Register Allocation

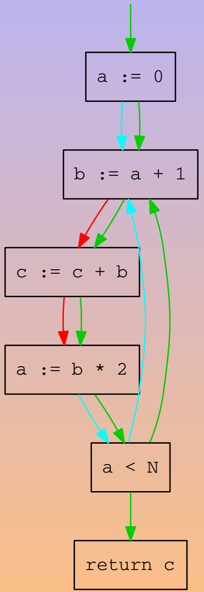
Akim Demaille Étienne Renault Roland Levillain
first.last@lrde.epita.fr

EPITA — École Pour l'Informatique et les Techniques Avancées

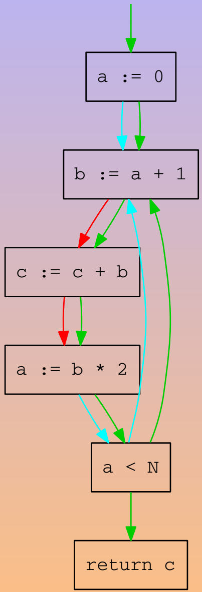
May 19, 2018

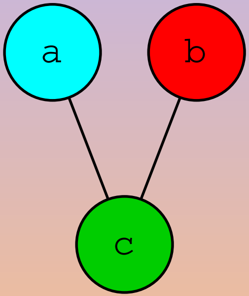
Register Allocation

Interference Graph

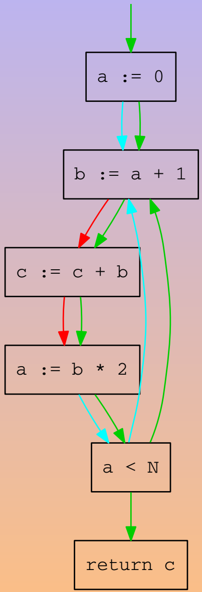


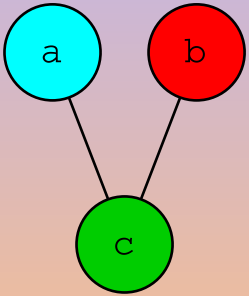
Interference Graph

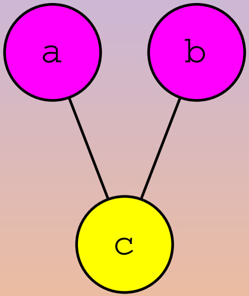




Interference Graph



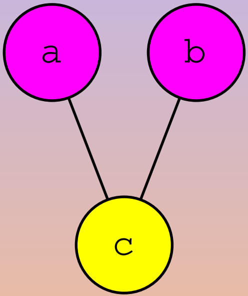




Register Allocation

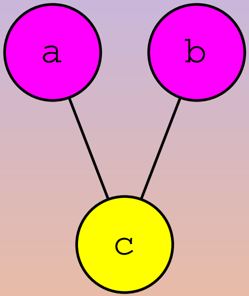
```
    a := 0
L1: b := a + 1
    c := c + b
    a := b * 2
    if a < N goto L1
    return c
```

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



Register Allocation

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

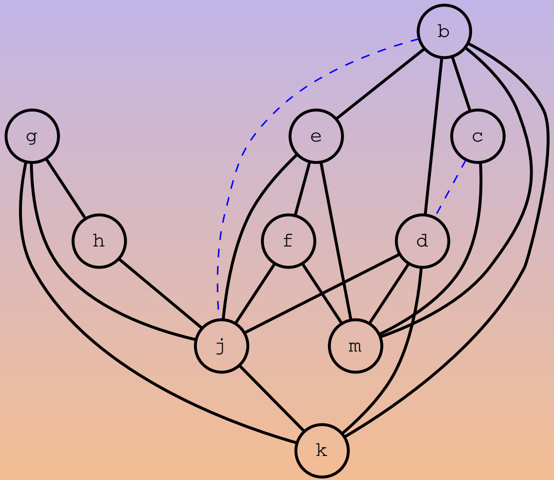



```
    r1 := 0
L1: r1 := r1 + 1
    r2 := r2 + r1
    r1 := r1 * 2
    if r1 < N goto L1
    return r2
```

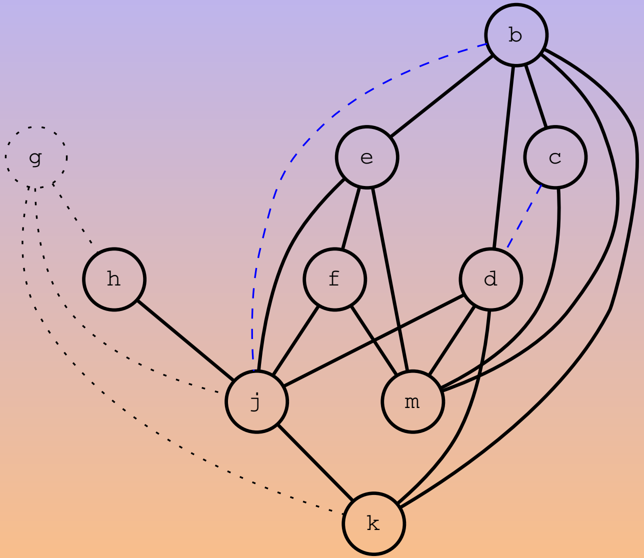
Interference Graph [Appel, 1998]

Four registers: r1, r2, r3, r4.

```
  live in: k j
    g := [j + 12]
    h := k - 1
    f := g * h
    e := [j + 8]
    m := [j + 16]
    b := [f]
    c := e + 8
    d := c
    k := m + 4
    j := b
  live out: d k j
```

