1 Code representations

2 Out-of-SSA translation and SSA properties

3 Register allocation
   - Register allocation formulation
   - Example: iterated register coalescing
   - Determining if $k$ registers are enough
Where did the NP-completeness disappear?

Chaitin et al.

Can each variable be mapped to one of the \( k \) registers so that simultaneously-live variables are mapped to different registers?

**NP-complete to decide.**

SSA-based register allocation

Can the (chordal) interference graph be colored with \( k \) colors?

**Can be checked in linear time.**

So a proof that \( P = NP \)?
Where did the NP-completeness disappear?

Chaitin et al.
Can each variable be mapped to one of the \( k \) registers so that simultaneously-live variables are mapped to different registers?

NP-complete to decide.

SSA-based register allocation
Can the (chordal) interference graph be colored with \( k \) colors?

Can be checked in linear time.

So a proof that \( P = NP \)? Of course not. But a new track to analyze register allocation subtleties, in particular the impact of:

- Strictness.
- Live-range splitting.
- Critical edges.
- Parallel copies (e.g., swap).
- Instruction types (ISA).
Interpretation of original proof

\[ a = 1 \]
\[ b = 2 \]
\[ x = a + b \]
Interpretation of original proof

\[
\begin{align*}
B_{a,b} & \quad \text{a = 1} \\
& \quad b = 2 \\
& \quad x = a + b \\
B_{a,c} & \quad \text{a = 3} \\
& \quad c = 4 \\
& \quad x = a + c \\
B_a & \quad \text{return a + x}
\end{align*}
\]
Interpretation of original proof

NP-complete if each variable is mapped to a unique register.

But ignore the possibility of using register-to-register moves!
Interpretation of original proof

NP-complete if each variable is mapped to a unique register.

Extension 1: NP-complete with live-range splitting but critical edges.
Interpretation of original proof

NP-complete if each variable is mapped to a unique register.

Extension 1: NP-complete with *live-range splitting* but *critical edges*.
Interpretation of original proof

NP-complete if each variable is mapped to a **unique** register.

Extension 1: NP-complete with *live-range splitting* but **critical edges**.
Interpretation of original proof

NP-complete if each variable is mapped to a unique register.

Extension 1: NP-complete with live-range splitting but critical edges.
Extension 2: Same if no critical edge but program is not strict.

Note: making a program strict (e.g., with SSA) can increase register pressure.
Useless proof if blocks & moves can be inserted!

Alain Darte  Compilation avancée et optimisation de programmes
Strict program, swaps, and edge splitting allowed

Maxlives = maximal number of distinct variables simultaneously live.

- One needs Maxlive ≤ k, so spill to get Maxlive ≤ k.
- Split critical edges (= add basic blocks).
- Color each program point independently with ≤ Maxlive colors.
- Use permutations to match colors (thanks to swaps).

Correct assignment... but with many many moves.
Strict program, swaps, and edge splitting allowed

Maxlive = maximal number of distinct variables simultaneously live.

- One needs Maxlive \( \leq k \), so spill to get Maxlive \( \leq k \).
- Split critical edges (= add basic blocks).
- Color each program point independently with \( \leq \) Maxlive colors.
- Use permutations to match colors (thanks to swaps).

Correct assignment... but with many many moves.

More promising approaches:

- Basic block coloring (*interval graph*).
- SSA-like coloring (*chordal graph*).
- Guided live-range/edge splitting + permutation motion.
What if swaps are not available?

**Pereira&Palsberg question (fossacs’06)**

“Can we do polynomial-time register allocation by first transforming the program to SSA form, then doing linear-time register allocation for the SSA form, and finally doing SSA elimination while maintaining the mapping from temporaries to registers?”

- **NP-complete if swaps are not available.**
  - Reduction from $k$-coloring circular-arc graph.
  - Make sure $k$ variables are live on the back edge (where SSA will split) so that a non-trivial permutation is impossible.

Note: polynomial for a fixed $k$. (See Garey, Johnson, Miller, Papadimitriou.)
If swaps not available: variant of Chaitin et al.

If swaps not available: variant of Chaitin et al.
If swaps not available: variant of Chaitin et al.

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

register pressure = 3 on all edges
If swaps are not available, what can we conclude?

NP-complete if moves on entry/exit of basic blocks only.
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?
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.

Proof: change

\[
\begin{align*}
y_a &= b + x_{a,b} \\
x_a &= 1
\end{align*}
\]

into

\[
(x_a, y_a) = f(b, x_{a,b}).
\]
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 or such instructions have low register pressure (ex: function call, 64 bits load).
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 or such instructions have low register pressure (ex: function call, 64 bits load).

Polynomial if instructions have only one result!

Proof: greedy traversal (backwards and forwards) along control flow where register pressure = $k$. 
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 or such instructions have low register pressure (ex: function call, 64 bits load).

Polynomial if instructions have only one result!

So, NP-completeness did not disappear, it was simply not there! The proof of Chaitin et al. does not say anything about register allocation with live-range splitting and critical edge splitting.
On the complexity of register allocation

If moves are more suitable than loads and stores, it is in general easy to decide if some spilling is necessary or not.

**Spill test**

Chaitin (degree $\geq k$) $\Rightarrow$ Briggs (potential spill) $\Rightarrow$ Appel-George (iterated) $\Rightarrow$ Biased coloring $\Rightarrow$ Optimal test

But register allocation remains difficult:

- When critical edges cannot be split or code is not strict. But compilers often go through strict SSA and almost always split critical edges...
- Because optimal spilling is hard
- Because optimal coalescing is hard
Summary on register allocation complexity

- Complexity has to be considered with care: determining if spilling is necessary is *easier than one can think*.
- Interference graphs of SSA-form programs are chordal.
- Optimal register assignment in linear time (*tree scan*).
- Do not need to construct interference graph.
- Use live-range splitting to handle register constraints.
- Register allocator without iteration (i.e., 2 decoupled phases):

  ![Diagram]
  
  - Lower pressure → Color/coalesce → Φ-Implem.
If moves can be anywhere, the proof is broken.