Outline

1. Code representations
   - Control-flow graph
   - Loop-nesting forest
   - Static single assignment

2. Out-of-SSA translation
   - Translation with copy insertions: pitfalls and solution
   - Improving code quality and ease of implementation
   - Fast implementation with reduced memory footprint

3. SSA properties and liveness
   - Dominance, liveness, interferences, and chordal graphs
   - Construction of liveness sets in reducible CFGs for strict SSA
   - Extensions to irreducible CFGs and for checking liveness
How to coalesce variables?

Two alternatives

- Use a **working interference graph** where, in case of coalescing, the corresponding nodes are merged. $O(1)$ interference query.
- Manipulate **congruence classes**, i.e., sets of coalesced variables. Interferences must be tested between sets.

Chaitin, Sreedhar, Budimlić use congruence classes. Also useful to avoid interference graph. Naive algorithm: quadratic complexity.
## How to coalesce variables?

### Two alternatives

- Use a **working interference graph** where, in case of coalescing, the corresponding nodes are merged. $O(1)$ interference query.
- Manipulate **congruence classes**, i.e., sets of coalesced variables. Interferences must be tested between sets.

### Key properties for linear-complexity live range intersection

- 2 variables intersect if one is live at the definition of the other.
- In this case, the first definition dominates the second one.
- **Budimlić**: a set contains 2 intersecting variables if it contains a variable that intersects its “parent dominating” variable.

---

Chaitin, Sreedhar, Budimlić use congruence classes. Also useful to avoid interference graph. Naive algorithm: quadratic complexity.
Fast interference test for a set of variables

- Scan dominator tree in a depth-first search.
- Check interference with “parent dominating” variable.

```
b ← a + ···
c ← b + ···
c ← c
```
```
d ← ···
e ← d + ···
e ← a + e
```
Fast interference test for a set of variables

- Scan dominator tree in a depth-first search.
- Check interference with “parent dominating” variable.

```
  a ← ···
  b ← a + ···
  c ← b + ···
  c ← c

  d ← ···
  e ← d + ···
  e ← a + e
```

Alain Darte
Cours M2: Compilation avancée et optimisation de programmes
Fast interference test for a set of variables

- Scan dominator tree in a depth-first search.
- Check interference with “parent dominating” variable.

```
b ← a + ···
c ← b + ···
a ← c
```
```
d ← ···
e ← d + ···
a ← e
```

```
b
```
```
a
```
Fast interference test for a set of variables

- Scan dominator tree in a depth-first search.
- Check interference with “parent dominating” variable.
Scan dominator tree in a depth-first search.

Check interference with “parent dominating” variable.

\[\begin{align*}
  a &\leftarrow \cdots \\
  b &\leftarrow a + \cdots \\
  c &\leftarrow b + \cdots \\
  &\leftarrow c \\
  d &\leftarrow \cdots \\
  e &\leftarrow d + \cdots \\
  &\leftarrow a + e \\
\end{align*}\]
Fast interference test for a set of variables

- Scan dominator tree in a depth-first search.
- Check interference with “parent dominating” variable.
Fast interference test for a set of variables

- Scan dominator tree in a depth-first search.
- Check interference with “parent dominating” variable.
Fast interference test for a set of variables

- Scan dominator tree in a depth-first search.
- Check interference with “parent dominating” variable.

```
\begin{align*}
  b & \leftarrow a + \cdots \\
  c & \leftarrow b + \cdots \\
  \quad & \leftarrow c \\
\end{align*}
\begin{align*}
  d & \leftarrow \cdots \\
  e & \leftarrow d + \cdots \\
  \quad & \leftarrow a + e \\
\end{align*}
```
Algorithm 2: Check intersection in a set of variables

Data: list sorted according to a pre-DFS order of the dominance tree
Output: Returns TRUE if the list contains an interference

1. \( \text{dom} \leftarrow \text{empty \_ stack} ; i \leftarrow 0 ; \) /* stack of the traversal */
2. while \( i < \text{list\_size()} \) do
   3. \( \text{current} \leftarrow \text{list}(i++) ; \)
   4. \( \text{other} \leftarrow \text{dom\_top()} ; \) /* NULL if dom is empty */
   5. while \((\text{other} \neq \text{NULL}) \land \text{dominate}\text{(other, current)} = \text{FALSE}) \) do
      6. \( \text{dom}\_\text{pop}() ; \) /* not the desired parent, remove */
      7. \( \text{other} \leftarrow \text{dom\_top()} ; \) /* consider next one */
   8. \( \text{parent} \leftarrow \text{other} ; \)
   9. if \((\text{parent} \neq \text{NULL}) \land (\text{intersect}\text{(current, parent)} = \text{TRUE}) \) then
      10. \( \text{return} \text{TRUE} ; \) /* intersection detected */
      11. \( \text{dom}\_\text{push}(\text{current}) ; \) /* otherwise, keep checking */
12. \( \text{return} \text{FALSE} ; \)
Linear interference test of two congruence classes

Generalization to interference test of two sets

- Emulate a stack-based DFS traversal of dominance tree, for two sorted sets instead of one linear number of tests.

