Register allocation: What does the NP-completeness of Chaitin et al. really prove?

or

Revisiting register allocation: Why and how?

F. Bouchez*, A. Darte*, C. Guillon†, and F. Rastello*

*LIP UMR CNRS-Inria-ENS Lyon-UCBL
and †STMicroelectronics
France

LCPC’06, New Orleans
Outline

1 Back to school
   - Register allocation
   - Chaitin et al. proof
   - How to color a basic block?

2 Illustrating example
   - Edge splitting and live-range splitting
   - The problem of swaps
   - Summary

3 General programs
   - “Splitting Chaitin”
   - Checking $k$-colorability
   - Practical implications
Outline

1 Back to school
   - Register allocation
   - Chaitin et al. proof
   - How to color a basic block?

2 Illustrating example
   - Edge splitting and live-range splitting
   - The problem of swaps
   - Summary

3 General programs
   - “Splitting Chaitin”
   - Checking $k$-colorability
   - Practical implications
What is register allocation?

(Example borrowed from Appel’s book)

Assign variables to

- Registers:  
- Memory: infinite

Architectural subtleties

- Specific registers (sp, fp, r0, ...);
- Variable affinities (e.g., auto-inc),
  register pairing (e.g., 64 bits ops);
- Distributed register banks, variable
  sizes, etc.

Live-in: k j

g := mem[j+12]
h := k-1
f := g+h
e := mem[j+8]
m := mem[j+16]
b := mem[f]
c := e+8
d := c
k := m+4
j := b

Live-out: d k j
What is register allocation?

(Example borrowed from Appel’s book)

Assign variables to

- Registers: [ ] [ ] [ ]
- Memory: infinite

Architectural subtleties

- Specific registers (sp, fp, r0, ...);
- Variable affinities (e.g., auto-inc), register pairing (e.g., 64 bits ops);
- Distributed register banks, variable sizes, etc.

Live-in: k j
- g := mem[j+12]
- h := k-1
- f := g+h
- e := mem[j+8]
- m := mem[j+16]
- b := mem[f]
- c := e+8
- d := c
- k := m+4
- j := b

Live-out: d k j

Assign variables to

- Registers: [ ] [ ] [ ]
- Memory: infinite

Architectural subtleties

- Specific registers (sp, fp, r0, ...);
- Variable affinities (e.g., auto-inc), register pairing (e.g., 64 bits ops);
- Distributed register banks, variable sizes, etc.
What is register allocation?

(Example borrowed from Appel’s book)

Live-in: $k \ j$

- $g := \text{mem}[j+12]$
- $h := k-1$
- $f := g+h$
- $e := \text{mem}[j+8]$
- $m := \text{mem}[j+16]$
- $b := \text{mem}[f]$
- $c := e+8$
- $d := c$
- $k := m+4$
- $j := b$

Live-out: $d \ k \ j$

Rules of the game

- Fixed instruction schedule
- Insert LOADS and STORES: Spill
- Add register-to-register MOVES: Split
- Delete MOVES: Coalesce
Chaitin et al. model

Live-in: k j
  g := mem[j+12]
  h := k-1
  f := g+h
  e := mem[j+8]
  m := mem[j+16]
  b := mem[f]
  c := e+8
  d := c
  k := m+4
  j := b
Live-out: d k j
Chaitin et al. model

Interference graph

Live-ranges

Register allocation
Back to school
Register allocation
Chaitin et al. NP-completeness reduction

Problem (Chaitin et al. model)

Given a program and k colors, is it possible to color the program using one color per variable? NP-complete
Chaitin et al. NP-completeness reduction

Problem (Chaitin et al. model)

*Given a program and k colors, is it possible to color the program using one color per variable?* NP-complete

\[ b = 2 \]

\[ x = a + b \]
Chaitin et al. NP-completeness reduction

Problem (Chaitin et al. model)

*Given a program and k colors, is it possible to color the program using one color per variable?* NP-complete
Chaitin et al. NP-completeness reduction

Problem (Chaitin et al. model)

Given a program and k colors, is it possible to color the program using one color per variable? **NP-complete**
How to color a basic block?