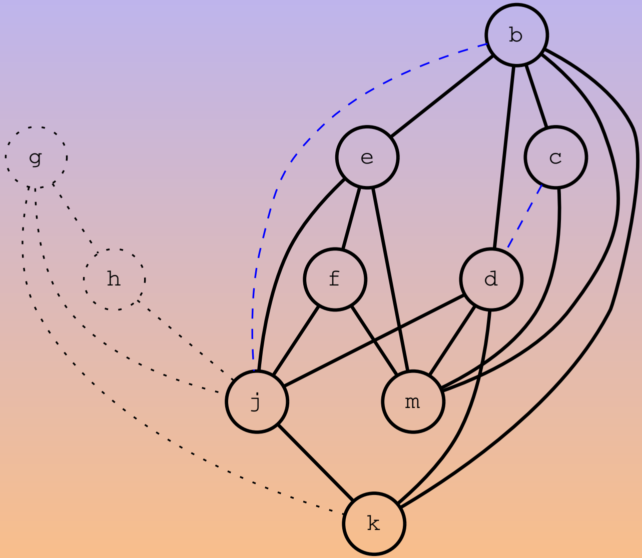


Interference Graph: Simplify 0



g

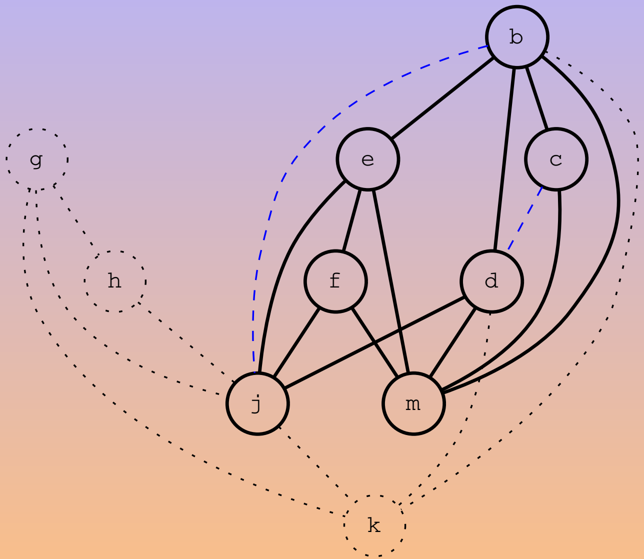
Interference Graph: Simplify 1



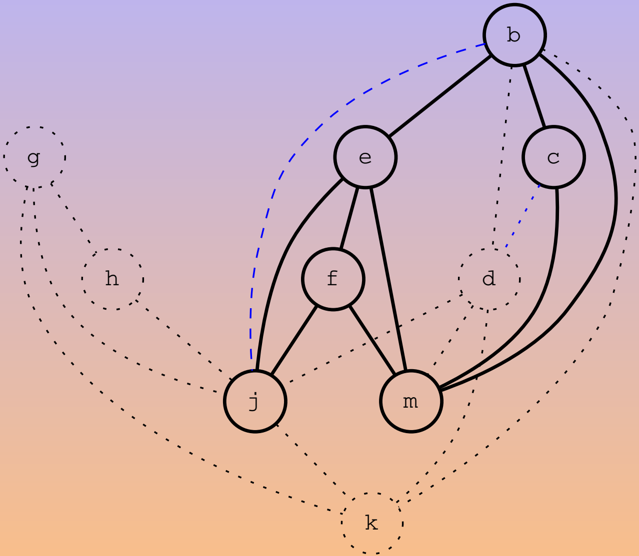
h
g

Interference Graph: Simplify 2

k
h
g



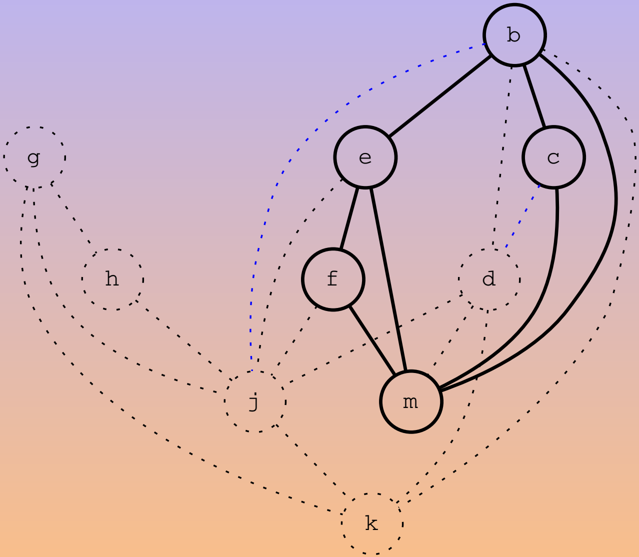
Interference Graph: Simplify 3



d
k
h
g

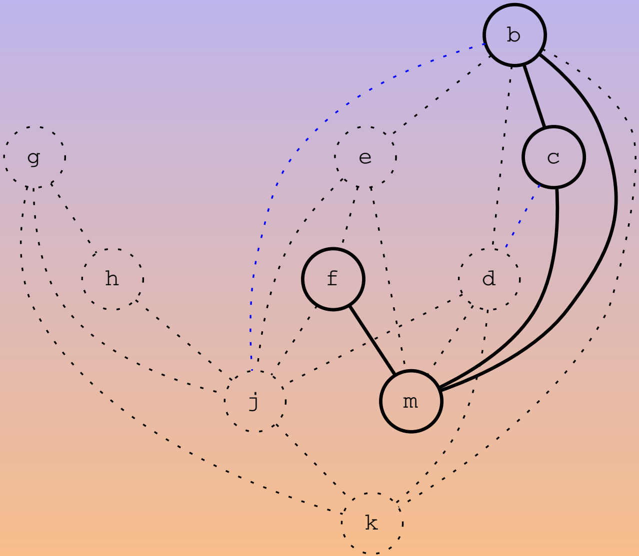
Interference Graph: Simplify 4

j
d
k
h
g



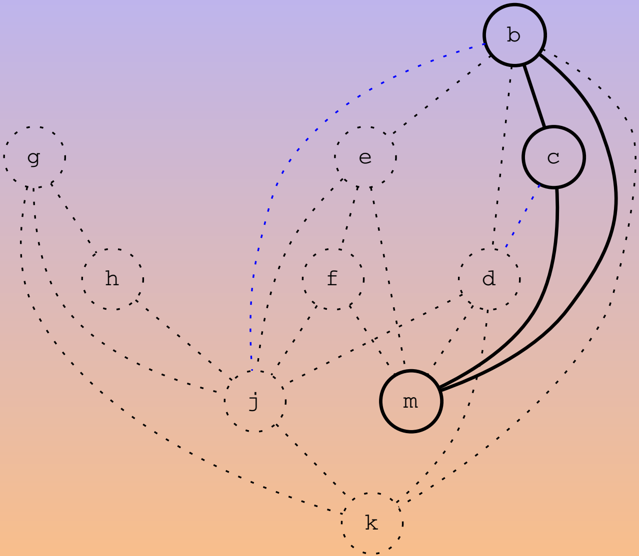
Interference Graph: Simplify 5

e
j
d
k
h
g



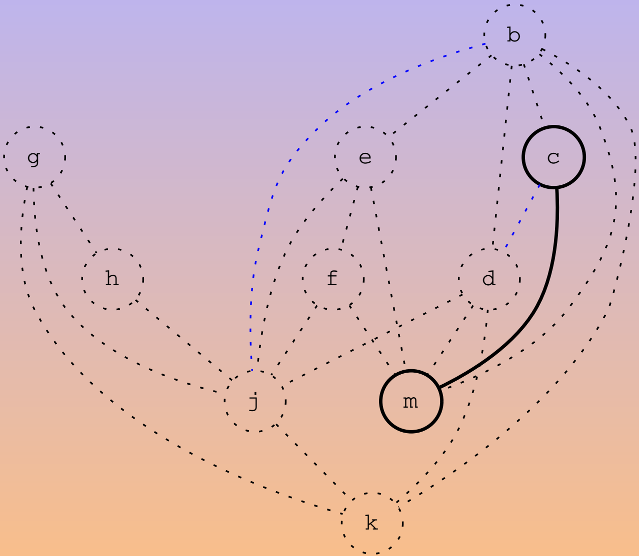
Interference Graph: Simplify 6

f
e
j
d
k
h
g



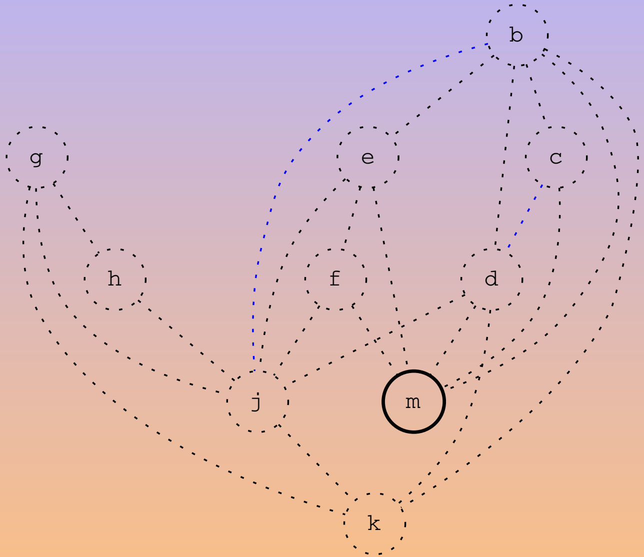
Interference Graph: Simplify 7

b
f
e
j
d
k
h
g



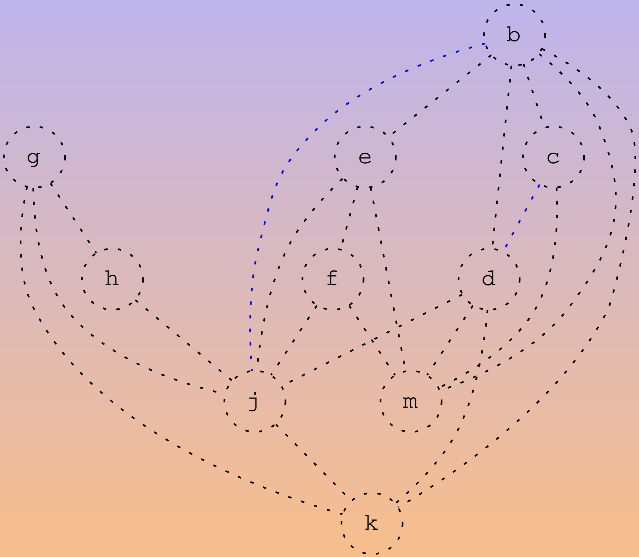
Interference Graph: Simplify 8

c
b
f
e
j
d
k
h
g



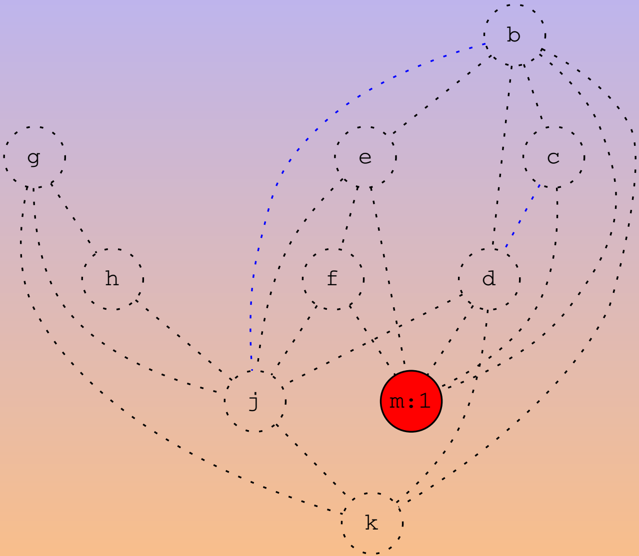
Interference Graph: Simplify 9

m
c
b
f
e
j
d
k
h
g



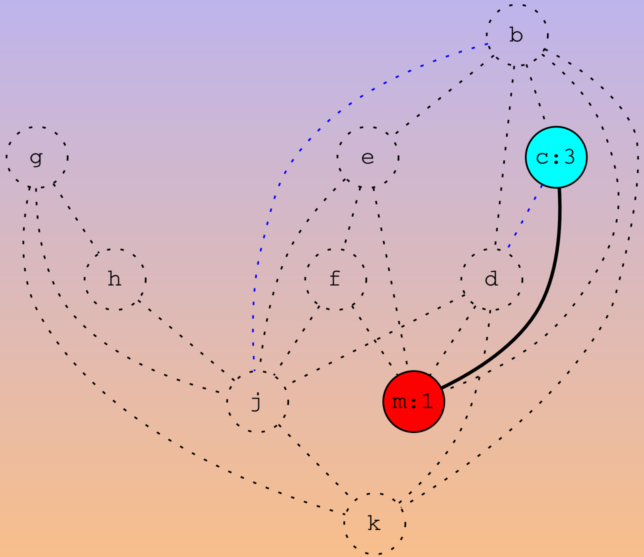
Interference Graph: Color 9

m
c
b
f
e
j
d
k
h
g



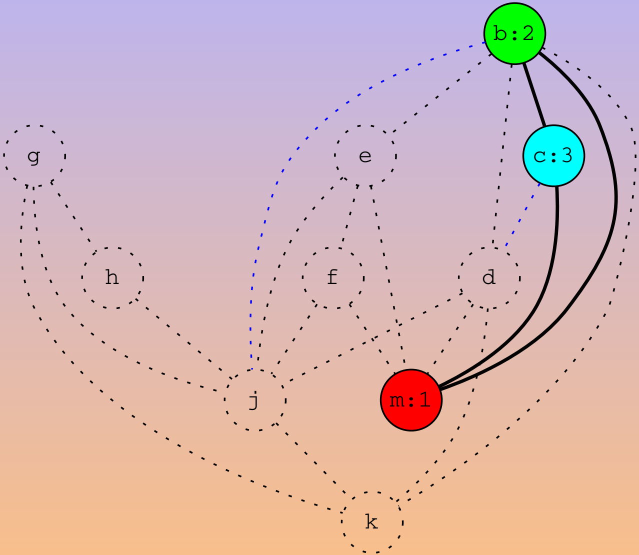
Interference Graph: Color 8

c
b
f
e
j
d
k
h
g



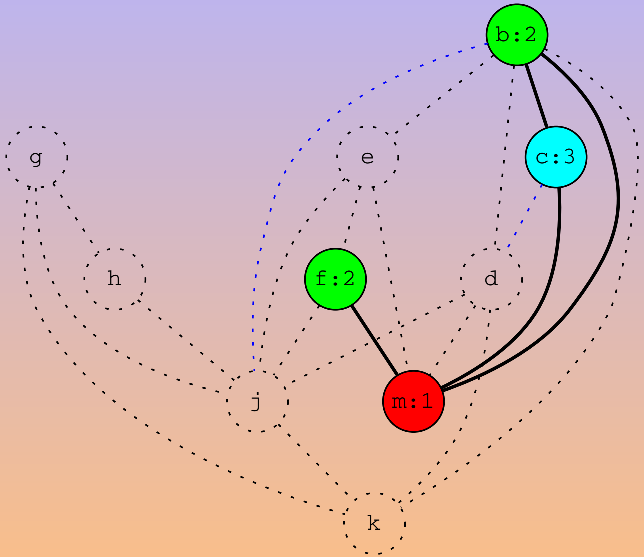
Interference Graph: Color 7

b
f
e
j
d
k
h
g



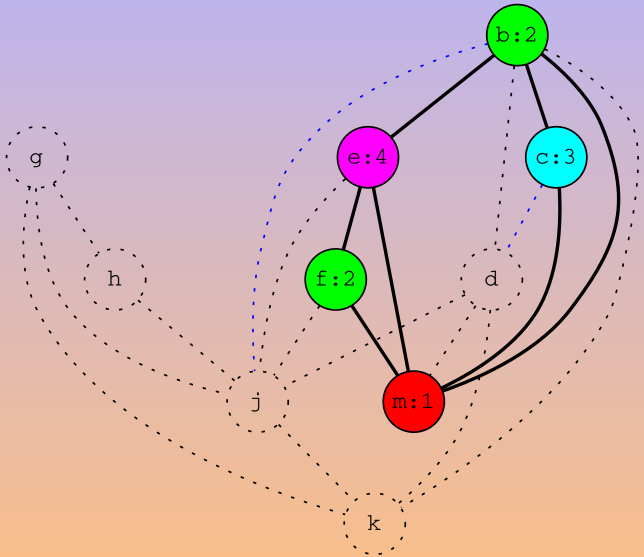
Interference Graph: Color 6

f
e
j
d
k
h
g



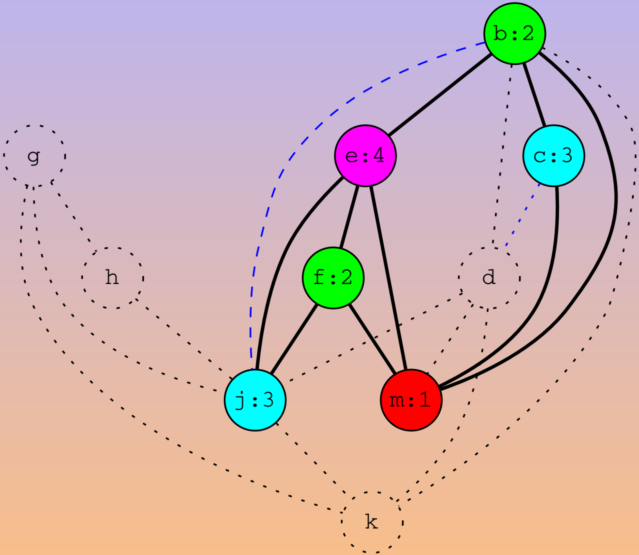
Interference Graph: Color 5

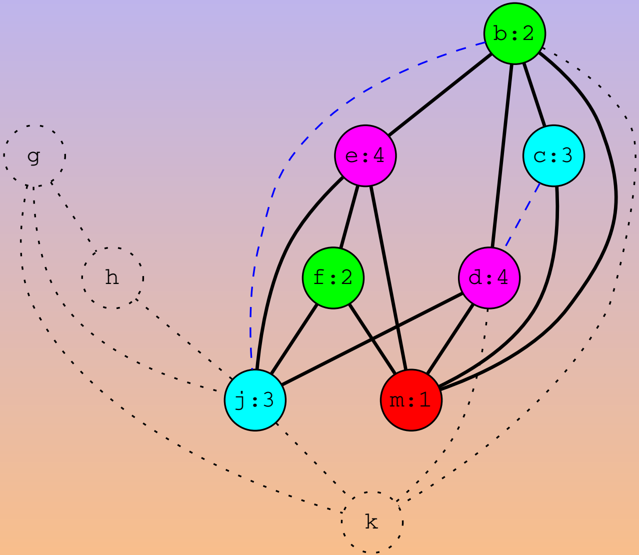
e
j
d
k
h
g



Interference Graph: Color 4

j
d
k
h
g

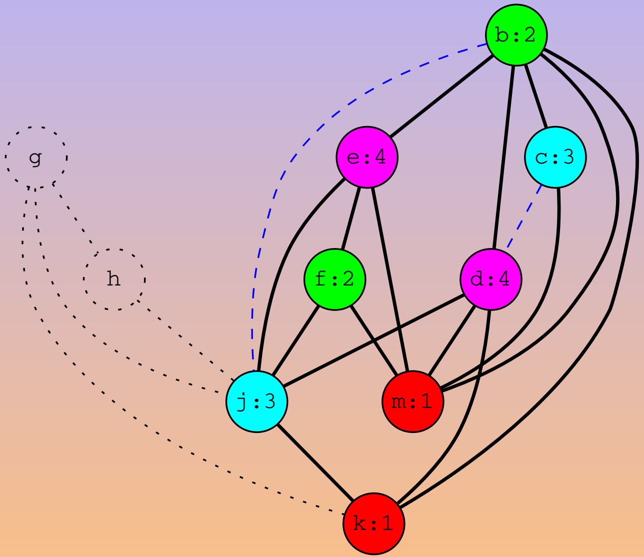




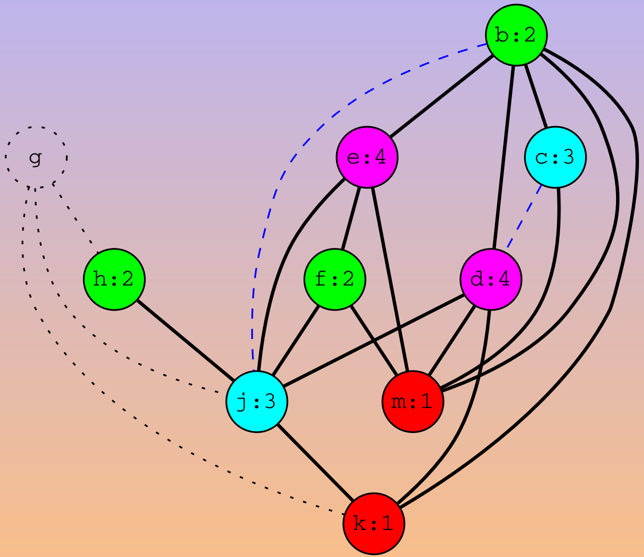
d
k
h
g

Interference Graph: Color 2

k
h
g



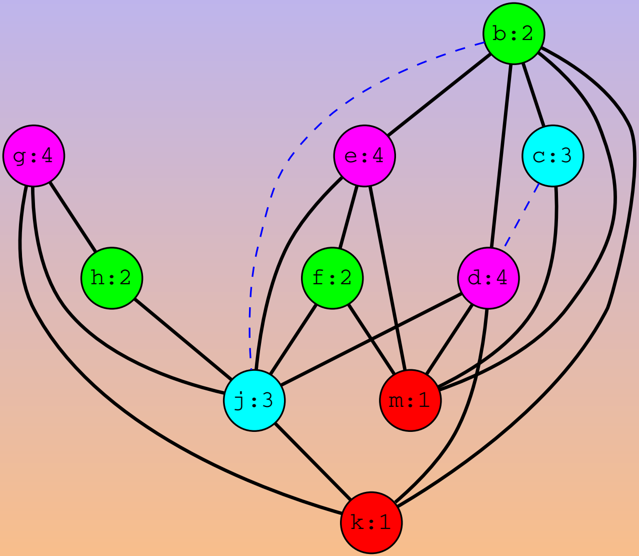
Interference Graph: Color 1



h
g

Interference Graph: Color 0

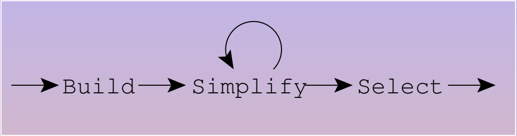
g



Result

```
live in: k j
  g := [j + 12]
  h := k - 1
  f := g * h
  e := [j + 8]
  m := [j + 16]
  b := [f]
  c := e + 8
  d := c
  k := m + 4
  j := b
live out: d k j
```

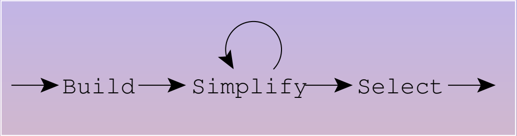
```
live in: r1 r3
  r4 := [r3 + 12]
  r2 := r1 - 1
  r2 := r4 * r2
  r4 := [r3 + 8]
  r1 := [r3 + 16]
  r2 := [r2]
  r3 := r4 + 8
  r4 := r3
  r1 := r1 + 4
  r3 := r2
live out: r4 r1 r3
```



build the conflict graph from the program

simplify the nodes with insignificant degree

select (or color) while rebuilding the graph.

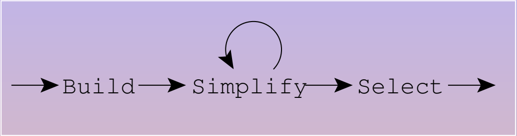


build the conflict graph from the program