No need to test intersection of variables in the same set. Take values into account for value-based interference: need links of "equal ancestors", which may increase complexity. Sort in linear time the resulting set, in case of coalescing.

Fewer intersection tests possible now to use more expensive queries for intersection/liveness and to avoid interference graph: Budimlić intersection test, still using liveness sets. Fast liveness checking of Boissinot et al. (CGO'08).
Linear interference test of two congruence classes

Generalization to interference test of two sets

- Emulate a stack-based DFS traversal of dominance tree, for two sorted sets instead of one.

```plaintext
1  i_r ← 0 ; i_b ← 0 ;
2  while (i_r < red.size() and i_b < blue.size()) do
3      if blue(i_b) ≺ red(i_r) then current ← blue(i_b++) else current ← red(i_r++)
4  while(i_r < red.size() and n_b > 0) do current ← red(i_r++) /* still n_b blue in stack */
5  while(i_b < blue.size() and n_r > 0) do current ← blue(i_b++) /* still n_r red in stack */
```

No need to test intersection of variables in the same set.
Take values into account for value-based interference: need links of "equal ancestors", which may increase complexity.
Sort in linear time the resulting set, in case of coalescing.
Fewer intersection tests possible now to use more expensive queries for intersection/liveness and to avoid interference graph: Budimlić intersection test, still using liveness sets.
Fast liveness checking of Boissinot et al. (CGO'08).

Alain Darte
Cours M2: Compilation avancée et optimisation de programmes
Linear interference test of two congruence classes

Generalization to interference test of two sets

- Emulate a stack-based DFS traversal of dominance tree, for two sorted sets instead of one (linear number of tests).
- No need to test intersection of variables in the same set.
- Take values into account for value-based interference: need links of “equal ancestors”, which may increase complexity.
- Sort in linear time the resulting set, in case of coalescing.
Linear interference test of two congruence classes

Generalization to interference test of two sets

- Emulate a stack-based DFS traversal of dominance tree, for two sorted sets instead of one linear number of tests.
- No need to test intersection of variables in the same set.
- Take values into account for value-based interference: need links of “equal ancestors”, which may increase complexity.
- Sort in linear time the resulting set, in case of coalescing.

Fewer intersection tests possible now to use more expensive queries for intersection/liveness and to avoid interference graph:

- Budimlić intersection test, still using liveness sets.
- Fast liveness checking of Boissinot et al. (CGO’08).
Algorithm 3: interference(a, b)

Data: A variable a and its parent b in the dominance tree

Output: Returns TRUE if a interferes (i.e., intersects and has a different value) with an already-visited variable. Also, update equal_anc information

/* a and b are assumed to not be equal to NULL */

1 equal_anc_out(a) ← NULL ; /* initialization */
2 if a and b are in the same set then
3    b ← equal_anc_out(b) ; /* check/update in other set */
4  if value(a) ≠ value(b) then
5     return chain_intersect(a, b) ; /* check with b and its equal intersecting ancestors in the other set */
6    else
7     update_equal_anc_out(a, b) ; /* update equal intersecting ancestor going up in the other set */
8     return FALSE ; /* no interference */
Algorithm 4: update_equal_anc_out(a, b)

Data: Variables $a$ and $b$, same value, but in different sets
Output: Set nearest intersecting ancestor of $a$, in other set, with same value (NULL if does not exist)

1. $\text{tmp} \leftarrow b$
2. while $(\text{tmp} \neq \text{NULL})$ and $(\text{intersect}(a, \text{tmp}) = \text{FALSE})$ do
   3. $\text{tmp} \leftarrow \text{equal_anc_in}($\text{tmp}$)$ ; /* follow the chain of equal intersecting ancestors in the other set */
3. $\text{equal_anc_out}(a) \leftarrow \text{tmp}$ ; /* $\text{tmp}$ intersects $a$ or NULL */

