solving the halting problem***.

Receiver Class Analysis

Only inline a method if we know which method to inline!

How?

Receiver Class Analysis

Only inline a method if we know which method to inline!

Data-flow analysis to the rescue!

Direction: ?

Values: ?

- **Init** ?

Meet operator:

- ?

Transfer functions:

- ?

Receiver Class Analysis

Only inline a method if we know which method to inline!

Data-flow analysis: determine dynamic type at call site.

Direction: forward

Values: type trees

- **Init** parameters to static types.

Meet operator:

- Common superclass.

Transfer functions:

- new T → T
- Method calls → hierarchy rooted at static return type.
- …

Inlining in Practice

For each method call:

- If inlining depth is too deep: skip.
- If call site allows overrides: cannot inline*.
- If method body is too long: should not inline.
- Insert method body in place of call site:
  - Update temporaries to avoid conflicts.
  - Insert loads, stores, and moves to align parameters and return value.

Inlining in Practice

For each method call:

- If inlining depth is too deep: skip.
- If call site allows overrides: cannot inline*.
- If method body is too long: should not inline.
- Insert method body in place of call site:
  - Update temporaries to avoid conflicts.
  - Insert loads, stores, and moves to align parameters and return value.

*You could insert a case expression with multiple inlines or inline the common case while calling the rest dynamically…

Backup

Live Interval with Gap

```
fibh(a:Int, b:Int, n:Int) : Int {
  let c:Int <- a+b in
  if 1 < n then
    fibh(b, c, n-1)
  else
    c
  fi
};
```

Live Interval with Gap

```
ld t1 <- sp[1]
unboxi t2 <- t1
ld t3 <- sp[2]
unboxi t4 <- t3
add t5 <- t2 t4
boxi t6 <- t5
li t7 <- 1
ld t8 <- sp[3]
unboxi t9 <- t8
ble t9 t7 L1
ld t10 <- sp[0]
```

```
li t11 <- 4
add sp <- sp t11
st sp[0] <- t11
st sp[1] <- t3
st sp[2] <- t6
sub t12 <- t9 t7
boxi t13 <- t12
st sp[3] <- t13
ld t14 <- t11[0]
ld t15 <- t14[6]
call t15
```

```
li t16 <- 4
sub sp <- sp t16
mov t17 <- r1
jmp L2
L1:
mov t18 <- t3
L2:
t19 <- ϕ(t17, t18)
mov r1 <- t19
ret
```

Live Interval with Gap

```
ld t1 <- sp[1]
unboxi t2 <- t1
ld t3 <- sp[2]
unboxi t4 <- t3
add t5 <- t2 t4
boxi t6 <- t5
li t7 <- 1
ld t8 <- sp[3]
unboxi t9 <- t8
ble t9 t7 L1
--------------------
ld t10 <- sp[0]
```

```
li t11 <- 4
add sp <- sp t11
st sp[0] <- t11
st sp[1] <- t3
st sp[2] <- t6
sub t12 <- t9 t7
boxi t13 <- t12
st sp[3] <- t13
ld t14 <- t11[0]
ld t15 <- t14[6]
call t15
```

```
li t16 <- 4
sub sp <- sp t16
mov t17 <- r1
jmp L2
--------------------
L1:
mov t18 <- t3
--------------------
L2:
t19 <- ϕ(t17, t18)
mov r1 <- t19
ret
```

A Running Example