simplify the nodes with insignificant degree

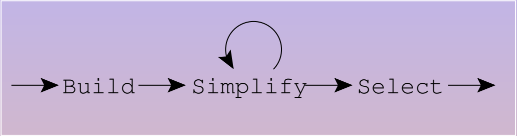
select (or color) while rebuilding the graph.

Simple Register Allocation



build	the conflict graph from the program
simplify	the nodes with insignificant degree
select (or color)	while rebuilding the graph.

Simple Register Allocation



build	the conflict graph from the program
simplify	the nodes with insignificant degree
select (or color)	while rebuilding the graph.

Based on:

> *A.B. Kempe. On the Geographical problem of the four colors, Am. J. Math 2, 193–200, 1879.*

[Appel, 1998, Matz, 2003]

Yes, but What Color? [Matz, 2003]

- Usually, first-fit (registers are ordered).
- Trying caller save first helps.
- **Biased Coloring.** [Briggs, 1992]
  Use a color already unavailable to our neighbors.

Yes, but What Color? [Matz, 2003]

- Usually, first-fit (registers are ordered).
- Trying caller save first helps.
- **Biased Coloring.** [Briggs, 1992]
  Use a color already unavailable to our neighbors.

Yes, but What Color? [Matz, 2003]

- Usually, first-fit (registers are ordered).
- Trying caller save first helps.
- **Biased Coloring.** [Briggs, 1992]
  Use a color already unavailable to our neighbors.

Spilling

A map can always be colored with 4 colors...

Spilling

A map can always be colored with 4 colors...

But for graph coloring, there is no reason for:

- this simple heuristics to always find a solution,
- a solution to always exist...

Spilling

A map can always be colored with 4 colors. . .

But for graph coloring, there is no reason for:

- this simple heuristics to always find a solution,
- a solution to always exist. . .

Spilling

- Not enough registers
```
t1 := t1 + t2
```
- So use the stack
```
[sp + 4] := [sp + 4] + [sp + 8]
```
- But use temporaries to do so!
```
t12 := [sp + 4]
t13 := [sp + 8]
t12 := t12 + t13
[sp + 4] := t12
```
- Why should it solve the problem?

Spilling

- Not enough registers
  `t1 := t1 + t2`
- So use the stack
  `[sp + 4] := [sp + 4] + [sp + 8]`
- But use temporaries to do so!

```
t12 := [sp + 4]
t13 := [sp + 8]
t12 := t12 + t13
[sp + 4] := t12
```

- Why should it solve the problem?

Spilling

- Not enough registers
  `t1 := t1 + t2`
- So use the stack
  `[sp + 4] := [sp + 4] + [sp + 8]`
- But use temporaries to do so!