Algorithm 5: chain_intersect(a, b)

Data: Variables $a$ and $b$, different value, in different sets
Output: Returns TRUE if $a$ intersects $b$ or one of its equal intersecting ancestors in the same set

1. $\text{tmp} \leftarrow b$
2. while $(\text{tmp} \neq \text{NULL})$ and $(\text{intersect}(a, \text{tmp}) = \text{FALSE})$ do
   3. $\text{tmp} \leftarrow \text{equal_anc_in}($\text{tmp}$)$ ; /* follow the chain of equal intersecting ancestors */
3. if $\text{tmp} = \text{NULL}$ then return FALSE else return TRUE ;
Speed-up for SPEC CINT2000: x2

General scheme
- Sreedhar III: w. virtualization.
- Us I, Us III: our proposal, w.o./w. virtualization.

Interference checks
- Default: liveness sets + interference graph.
- InterCheck: Budimlić with liveness sets.
- LiveCheck: Fast liveness checking.
- Linear: Linear check instead of quadratic.
Memory footprint reduction for SPEC CINT2000: x10

- Interference graph: half-size bit matrix.

Data structures grow during virtualization. “Perfect memory” evaluated, with both enumerated/bit sets for liveness sets.

### Sum of memory footprint

![Graph showing sum of memory footprint](image)

### Max of memory footprint

![Graph showing max of memory footprint](image)
General framework

- Correctness clarified even for complex cases
- Two-phases solution, based on coalescing

Results

- Value-based interferences, for free, as good as Sreedhar III
- Fast algorithm: Speed-up x2, memory reduction x10.

Implementation

- No need to virtualize (at least for us)
- Simpler implementation
Outline

1. Code representations
   - Control-flow graph
   - Loop-nesting forest
   - Static single assignment

2. Out-of-SSA translation
   - Translation with copy insertions: pitfalls and solution
   - Improving code quality and ease of implementation
   - Fast implementation with reduced memory footprint

3. SSA properties and liveness
   - Dominance, liveness, interferences, and chordal graphs
   - Construction of liveness sets in reducible CFGs for strict SSA
   - Extensions to irreducible CFGs and for checking liveness
A variable $v$ is live-in at program point $p$ if there is a path, not containing the definition of $v$, from $q$ to a use of $v$.

Each instruction $\ell$, where $v$ is live, is dominated by $\text{def}(v)$ the definition point of $v$: $\text{def}(v) \succeq \ell$. 

### Proof

If $\ell$ is not dominated by $\text{def}(v)$, there is a path from start to $\ell$ that does not visit $\text{def}(v)$. Thus, there is a path from start to a use of $v$ that does not visit $\text{def}(v)$.
A variable $v$ is live(-in) at program point $p$ if there is a path, not containing the definition of $v$, from $q$ to a use of $v$.

Each instruction $\ell$, where $v$ is live, is dominated by $\text{def}(v)$ the definition point of $v$: $\text{def}(v) \succeq \ell$.

**Proof:** if $\ell$ is not dominated by $\text{def}(v)$

- there is a path from $\text{start}$ to $\ell$ that does not visit $\text{def}(v)$.
- $v$ is live at $\ell$: there is a path from $\ell$ to a use of $v$ that does not visit $\text{def}(v)$.
- Thus, there is a path from $\text{start}$ to a use of $v$ that does not visit $\text{def}(v)$. 
A variable $v$ is live(-in) at program point $p$ if there is a path, not containing the definition of $v$, from $q$ to a use of $v$.

Each instruction $\ell$, where $v$ is live, is dominated by $\text{def}(v)$ the definition point of $v$: $\text{def}(v) \succeq \ell$.

Proof: if $\ell$ is not dominated by $\text{def}(v)$

- there is a path from $\text{start}$ to $\ell$ that does not visit $\text{def}(v)$.
- $v$ is live at $\ell$: there is a path from $\ell$ to a use of $v$ that does not visit $\text{def}(v)$.
- Thus, there is a path from $\text{start}$ to a use of $v$ that does not visit $\text{def}(v)$.

No: each use of $v$ is dominated by $\text{def}(v)$. 
Assume that $v$ and $w$ are both live at some instruction $\ell$.

Then, $\text{def}(v) \succeq \ell$ and $\text{def}(w) \succeq \ell$.

Dominance = tree:

- either $\text{def}(v) \succeq \text{def}(w)$ (and, in this case, $v$ is live at $\text{def}(w)$);
- or $\text{def}(w) \succeq \text{def}(v)$ (and, in this case, $w$ is live at $\text{def}(v)$).