Example of basic block 3-coloring

Top-down scan

Live-in: k j

\[
g := \text{mem}[j+12] \\
h := k-1 \\
f := g+h \\
e := \text{mem}[j+8] \\
m := \text{mem}[j+16] \\
b := \text{mem}[f] \\
c := e+8 \\
d := c \\
k := m+4 \\
j := b \\
\]

Live-out: d k j

Live-ranges
Example of basic block 3-coloring **Top-down scan**

**Live-in:** $k \ j$
- $g := \text{mem}[j+12]$
- $h := k-1$
- $f := g+h$
- $e := \text{mem}[j+8]$
- $m := \text{mem}[j+16]$
- $b := \text{mem}[f]$
- $c := e+8$
- $d := c$
- $k' := m+4$
- $j' := b$

**Live-out:** $d \ k' \ j'$
Example of basic block 3-coloring Top-down scan

Live-in: $k \ j$

- $g := \text{mem}[j+12]$
- $h := k-1$
- $f := g+h$
- $e := \text{mem}[j+8]$
- $m := \text{mem}[j+16]$
- $b := \text{mem}[f]$
- $c := e+8$
- $d := c$
- $k' := m+4$
- $j' := b$

Live-out: $d \ k' \ j'$
Example of basic block 3-coloring Top-down scan

Live-in: $k \ j$

\[
\begin{align*}
g &:= \text{mem}[j+12] \\
h &:= k-1 \\
f &:= g+h \\
e &:= \text{mem}[j+8] \\
m &:= \text{mem}[j+16] \\
b &:= \text{mem}[f] \\
c &:= e+8 \\
d &:= c \\
k' &:= m+4 \\
j' &:= b
\end{align*}
\]

Live-out: $d \ k' \ j'$
Example of basic block 3-coloring Top-down scan

Live-in: $k \ j$

$g := \text{mem}[j+12]$
$h := k - 1$
$f := g + h$
$e := \text{mem}[j+8]$
$m := \text{mem}[j+16]$
$b := \text{mem}[f]$
$c := e + 8$
$d := c$

$k' := m + 4$
$j' := b$

Live-out: $d \ k' \ j'$
Example of basic block 3-coloring **Top-down scan**

**Live-in:** $k\ j$
- $g := \text{mem}[j+12]$
- $h := k-1$
- $f := g+h$
- $e := \text{mem}[j+8]$
- $m := \text{mem}[j+16]$
- $b := \text{mem}[f]$
- $c := e+8$
- $d := c$
- $k' := m+4$
- $j' := b$

**Live-out:** $d\ k'\ j'$
- $g$
- $h$
- $f$
- $e$
- $m$
- $b$
- $c$
- $d$
- $k$
- $j$
Example of basic block 3-coloring Top-down scan

**Live-in:** k j
- \( g := \text{mem}[j+12] \)
- \( h := k-1 \)
- \( f := g+h \)
- \( e := \text{mem}[j+8] \)
- \( m := \text{mem}[j+16] \)
- \( b := \text{mem}[f] \)
- \( c := e+8 \)
- \( d := c \)
- \( k' := m+4 \)
- \( j' := b \)

**Live-out:** d k’ j’
Problem (On a program, with Chaitin et al. view)

Deciding if a program can be $k$-colored. *NP-complete*

Problem (On a Basic-Block, with some renaming)

Deciding if a basic block can be $k$-colored. $\text{Maxlive} \leq k$

(Maxlive: maximum number of simultaneously live variables)
Outline

1. Back to school
   - Register allocation
   - Chaitin et al. proof
   - How to color a basic block?

2. Illustrating example
   - Edge splitting and live-range splitting
   - The problem of swaps
   - Summary

3. General programs
   - “Splitting Chaitin”
   - Checking k-colorability
   - Practical implications
How to color a loop?

circular-arc graphs

```plaintext
Live-in: k j
g := mem[j+12]
h := k-1
f := g+h
e := mem[j+8]
m := mem[j+16]
b := mem[f]
c := e+8
d := c
k := m+4
j := b
Live-out: d k j
```
How to color a loop?

circular-arc graphs

Live-in: \( k \) \( j \)
- \( g := \text{mem}[j+12] \)
- \( h := k-1 \)
- \( f := g+h \)
- \( e := \text{mem}[j+8] \)
- \( m := \text{mem}[j+16] \)
- \( b := \text{mem}[f] \)
- \( c := e+8 \)
- \( d := c \)
- \( k := m+4 \)
- \( j := b \)

Live-out: \( d \) \( k \) \( j \)
How to color a loop?

circular-arc graphs

In practice
- split on back-edge
- color as a basic block
- repair on back-edge
How to color a loop?

circular-arc graphs

In practice

- split on back-edge
- color as a basic block
- repair on back-edge
How to color a loop?