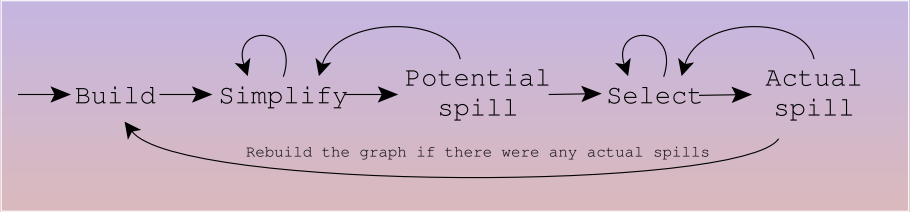
```
t12 := [sp + 4]
t13 := [sp + 8]
t12 := t12 + t13
[sp + 4] := t12
```

- Why should it solve the problem?

Spilling

- Not enough registers
  `t1 := t1 + t2`
- So use the stack
  ```
  [sp + 4] := [sp + 4] + [sp + 8]
  ```
- But use temporaries to do so!
  ```
  t12 := [sp + 4]
  t13 := [sp + 8]
  t12 := t12 + t13
  [sp + 4] := t12
  ```
- Why should it solve the problem?

Register Allocation with Spills



spill when one cannot simplify, the (uses of the) temporary must be rewritten using the stack.

rebuild but then, the conflict graph is to be rewritten

[Appel, 1998, Matz, 2003]

Yes, But Who Should be Spilled?

- The simplification order does not matter
- The spilling order matters
- Spilling decreases the degree of the neighbors
- ...hence it enables additional simplifications
- ...so "first spilled, last served"
- ...therefore: spill cheap temporaries
  - few def/uses
  - pay attention to loops

Yes, But Who Should be Spilled?

- The simplification order does not matter
- The spilling order matters
- Spilling decreases the degree of the neighbors
- ... hence it enables additional simplifications
- ... so "first spilled, last served"
- ... therefore: spill cheap temporaries
  - few def/uses
  - pay attention to loops

Yes, But Who Should be Spilled?

- The simplification order does not matter
- The spilling order matters
- Spilling decreases the degree of the neighbors
- ...hence it enables additional simplifications
- ...so "first spilled, last served"
- ...therefore: spill cheap temporaries
  - few def/uses
  - pay attention to loops

Yes, But Who Should be Spilled?

- The simplification order does not matter
- The spilling order matters
- Spilling decreases the degree of the neighbors
- ... hence it enables additional simplifications
- ... so "first spilled, last served"
- ... therefore: spill cheap temporaries
  - few def/uses
  - pay attention to loops

Yes, But Who Should be Spilled?

- The simplification order does not matter
- The spilling order matters
- Spilling decreases the degree of the neighbors
- ...hence it enables additional simplifications
- ...so "first spilled, last served"
- ...therefore: spill cheap temporaries
  - few def/uses
  - pay attention to loops

Yes, But Who Should be Spilled?

- The simplification order does not matter
- The spilling order matters
- Spilling decreases the degree of the neighbors
- . . . hence it enables additional simplifications
- . . . so "first spilled, last served"
- . . . therefore: spill cheap temporaries
  - few def/uses
  - pay attention to loops

Yes, But Who Should be Spilled?

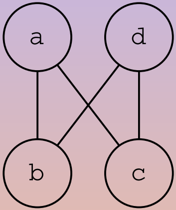
- The simplification order does not matter
- The spilling order matters
- Spilling decreases the degree of the neighbors
- . . . hence it enables additional simplifications
- . . . so "first spilled, last served"
- . . . therefore: spill cheap temporaries
  - few def/uses
  - pay attention to loops

Yes, But Who Should be Spilled?

- The simplification order does not matter
- The spilling order matters
- Spilling decreases the degree of the neighbors
- ... hence it enables additional simplifications
- ... so "first spilled, last served"
- ... therefore: spill cheap temporaries
    - few def/uses
    - pay attention to loops

Optimistic Coloring

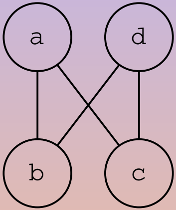
- We miss many opportunities to avoid the stack



- Handle spills as if they were simplified (*potential spills*)
- then try to color them
- There might not be *actual spills*

Optimistic Coloring

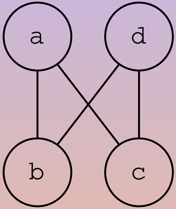
- We miss many opportunities to avoid the stack



- Handle spills as if they were simplified (*potential spills*)
- then try to color them
- There might not be *actual spills*

Optimistic Coloring

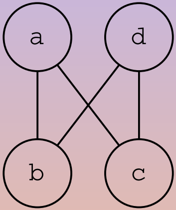
- We miss many opportunities to avoid the stack



- Handle spills as if they were simplified (*potential spills*)
- then try to color them
- There might not be *actual spills*

Optimistic Coloring

- We miss many opportunities to avoid the stack



- Handle spills as if they were simplified (*potential spills*)
- then try to color them
- There might not be *actual spills*

Coalescing

Coalescing

- Some low-level form of *copy propagation*
- While building traces we tried to remove `jumps`
- While allocating registers, we try to remove `moves`

live-in: `t2`

```
t1 := ...
t2 := t1 + t2
t3 := t2
t4 := t1 + t3
t2 := t3 + t4
t1 := t2 - t4
```

live-out: `t1`

Coalescing

- Some low-level form of *copy propagation*
- While building traces we tried to remove `jumps`
- While allocating registers, we try to remove `moves`

live-in: `t2`

```
t1 := ...
t2 := t1 + t2
t3 := t2
t4 := t1 + t3
t2 := t3 + t4
t1 := t2 - t4
```

live-out: `t1`

Coalescing

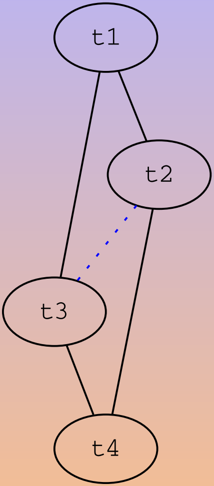
- Some low-level form of *copy propagation*
- While building traces we tried to remove `jumps`
- While allocating registers, we try to remove `moves`

  live-in: `t2`

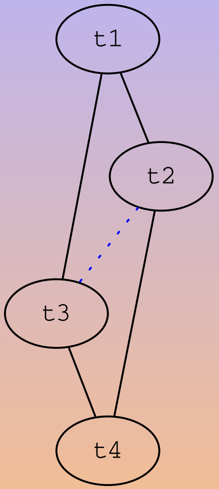
  ```
  t1 := ...
  t2 := t1 + t2
  t3 := t2
  t4 := t1 + t3
  t2 := t3 + t4
  t1 := t2 - t4
  ```

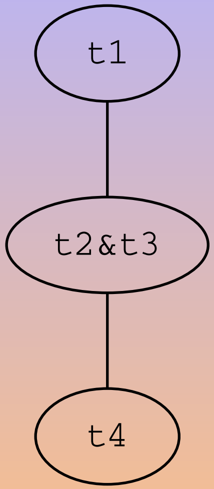
  live-out: `t1`

Coalescing Improves the Coloralibility

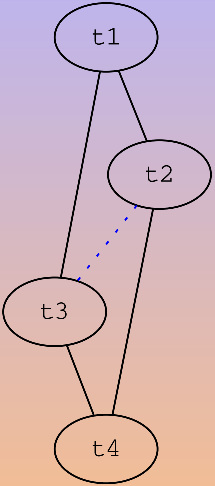


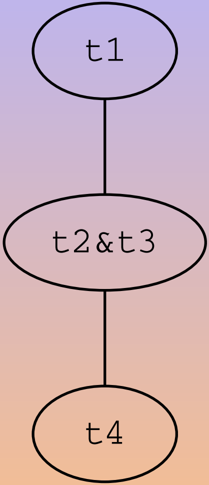
Coalescing Improves the Coloralibility





Coalescing Improves the Coloralibility





t1 and t4 have one neighbor less!

Yes, But Coalesce Who?

- *Conservative Coalescing*: don't make it harder.
- Coalesce `a` and `b` if

  Briggs: `ab` has fewer than k neighbors of significant degree

  George: every neighbor of `a` is

  - of insignificant degree
  - already interfering with `b`
- George's criterion is well suited for real registers

Yes, But Coalesce Who?

- *Conservative Coalescing*: don't make it harder.
- Coalesce `a` and `b` if

  Briggs `ab` has fewer than k neighbors of significant degree.

  George every neighbor of `a` is
  - of insignificant degree
  - already interfering with `b`
- George's criterion is well suited for real registers

Yes, But Coalesce Who?

- *Conservative Coalescing*: don't make it harder.
- Coalesce a and b if

  Briggs ab has fewer than k neighbors of significant degree.

  George every neighbor of a is

  - of insignificant degree
  - already interfering with b
- George's criterion is well suited for real registers

Yes, But Coalesce Who?

- *Conservative Coalescing*: don't make it harder.
- Coalesce a and b if

  Briggs ab has fewer than k neighbors of significant degree.

  George every neighbor of a is
  - of insignificant degree
  - already interfering with b
- George's criterion is well suited for real registers

Yes, But Coalesce Who?

- *Conservative Coalescing*: don't make it harder.
- Coalesce a and b if

  Briggs ab has fewer than k neighbors of significant degree.

  George every neighbor of a is

  - of insignificant degree
  - already interfering with b
- George's criterion is well suited for real registers

Yes, But Coalesce Who?

- *Conservative Coalescing*: don't make it harder.
- Coalesce a and b if

  Briggs ab has fewer than k neighbors of significant degree.

  George every neighbor of a is
    - of insignificant degree
    - already interfering with b
- George's criterion is well suited for real registers

Yes, But Coalesce Who?

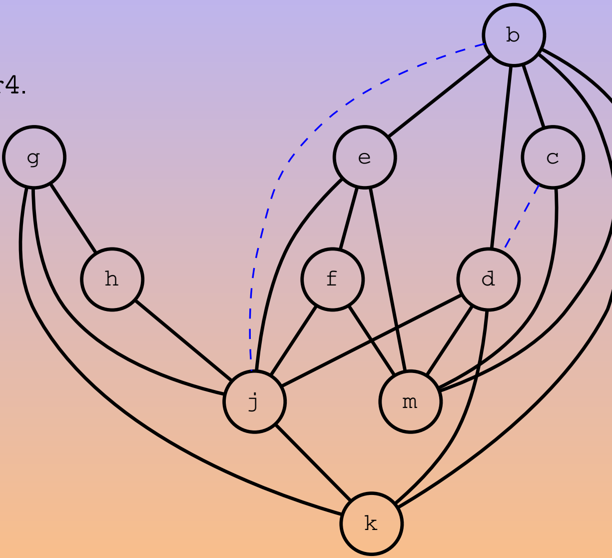
- *Conservative Coalescing*: don't make it harder.
- Coalesce `a` and `b` if

  Briggs `ab` has fewer than k neighbors of significant degree.