Interference can be directed according to dominance.
Dominance, liveness, and interference

- Assume that $v$ and $w$ are both live at some instruction $\ell$.
- Then, $\text{def}(v) \succeq \ell$ and $\text{def}(w) \succeq \ell$.
- Dominance $=$ tree:
  - either $\text{def}(v) \succeq \text{def}(w)$ (and, in this case, $v$ is live at $\text{def}(w)$);
  - or $\text{def}(w) \succeq \text{def}(v)$ (and, in this case, $w$ is live at $\text{def}(v)$).
- Interference can be directed according to dominance.

Consequences

- Strictness implies two equivalent notions of interferences:
  - live ranges intersect;
  - one variable is live at the definition of the other.
- Assume no equality among intersecting variables: then, the interference graph of an SSA program is chordal/triangulated.
Intersecting live ranges, subtrees of a tree

- Assume \( v \) \text{dom} \( w \)

- Then, \( v \) is live at \( \text{def}(w) \)
Intersecting live ranges, subtrees of a tree

Assume $v \xrightarrow{\text{dom}} w$

Then, $v$ is live at $\text{def}(w)$

- Both live at $\text{def}(w)$
- Dominance subtree of $v$
- Live ranges of variables can be represented as subtrees of the dominance tree
- Intersection graph = chordal graph.
- Other proof: no chordless cycle
- Consider a cycle in the interference graph. There must be three vertices $u, v, w$, such that:
  - $v \succ w \succ u$
  - Both live at $\text{def}(w)$
  - They thus interfere (chord).
Intersecting live ranges, subtrees of a tree

- Assume \( v \) \( \text{dom} \) \( w \)
- Then, \( v \) is live at \( \text{def}(w) \)

Live ranges of variables can be represented as subtrees of the dominance tree \( \triangleright \) intersection graph = chordal graph.

Assume \( v \) \( \text{dom} \) \( w \)
Then, \( v \) is live at \( \text{def}(w) \)

Live ranges of variables can be represented as subtrees of the dominance tree \( \triangleright \) intersection graph = chordal graph.

Consider a cycle in the interference graph. There must be three vertices \( u, v, w \), such that:

\[ \preceq, \preceq, \preceq \]

\( v \) \( w \) \( u \)
and \( v \) are both live at \( \text{def}(w) \).

They thus interfere (chord).
Intersecting live ranges, subtrees of a tree

- Assume $v \xrightarrow{\text{dom}} w$
- Then, $v$ is live at $\text{def}(w)$

Live ranges of variables can be represented as subtrees of the dominance tree. Intersection graph = chordal graph.

Other proof: no chordless cycle

Consider a cycle in the interference graph. There must be three vertices $u, v, w$, such that:

- $v \xleftarrow{} w$
- $v$ and $w$ are both live at $\text{def}(w)$.
- They thus interfere (chord).
Intersecting live ranges, subtrees of a tree

- Assume $v \xrightarrow{\text{dom}} w$
- Then, $v$ is live at $\text{def}(w)$

Live ranges of variables can be represented as subtrees of the dominance tree intersection graph = chordal graph.

Other proof: no chordless cycle
Consider a cycle in the interference graph. There must be three vertices $u$, $v$, $w$, such that:

$u$ and $v$ are both live at $\text{def}(w)$. 

Intersecting live ranges, subtrees of a tree

- Assume \( v \xrightarrow{\text{dom}} w \)
- Then, \( v \) is live at \( \text{def}(w) \)

Live ranges of variables can be represented as subtrees of the dominance tree \( \Rightarrow \) intersection graph = chordal graph.

Other proof: no chordless cycle

Consider a cycle in the interference graph. There must be three vertices \( u, v, w \), such that:

\[
\{\geq, \\leq\}
\]

\( u \) and \( v \) are both live at \( \text{def}(w) \). They thus interfere (chord).
SSA versus non-SSA interference graphs

Program          Live Ranges

\[ a \leftarrow \cdots \]          \[ a \]
\[ b \leftarrow \cdots \]          \[ b \]
\[ c \leftarrow \cdots \]          \[ c \]
\[ d \leftarrow a + b \]          \[ d \]
\[ e \leftarrow c + 1 \]          \[ e \]

How can we create a 4-cycle \( \{a, c, d, e\} \)?
SSA versus non-SSA interference graphs