In practice

- split on back-edge
- color as a basic block
- repair on back-edge
If swaps are not allowed

Live = $k$ on back-edge.

NP-complete: Pereira & Palsberg [FOSSACS’06]
But why split live-ranges there?
Split wherever Live < $k$
Scan color from split point.
If swaps are not allowed

Live = $k$ on back-edge.

**NP-complete**: Pereira & Palsberg [FOSSACS’06]

But why split live-ranges there?

Split wherever Live < $k$

Scan color from split point.
If swaps are not allowed

Live = $k$ on back-edge.

**NP-complete:** Pereira & Palsberg [FOSSACS’06]
But why split live-ranges there?

Split wherever Live < $k$

Scan color from split point.
If swaps are not allowed

Live = $k$ on back-edge.

NP-complete: Pereira & Palsberg [FOSSACS’06]
But why split live-ranges there?
Split wherever Live < $k$
Scan color from split point.
If swaps are not allowed

Live $= k$ on back-edge.

**NP-complete:** Pereira & Palsberg [FOSSACS’06]
But why split live-ranges there?
**Split wherever Live $< k$$**
Scan color from split point.
If swaps are not allowed

\( \text{Live} = k \) on back-edge.

**NP-complete:** Pereira & Palsberg [FOSSACS’06]
But why split live-ranges there?
**Split wherever Live < k**
Scan color from split point.
If swaps are not allowed

Live = $k$ on back-edge.

NP-complete: Pereira & Palsberg [FOSSACS’06]
But why split live-ranges there?
Split wherever Live < $k$
Scan color from split point.
If swaps are not allowed

Live = \( k \) on back-edge.

**NP-complete:** Pereira & Palsberg [FOSSACS’06]
But why split live-ranges there?
**Split wherever Live < \( k \)**
Scan color from split point.
If swaps are not allowed

Live = \( k \) on back-edge.

NP-complete: Pereira & Palsberg [FOSSACS’06]
But why split live-ranges there?
Split wherever Live < \( k \)
Scan color from split point.
If swaps are not allowed

Live = \( k \) on back-edge.

**NP-complete**: Pereira & Palsberg [FOSSACS’06]

But why split live-ranges there?

Split wherever Live < \( k \)

Scan color from split point.
If swaps are not allowed

Live = \( k \) on back-edge.

**NP-complete**: Pereira & Palsberg [FOSSACS’06]

But why split live-ranges there?

Split wherever Live < \( k \)

Scan color from split point.
If swaps are not allowed

Live = $k$ on back-edge.

NP-complete: Pereira & Palsberg [FOSSACS’06]

But why split live-ranges there?

Split wherever Live < $k$

Scan color from split point.
If swaps are not allowed

Live = $k$ on back-edge.

NP-complete: Pereira & Palsberg [FOSSACS’06]
But why split live-ranges there?
Split wherever Live < $k$
Scan color from split point.
If swaps are not allowed

Live = $k$ on back-edge.

NP-complete: Pereira & Palsberg [FOSSACS’06]
But why split live-ranges there?
Split wherever Live < $k$
Scan color from split point.
When \( \text{Live} = k \) everywhere

Scan check
- Start anywhere
- Scan coloring

Example not 3-colorable
When Live = k everywhere

Scan check
- Start anywhere
- Scan coloring

Example not 3-colorable
When \( \text{Live} = k \) everywhere

**Scan check**
- Start anywhere
- Scan coloring

Example not 3-colorable
When \( \text{Live} = k \) everywhere

Scan check
- Start anywhere
- Scan coloring

Example not 3-colorable
When $\text{Live} = k$ everywhere

Scan check
- Start anywhere
- Scan coloring

Example not 3-colorable
When $\text{Live} = k$ everywhere

Scan check
- Start anywhere
- Scan coloring

Example not 3-colorable
When $\text{Live} = k$ everywhere

Scan check

- Start anywhere
- Scan coloring

Example not 3-colorable
When $\text{Live} = k$ everywhere

Scan check

- Start anywhere
- Scan coloring

Example not 3-colorable
When \( \text{Live} = k \) everywhere

Scan check

- Start anywhere
- Scan coloring

Example not 3-colorable
Questions raised

Problem (Determining the k-colorability of a program)

- (Critical) edge splitting: allowed or not?
- Live-range splitting: no? On edges? Anywhere?
- Swap instructions: with or without?
Outline