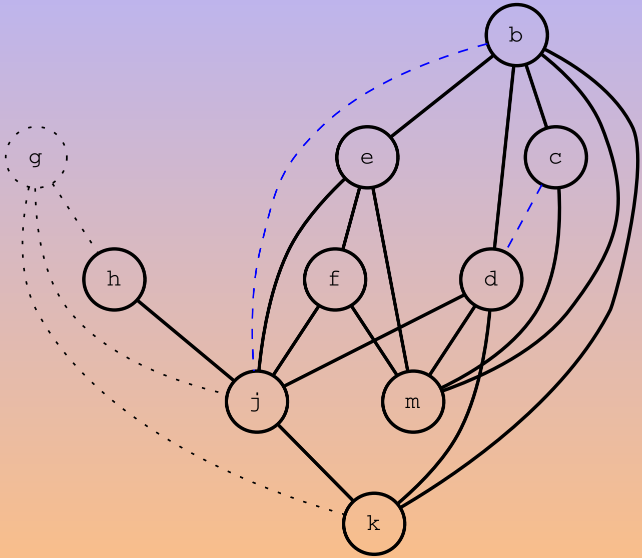
  George every neighbor of `a` is
    - of insignificant degree
    - already interfering with `b`
- George's criterion is well suited for real registers

Four registers: `r1`, `r2`, `r3`, `r4`.

```
live in: k j
  g := [j + 12]
  h := k - 1
  f := g * h
  e := [j + 8]
  m := [j + 16]
  b := [f]
  c := e + 8
  d := c
  k := m + 4
  j := b
live out: d k j
```