1 Back to school
   - Register allocation
   - Chaitin et al. proof
   - How to color a basic block?

2 Illustrating example
   - Edge splitting and live-range splitting
   - The problem of swaps
   - Summary

3 General programs
   - “Splitting Chaitin”
   - Checking $k$-colorability
   - Practical implications
Splitting edges: Chaitin et al. proof doesn’t hold

\[ a = 1 \]
\[ b = 2 \]
\[ x = a + b \]

\[ a = 3 \]
\[ c = 4 \]
\[ x = a + c \]

\[ b = 5 \]
\[ d = 6 \]
\[ x = b + d \]

\[ c = 7 \]
\[ d = 8 \]
\[ x = c + d \]

\[ B_{a,b} \]
\[ B_{a,c} \]
\[ B_{b,d} \]
\[ B_{c,d} \]

\[ B_a \]
\[ B_b \]
\[ B_c \]
\[ B_d \]

\[ \text{return } a + x \]
\[ \text{return } b + x \]
\[ \text{return } c + x \]
\[ \text{return } d + x \]
Splitting edges: Chaitin et al. proof doesn’t hold

\[
a = 1 \\
b = 2 \\
x = a + b
\]

\[
a = 3 \\
c = 4 \\
x = a + c
\]

\[
b = 5 \\
d = 6 \\
x = b + d
\]

\[
c = 7 \\
d = 8 \\
x = c + d
\]
Splitting edges: Chaitin et al. proof doesn’t hold
Splitting edges: Chaitin et al. proof doesn’t hold

\[
\begin{align*}
  a &= 1 \\
  b &= 2 \\
  x &= a + b \\
  a &= 3 \\
  c &= 4 \\
  x &= a + c \\
  b &= 5 \\
  d &= 6 \\
  x &= b + d \\
  c &= 7 \\
  d &= 8 \\
  x &= c + d
\end{align*}
\]
Splitting edges: Chaitin et al. proof doesn’t hold

```
switch

B_{a,b}  B_{a,c}  B_{b,d}  B_{c,d}

\begin{align*}
  a &= 1 \\
  b &= 2 \\
  x &= a + b \\
  &\quad \text{return } a + x \quad B_a
\\
  a &= 3 \\
  c &= 4 \\
  x &= a + c \\
  &\quad \text{return } b + x \quad B_b \\
\\
  b &= 5 \\
  d &= 6 \\
  x &= b + d \\
  &\quad \text{return } c + x \quad B_c
\\
  c &= 7 \\
  d &= 8 \\
  x &= c + d \\
  &\quad \text{return } d + x \quad B_d
\end{align*}
```
## Splitting edges: easy to check $k$-colorability

### Problem (With swaps, edge splitting, and live-range splitting)

\[ \exists \ k\text{-coloration of the program} \iff Maxlive \leq k \]

### Correct (but inefficient) coloring method

- Split *everywhere*
- Color each program point independently
- Insert permutations to repair coloring + coalescing

### More promising approaches, using fewer MOVES

- Basic block coloring (*interval graph*)
- SSA-like coloring = subtrees of a tree (*chordal graph*)
**Register allocation**

- General programs
- “Splitting Chaitin”

## Splitting edges: easy to check k-colorability

### Problem (With swaps, edge splitting, and live-range splitting)

\[ \exists \text{ } k\text{-coloration of the program } \iff \text{Maxlive} \leq k \]

### Correct (but inefficient) coloring method

- Split everywhere
- Color each program point independently
- Insert permutations to repair coloring + coalescing

### More promising approaches, using fewer MOVES

- Basic block coloring (interval graph)
- SSA-like coloring = subtrees of a tree (chordal graph)
Chaitin et al’s variant if swaps are not available

NP-complete if MOVES on entry/exit of basic blocks only, even for $k = 3$. 

```
switch

B_{a,b} 
\[a = 1\]
\[b = 2\]
\[x_{a,b} = a + b\]

B_{a,c} 
\[a = 3\]
\[c = 4\]
\[x_{a,c} = a + c\]

B_{b,d} 
\[b = 5\]
\[d = 6\]
\[x_{b,d} = b + d\]

B_{c,d} 
\[c = 7\]
\[d = 8\]
\[x_{c,d} = c + d\]

B_{a} 
return $x_a + y_a + a$

B_{b} 
return $x_b + y_b + b$

B_{c} 
return $x_c + y_c + c$

B_{d} 
return $x_d + y_d + d$
```
NP-complete if MOVES on entry/exit of basic blocks only, even for \( k = 3 \).
Chaitin et al’s variant if swaps are not available

NP-complete if MOVES on entry/exit of basic blocks only, even for $k = 3$. 
If swaps are not available, what can we conclude?