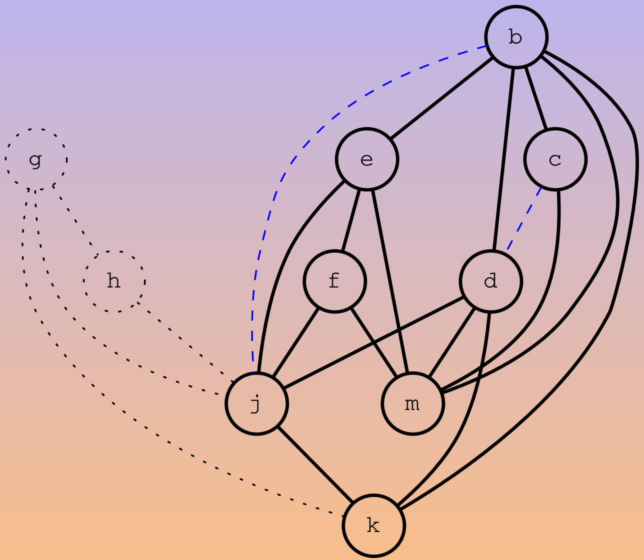


Interference Graph: Simplify 0



g

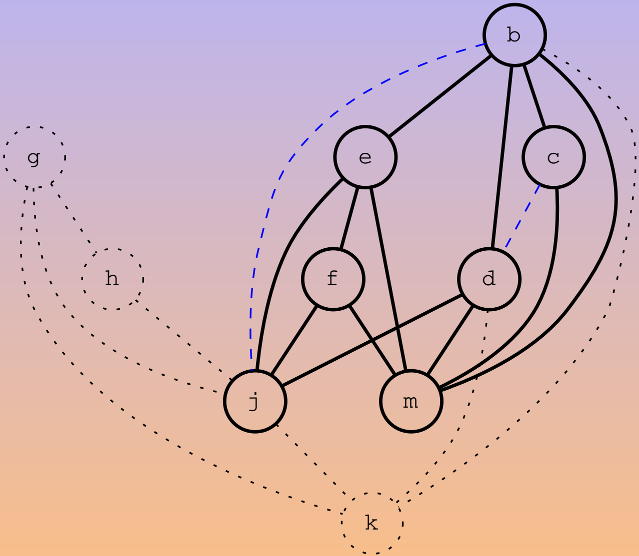
Interference Graph: Simplify 1



h
g

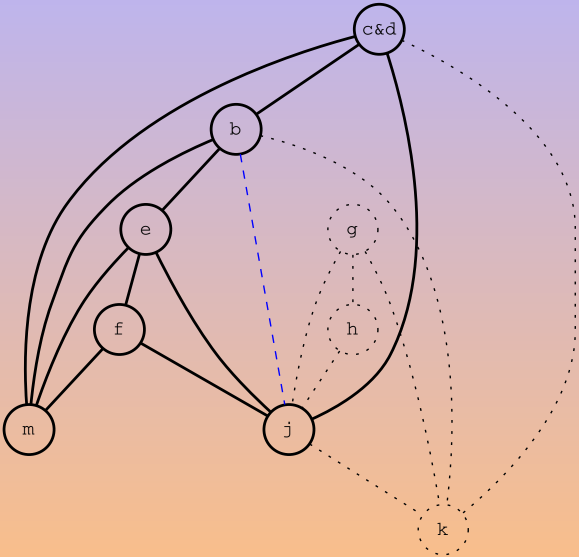
Interference Graph: Simplify 2

k
h
g

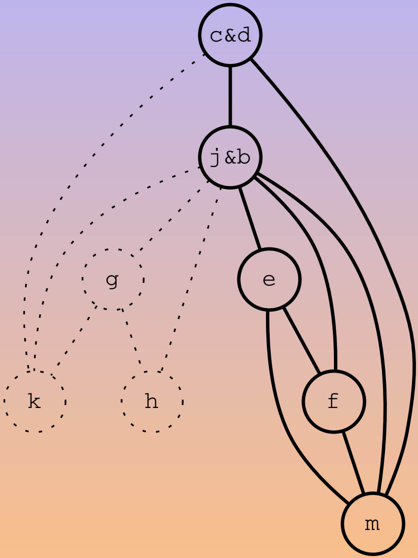


Interference Graph: Simplify 3

c&d
k
h
g

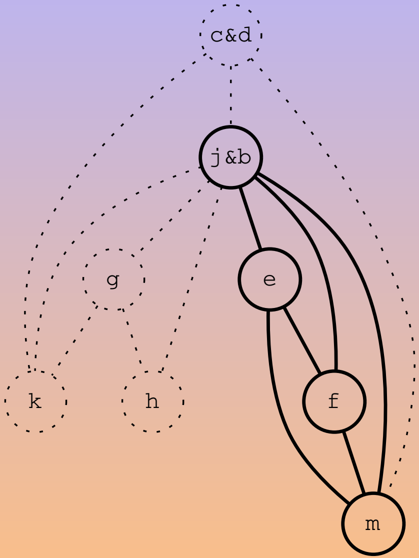


j&b
c&d
k
h
g



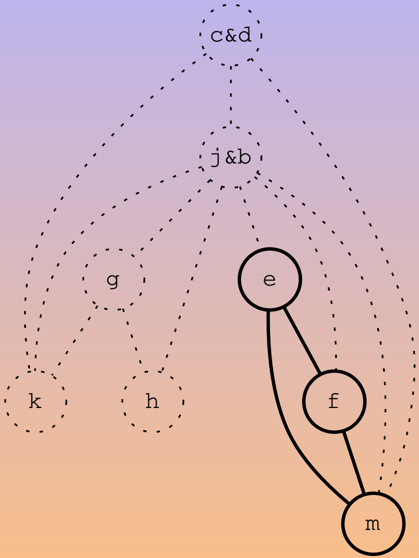
Interference Graph: Simplify 5

c&d
j&b
c&d
k
h
g



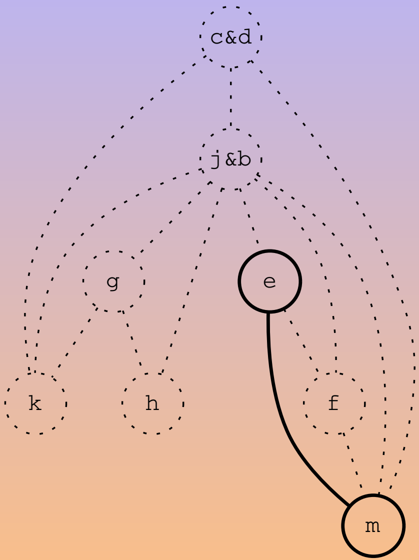
Interference Graph: Simplify 6

j&b
c&d
j&b
c&d
k
h
g



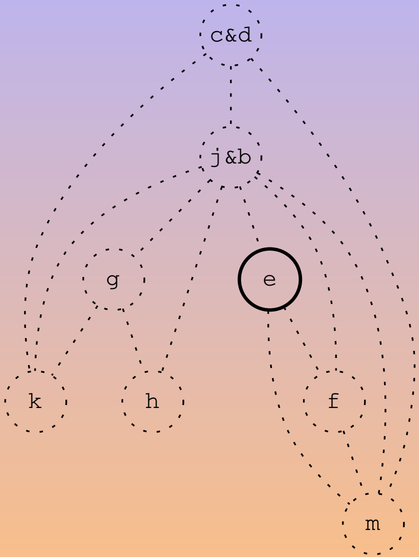
Interference Graph: Simplify 7

f
j&b
c&d
j&b
c&d
k
h
g



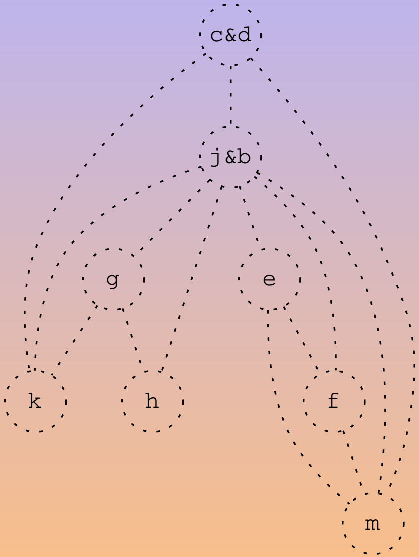
Interference Graph: Simplify 8

m
f
j&b
c&d
j&b
c&d
k
h
g



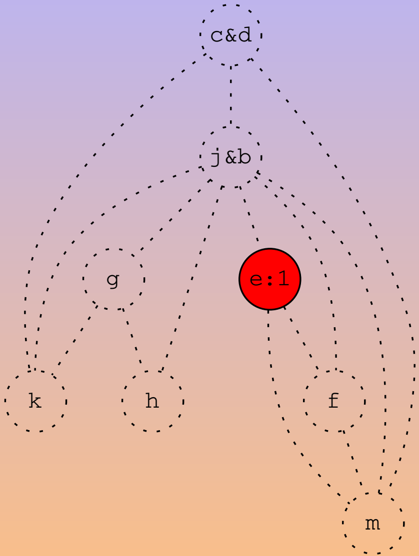
Interference Graph: Simplify 9

e
m
f
j&b
c&d
j&b
c&d
k
h
g



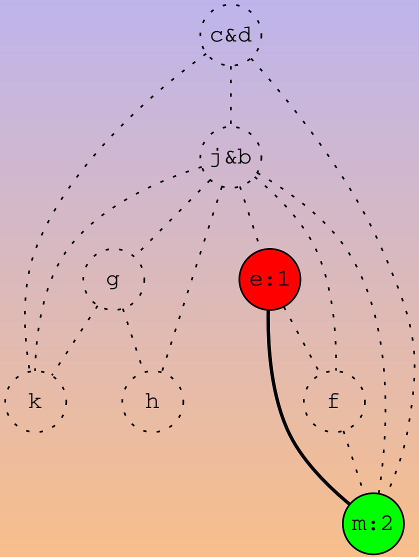
Interference Graph: Simplify 9

e
m
f
j&b
c&d
j&b
c&d
k
h
g



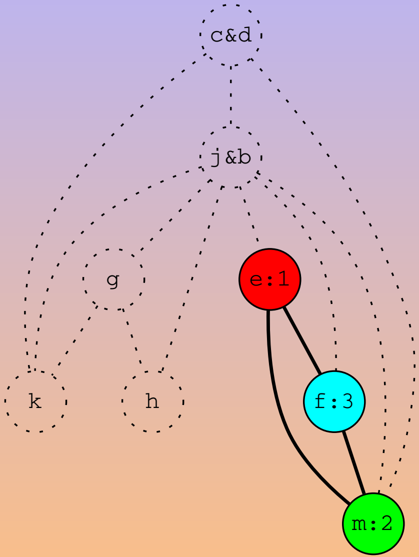
Interference Graph: Simplify 8

m
f
j&b
c&d
j&b
c&d
k
h
g



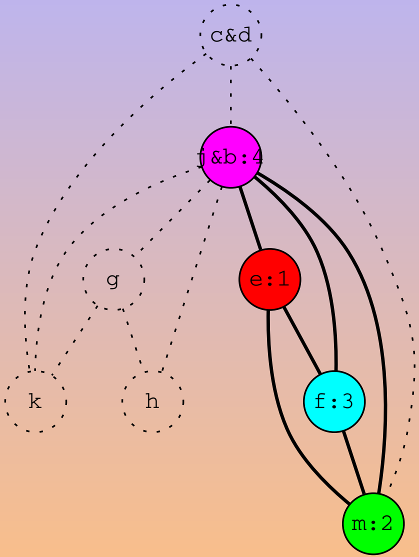
Interference Graph: Simplify 7

f
j&b
c&d
j&b
c&d
k
h
g



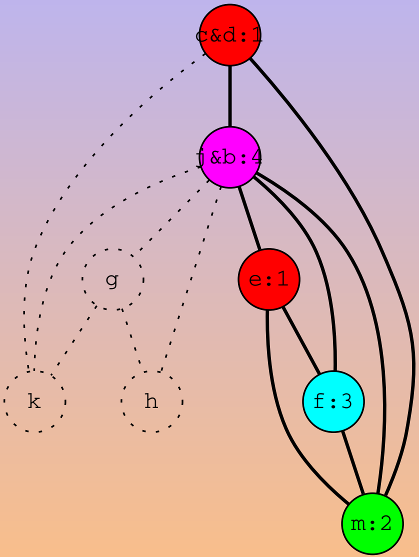
Interference Graph: Simplify 6

j&b
c&d
j&b
c&d
k
h
g



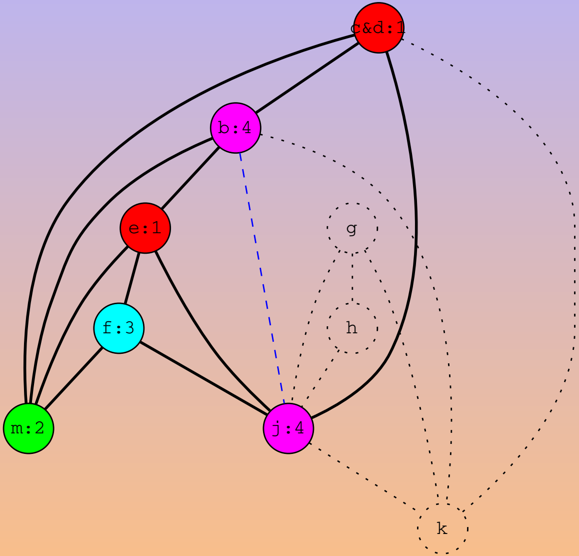
Interference Graph: Simplify 5

c&d
j&b
c&d
k
h
g



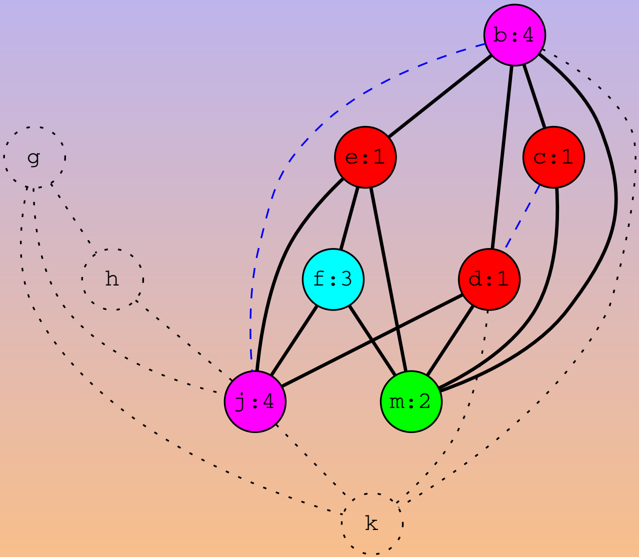
Interference Graph: Simplify 4

j&b
c&d
k
h
g



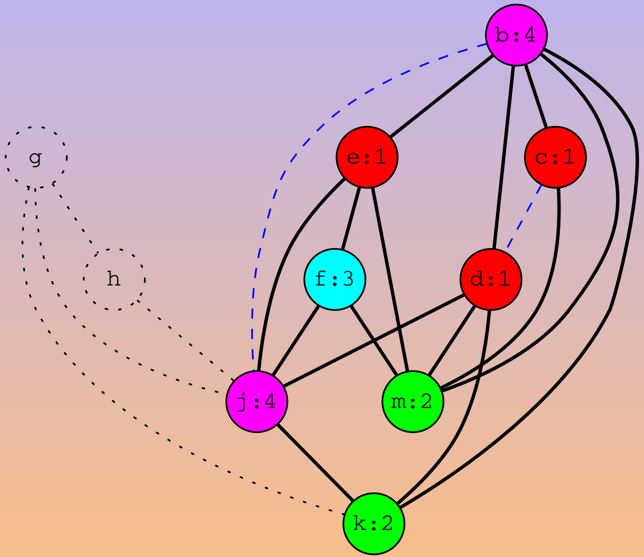
Interference Graph: Simplify 3

c&d
k
h
g

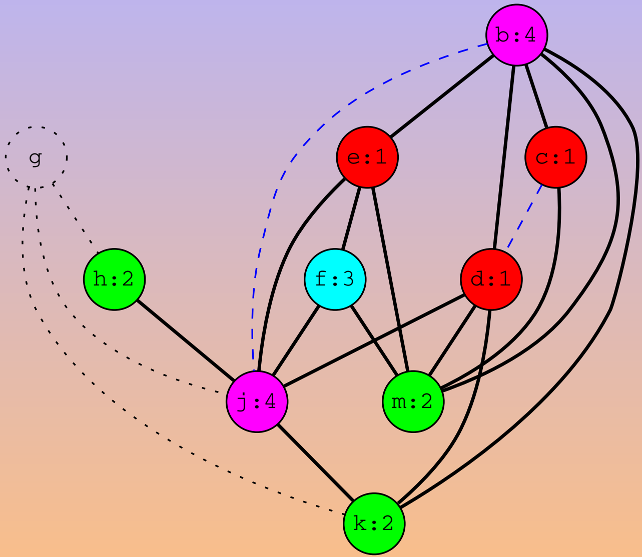


Interference Graph: Simplify 2

k
h
g

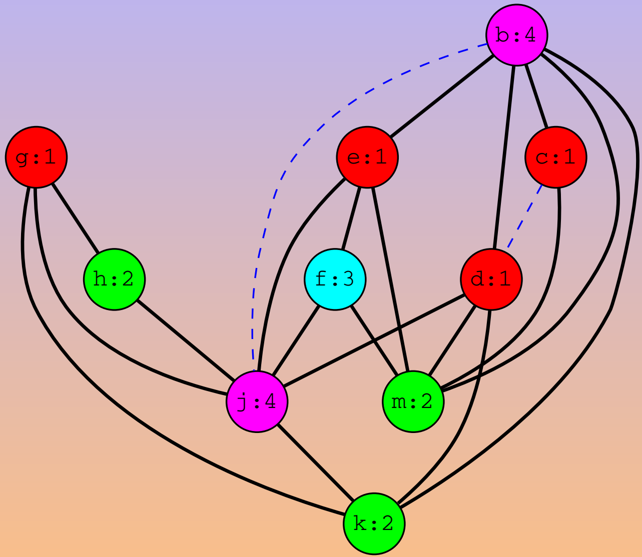


Interference Graph: Simplify 1



h
g

Interference Graph: Simplify 0



g

Interference Graph: Result

```
live in: k j
  g := [j + 12]
  h := k - 1
  f := g * h
  e := [j + 8]
  m := [j + 16]
  b := [f]
  c := e + 8
  d := c
  k := m + 4
  j := b
live out: d k j
```

```
live in: r2 r4
  r1 := [r4 + 12]
  r2 := r2 - 1
  r3 := r1 * r2
  r1 := [r4 + 8]
  r2 := [r4 + 16]
  r4 := [r3]
  r1 := r1 + 8
# r1 := r1
  r2 := r2 + 4
# r4 := r4
live out: r1 r2 r4
```

Precolored Nodes

Hard Registers

- Some nodes are precolored: the real registers
  - the stack pointer (`$sp`)
  - the frame pointer (`$fp`)
  - the argument registers (`$a0`, `$a1`, etc.)
  - the return value (`$v0`, `$v1`)
  - the return address (`$ra`)
  - etc.
- They all interfere with each other
- They cannot be simplified (infinite degree)

Hard Registers

- Some nodes are precolored: the real registers
  - the stack pointer (`$sp`)
  - the frame pointer (`$fp`)
  - the argument registers (`$a0`, `$a1`, etc.)
  - the return value (`$v0`, `$v1`)
  - the return address (`$ra`)
  - etc.
- They all interfere with each other
- They cannot be simplified (infinite degree)

Hard Registers

- Some nodes are precolored: the real registers
  - the stack pointer (`$sp`)
  - the frame pointer (`$fp`)
  - the argument registers (`$a0`, `$a1`, etc.)
  - the return value (`$v0`, `$v1`)
  - the return address (`$ra`)
  - etc.
- They all interfere with each other
- They cannot be simplified (infinite degree)

Hard Registers

- Some nodes are precolored: the real registers
  - the stack pointer (`$sp`)
  - the frame pointer (`$fp`)
  - the argument registers (`$a0`, `$a1`, etc.)
  - the return value (`$v0`, `$v1`)
  - the return address (`$ra`)
  - etc.
- They all interfere with each other
- They cannot be simplified (infinite degree)

Hard Registers

- Some nodes are precolored: the real registers
  - the stack pointer (`$sp`)
  - the frame pointer (`$fp`)
  - the argument registers (`$a0`, `$a1`, etc.)
  - the return value (`$v0`, `$v1`)
  - the return address (`$ra`)
  - etc.
- They all interfere with each other
- They cannot be simplified (infinite degree)

Callee & Caller Save Registers

- It just rocks!