**NP-complete** if MOVES on entry & exit of basic blocks only.

**But** why not inserting MOVES in the middle of a block?

NP-complete if instructions can define two variables simultaneously.

Often, either swaps are available, can be emulated (XOR), or such instructions have low register pressure (e.g., function call, 64-bit load).

Polynomial if at most one definition per instruction.

Greedy traversal along the control-flow graph where Live = k.
If swaps are not available, what can we conclude?

\[ a = 1 \quad b = 2 \quad x_{a,b} = a + b \]
\[ a = 3 \quad c = 4 \quad x_{a,c} = a + c \]
\[ b = 5 \quad d = 6 \quad x_{b,d} = b + d \]
\[ c = 7 \quad d = 8 \quad x_{c,d} = c + d \]
If swaps are not available, what can we conclude?
If swaps are not available, what can we conclude?
If swaps are not available, what can we conclude?

**NP-complete** if MOVES on entry & exit of basic blocks only.

**But** why not inserting MOVES in the middle of a block?

**NP-complete** if instructions can define two variables simultaneously.

**But**, often, either swaps are available, can be emulated (XOR), or such instructions have low register pressure (ex: function call, 64 bits load).

**Polynomial** if at most one definition per instruction.

Greedy traversal along the control-flow graph where Live = k.
If swaps are not available, what can we conclude?

**NP-complete** if MOVES on entry & exit of basic blocks only.

**But** why not inserting MOVES in the middle of a block?

**NP-complete** if instructions can define two variables simultaneously.

**But**, often, either swaps are available, can be emulated (XOR), or such instructions have low register pressure (ex: function call, 64 bits load).

**Polynomial** if at most one definition per instruction.

Greedy traversal along the control-flow graph where Live = k.
If swaps are not available, what can we conclude?

**NP-complete** if MOVES on entry & exit of basic blocks only.

But why not inserting MOVES in the middle of a block?

**NP-complete** if instructions can define two variables simultaneously.

But, often, either swaps are available, can be emulated (XOR), or such instructions have low register pressure (ex: function call, 64 bits load).

**Polynomial** if at most one definition per instruction.

Greedy traversal along the control-flow graph where Live = \( k \).
On the complexity of register allocation

No contradiction with Chaitin et al’s proof. It does not say anything about register allocation with live-range splitting and critical edge splitting, which, in most cases, make the problem simpler.

Register allocation remains difficult

- When critical edges cannot be split.
- Because optimal spilling, i.e., what to spill and where, is (almost) always hard.
- Because optimal coalescing is, although easier to formulate, NP-complete in most cases.
On the complexity of register allocation

No contradiction with Chaitin et al’s proof. It does not say anything about register allocation with live-range splitting and critical edge splitting, which, in most cases, make the problem simpler.

Register allocation remains difficult

- When critical edges cannot be split.
- Because optimal spilling, i.e., what to spill and where, is (almost) always hard.
- Because optimal coalescing is, although easier to formulate, NP-complete in most cases.
How to know if spilling is necessary?

Up to now, spilling was done because of a coloring failure in graph-coloring heuristics

- Chaitin (degree \( \geq k \))
- Briggs (potential spill)
- Appel-George
- Biased coloring

But in fact, the spill test is only a check of Maxlive

1. Spill only if necessary
2. Add \texttt{MOVES}, \( k \)-color, coalesce
How to know if spilling is necessary?

Up to now, spilling was done because of a coloring failure in graph-coloring heuristics

- Chaitin (degree \( \geq k \))
- Briggs (potential spill)
- Appel-George
- Biased coloring

But in fact, the spill test is only a check of Maxlive

1. Spill only if necessary
2. Add \texttt{MOVES, k-color, coalesce}
Conclusions and future works

Be careful: Chaitin et al. reduction from graph $k$-coloring only applies when it is not allowed to split neither critical edges nor live-ranges. Only one register per variable.

Maxlive $\leq k$ is in general a good test for deciding if spilling is necessary. Even iterated register coalescing can overspill.

Spilling under SSA does not seem to be a good strategy.

Spilling is hard what to spill and where is challenging.

Coalescing is hard in theory, even with “nice” graph structures. But good optimistic heuristics should be possible.

More experiments need to be done for exploring this new view and tradeoffs between spilling & coalescing.