  Caller Save Def'd by `calls`.

  Callee Save Def'd at entry, used at exit of functions.

- Register pressure will push temporaries live across calls into callee save.

Callee & Caller Save Registers

- It just rocks!

  Caller Save — Def'd by `calls`.

  Callee Save — Def'd at entry, used at exit of functions.

- Register pressure will push temporaries live across calls into callee save.

Callee & Caller Save Registers

- It just rocks!

  Caller Save
  : Def'd by `calls`.

  Callee Save
  : Def'd at entry, used at exit of functions.

- Register pressure will push temporaries live across calls into callee save.

Callee & Caller Save Registers

- It just rocks!

  Caller Save
  : Def'd by `calls`.

  Callee Save
  : Def'd at entry, used at exit of functions.

- Register pressure will push temporaries live across calls into callee save.

Conflicts

Minimize the conflicts ("pressure") with hard regs. Source and sink.

```
# Routine: fact
l0:                     # def $s0, $s1...
  move $x11, $s0        # def: $x11 use: $s0
  move $x12, $s1        # def: $x12 use: $s1
  ...
l6:
  move $s0, $x11        # def: $s0 use: $x11
  move $s1, $x12        # def: $s1 use: $x12
  ...
                        # use: $fp, $ra, $sp,
                        # ...  $v0, $zero
```

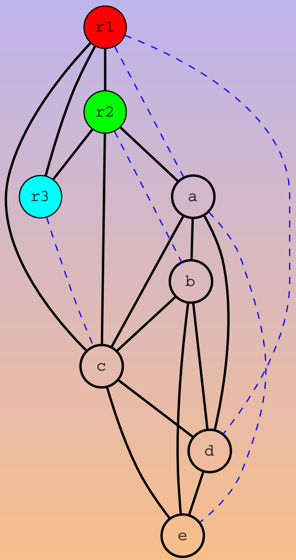
Example [Appel, 1998]

```c
int
f (int a, int b)
{
  int d = 0;
  int e = a;
  do
  {
    d += b;
    --e;
  } while (e > 0);
  return d;
}
```

```
enter:
  c := r3
  a := r1
  b := r2
  d := 0
  e := a
loop:
  d := d + b
  e := e - 1
  if e > 0 goto loop
  r1 := d
  r3 := c
  return
# liveout: r1, r3
```

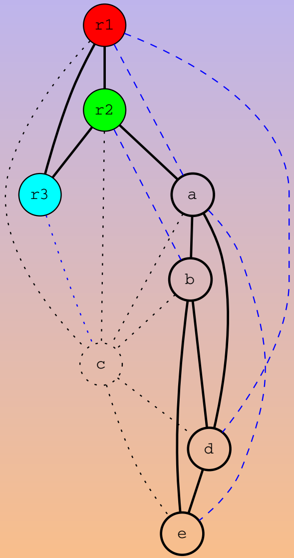
Example

```
enter:
  c := r3
  a := r1
  b := r2
  d := 0
  e := a
loop:
  d := d + b
  e := e - 1
  if e > 0 goto loop
  r1 := d
  r3 := c
  return
# liveout: r1, r3
```



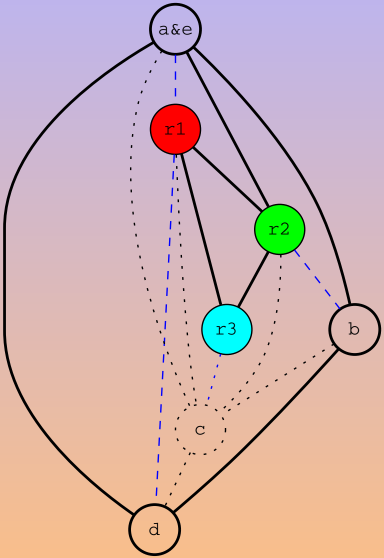
Interference Graph: Simplify 0

c



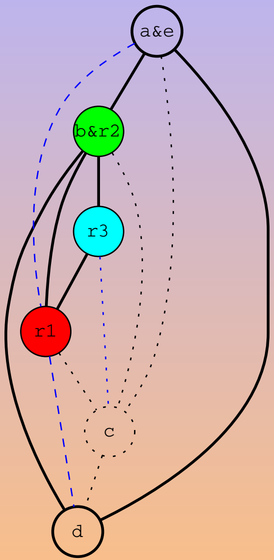
Interference Graph: Simplify 1

a&e
c



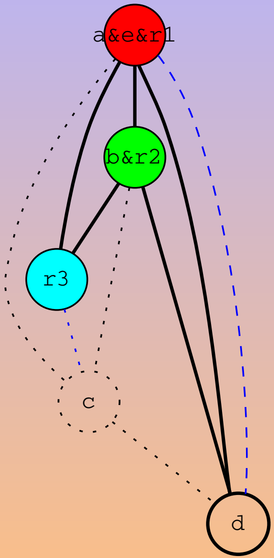
Interference Graph: Simplify 2

b&r2
a&e
c



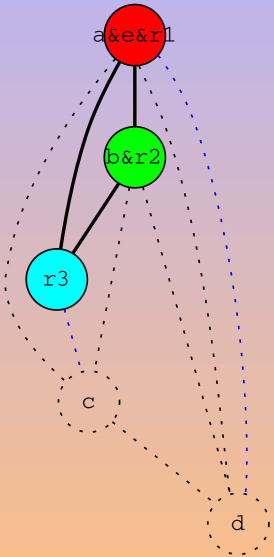
Interference Graph: Simplify 3

a&e&r1
b&r2
a&e
c



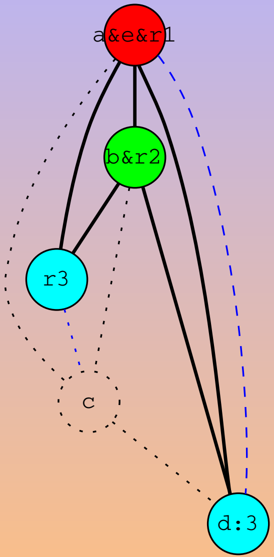
Interference Graph: Simplify 4

d
*a&e&r*1
*b&r*2
a&e
c



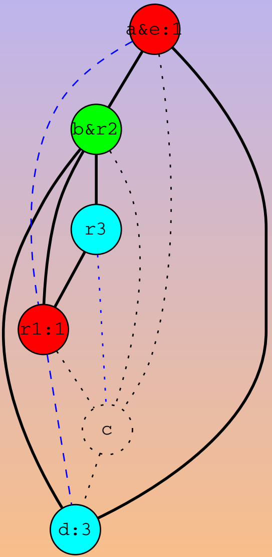
Interference Graph: Simplify 4

d
a&e&r1
b&r2
a&e
c



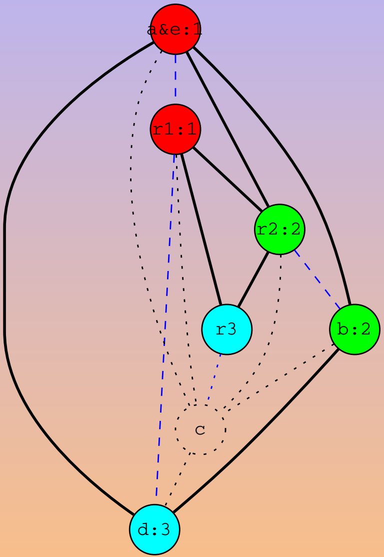
Interference Graph: Simplify 3

$a\&e\&r1$
$b\&r2$
$a\&e$
c



Interference Graph: Simplify 2

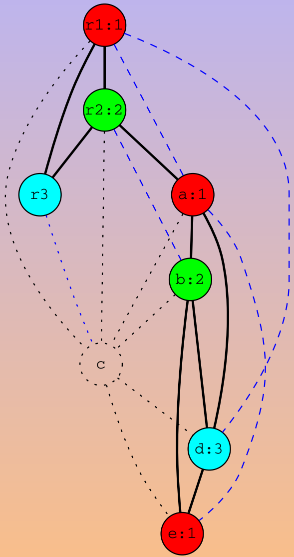
b&r2
a&e
c



Interference Graph: Simplify 1

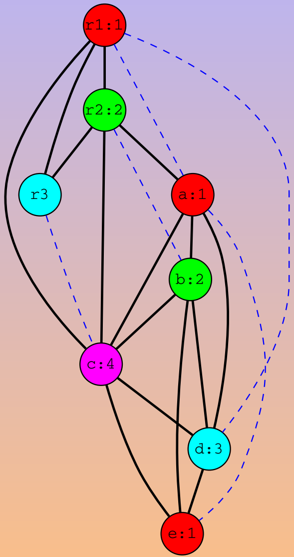
a&*e*

c



Interference Graph: Simplify 0

c



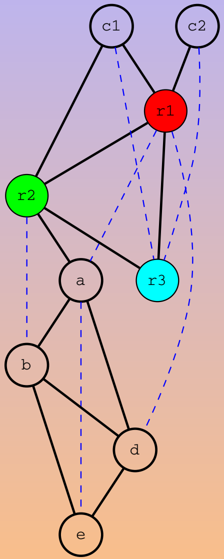
Spilling

```
enter:
  c := r3
  a := r1
  b := r2
  d := 0
  e := a
loop:
  d := d + b
  e := e - 1
  if e > 0 goto loop
  r1 := d
  r3 := c
  return
# liveout: r1, r3
```

```
enter:
  c1   := r3
  [sp+8]  := c1
  a    := r1
  b    := r2
  d    := 0
  e    := a
loop:
  d    := d + b
  e    := e - 1
  if e > 0
     goto loop
  r1    := d
  c2    := [sp+8]
  r3    := c2
  return
# liveout: r1, r3
```

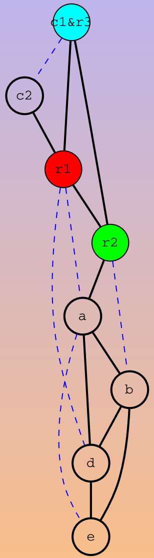
Example

```
enter:
  c1   := r3
  [sp+8] := c1
  a    := r1
  b    := r2
  d    := 0
  e    := a
loop:
  d    := d + b
  e    := e - 1
  if e > 0
     goto loop
  r1    := d
  c2    := [sp+8]
  r3    := c2
  return
# liveout: r1, r3
```



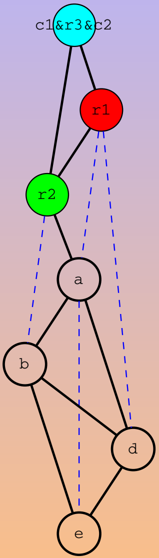
Interference Graph: Simplify 0

$c1\&r3$



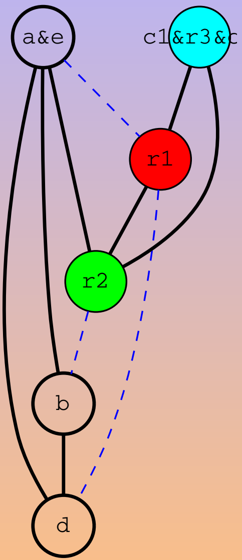
Interference Graph: Simplify 1

$c1\&r3\&c2$
$c1\&r3$



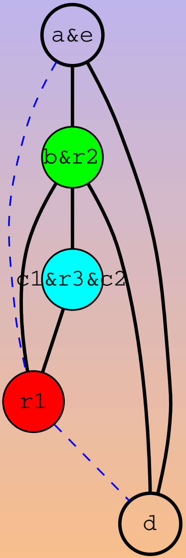
Interference Graph: Simplify 2

$a\&e$
$c1\&r3\&c2$
$c1\&r3$



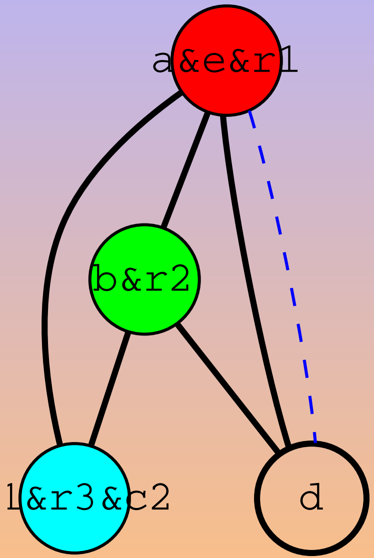
Interference Graph: Simplify 3

b&r2
a&e
c1&r3&c2
c1&r3



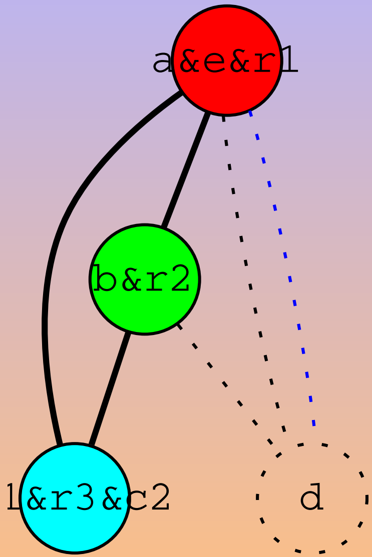
Interference Graph: Simplify 4

$a\&e\&r1$
$b\&r2$
$a\&e$
$c1\&r3\&c2$
$c1\&r3$



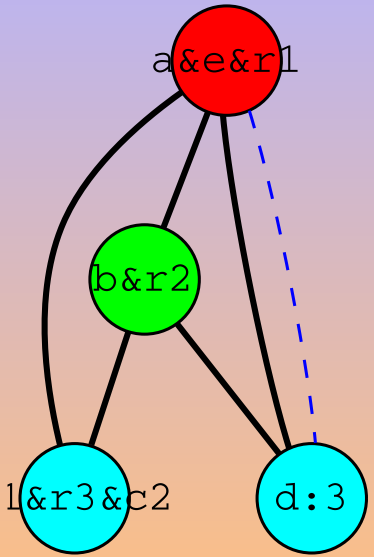
Interference Graph: Simplify 5

d
$a\&e\&r1$
$b\&r2$
$a\&e$
$c1\&r3\&c2$
$c1\&r3$

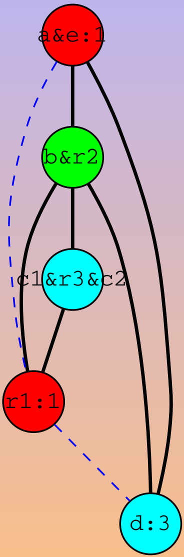


Interference Graph: Simplify 5

d
$a\&e\&r1$
$b\&r2$
$a\&e$
$c1\&r3\&c2$
$c1\&r3$

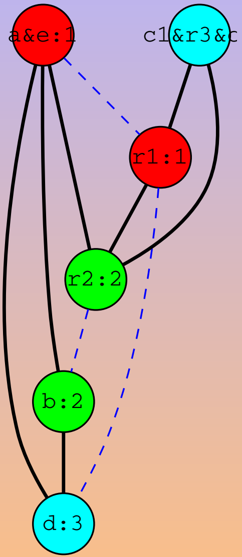


$a\&e\&r1$
$b\&r2$
$a\&e$
$c1\&r3\&c2$
$c1\&r3$



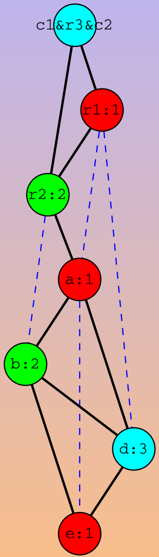
Interference Graph: Simplify 3

$b\&r2$
$a\&e$
$c1\&r3\&c2$
$c1\&r3$



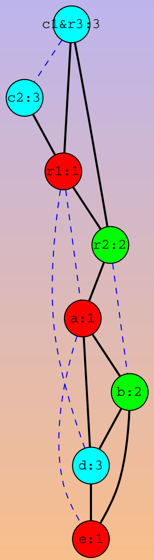
Interference Graph: Simplify 2

$a\&e$
$c1\&r3\&c2$
$c1\&r3$



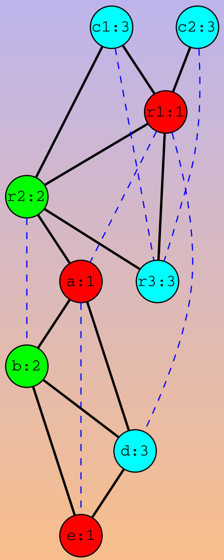
Interference Graph: Simplify 1

$c1\&r3\&c2$
$c1\&r3$



Interference Graph: Simplify 0

$c1 \& r3$



Result

```
enter:
  c1    := r3
  [sp+8] := c1
  a     := r1
  b     := r2
  d     := 0
  e     := a
loop:
  d     := d + b
  e     := e - 1
  if e > 0
     goto loop
  r1    := d
  c2    := [sp+8]
  r3    := c2
  return
# liveout: r1, r3
```

```
enter:
  r3    := r3
  [sp+8] := r3
  r1    := r1
  r2    := r2
  r3    := 0
  r1    := r1
loop:
  r3    := r3 + r2
  r1    := r1 - 1
  if r1 > 0
     goto loop
  r1    := r3
  r3    := [sp+8]
  r3    := r3
  return
# liveout: r1, r3
```

```
enter:

  [sp+8] := r3


  r3    := 0

loop:
  r3    := r3 + r2
  r1    := r1 - 1
  if r1 > 0
     goto loop
  r1    := r3
  r3    := [sp+8]

  return
# liveout: r1, r3
```

Implementation

1. Interference Graph

2. Coloring by Simplification
   - Spilling
   - Coalescing
   - Precolored Nodes
   - Implementation

3. Alternatives to Graph Coloring

Implementation

- Naive implementation is quadratic
- Lower with heavy use of *worklists*
- Queries on the conflict graph
  - Iterate over neighbors, hence adjacency list
  - Existence of an edge between two nodes, hence bit matrix.
- For more information, see [Appel, 1998].

Implementation

- Naive implementation is quadratic
- Lower with heavy use of *worklists*
- Queries on the conflict graph
  - Iterate over neighbors, hence adjacency list
  - Existence of an edge between two nodes, hence bit matrix
- For more information, see [Appel, 1998].

Implementation

- Naive implementation is quadratic
- Lower with heavy use of *worklists*
- Queries on the conflict graph
  - Iterate over neighbors, hence adjacency list
  - Existence of an edge between two nodes, hence bit matrix.

- For more information, see [Appel, 1998].

Implementation

- Naive implementation is quadratic
- Lower with heavy use of *worklists*
- Queries on the conflict graph
    - Iterate over neighbors, hence adjacency list
    - Existence of an edge between two nodes, hence bit matrix.

- For more information, see [Appel, 1998].

Implementation

- Naive implementation is quadratic
- Lower with heavy use of *worklists*
- Queries on the conflict graph
  - Iterate over neighbors, hence adjacency list
  - Existence of an edge between two nodes, hence bit matrix.

- For more information, see [Appel, 1998].

Implementation

- Naive implementation is quadratic
- Lower with heavy use of *worklists*
- Queries on the conflict graph
  - Iterate over neighbors, hence adjacency list
  - Existence of an edge between two nodes, hence bit matrix.

  Use both!
- For more information, see [Appel, 1998].

Implementation

- Naive implementation is quadratic
- Lower with heavy use of *worklists*
- Queries on the conflict graph
  - Iterate over neighbors, hence adjacency list
  - Existence of an edge between two nodes, hence bit matrix.

  Use both!
- For more information, see [Appel, 1998].

Register Allocation for Trees

Can be done during instruction selection with maximal munch

```
function SimpleAlloc (t)
  for each nontrivial tile u child of t
    SimpleAlloc (u)
  for each nontrivial tile u child of t
    n := n - 1
  n := n + 1
  assign rn to (the root of) t
```

[Appel, 1998]

Bibliography I

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

- Briggs, P. (1992).
  Register Allocation via Graph Coloring.
  PhD thesis, Rice University, Houston, Texas.

- Matz, M. (2003).
  Design and Implementation of a Graph Coloring Register Allocator for gcc.
  pages 151–169.