Program

\[ a \leftarrow \cdots \]
\[ b \leftarrow \cdots \]
\[ c \leftarrow \cdots \]
\[ d \leftarrow a + b \]
\[ e \leftarrow c + 1 \]
\[ a \leftarrow \cdots \]

Live Ranges

\[ a \]
\[ b \]
\[ c \]
\[ d \]
\[ e \]
\[ a \]

Interference Graph

- How can we create a 4-cycle \( \{a, c, d, e\} \)?
- Redefine \( a \Rightarrow \) SSA violated!
SSA versus non-SSA interference graphs

Program and live ranges

Interference Graph

a ← ···
d ← ···
e ← a + ···
   ← d
b ← ···
c ← a + ···
e ← b
   ← c
d

Alain Darte
Cours M2: Compilation avancée et optimisation de programmes
SSA versus non-SSA interference graphs

Program and live ranges

Interference Graph

\[
\begin{align*}
    a & \leftarrow \cdots \\
    d & \leftarrow \cdots \\
    e_1 & \leftarrow a + \cdots \\n    & \leftarrow d \\
    e_3 & \leftarrow \phi(e_1, e_2) \\
    b & \leftarrow \cdots \\
    c & \leftarrow a + \cdots \\
    e_2 & \leftarrow b \\
    & \leftarrow c \\
    d & \leftarrow e_1 \\
    b & \leftarrow e_2 \\
    c & \leftarrow e_3
\end{align*}
\]
Chordal $k$-colorable: greedy-$k$-colorable & tree-scan

- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-$k$-colorable: all vertices can be successively “simplified” ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.
Chordal $k$-colorable: greedy-$k$-colorable & tree-scan

If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.

Greedy-$k$-colorable: all vertices can be successively “simplified” ($d^\circ < k$).

A post order walk of the dominance tree gives such an elimination order.
Chordal $k$-colorable: greedy-$k$-colorable & tree-scan

- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-$k$-colorable: all vertices can be successively “simplified” ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.
Chordal $k$-colorable: greedy-$k$-colorable & tree-scan

- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-$k$-colorable: all vertices can be successively “simplified” ($d^\circ < k$).
- A post order walk of the dominance tree gives such an elimination order.
Chordal $k$-colorable: greedy-$k$-colorable & tree-scan

- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-$k$-colorable: all vertices can be successively “simplified” ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.
Chordal $k$-colorable: greedy-$k$-colorable & tree-scan

- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-$k$-colorable: all vertices can be successively “simplified” ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.
Chordal $k$-colorable: greedy-$k$-colorable & tree-scan

- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-$k$-colorable: all vertices can be successively “simplified” ($d^\circ < k$).
- A post order walk of the dominance tree gives such an elimination order.
Chordal $k$-colorable: greedy-$k$-colorable & tree-scan

If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.

- Greedy-$k$-colorable: all vertices can be successively “simplified” ($d^\circ < k$).
- A post order walk of the dominance tree gives such an elimination order.
- A pre order walk of the dominance tree directly yields a coloring sequence.
- No need to build the interference graph itself.

- $a \gets \cdots$
- $d \gets \cdots$
- $e_1 \gets a + \cdots$
- $\leftarrow d$
- $b \gets \cdots$
- $c \gets a + \cdots$
- $\leftarrow c$
- $e_2 \gets b$
- $e_3 \leftarrow \phi(e_1, e_2)$
- $\begin{align*}
  &a \\
  &d \\
  &e_1 \\
  &\leftarrow \ \\
  &e_2 \\
  &\leftarrow \ \\
  &e_3
\end{align*}$
Chordal $k$-colorable: greedy-$k$-colorable & tree-scan

- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-$k$-colorable: all vertices can be successively “simplified” ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.
- A pre order walk of the dominance tree directly yields a coloring sequence.
- No need to build the interference graph itself.
Chordal $k$-colorable: greedy-$k$-colorable & tree-scan

- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-$k$-colorable: all vertices can be successively “simplified” ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.
- A pre order walk of the dominance tree directly yields a coloring sequence.
- No need to build the interference graph itself.
Chordal $k$-colorable: greedy-$k$-colorable & tree-scan

If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
Greedy-$k$-colorable: all vertices can be successively “simplified” ($d^o < k$).
A post order walk of the dominance tree gives such an elimination order.
A pre order walk of the dominance tree directly yields a coloring sequence.
No need to build the interference graph itself.
Chordal $k$-colorable: greedy-$k$-colorable & tree-scan

- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-$k$-colorable: all vertices can be successively “simplified” ($d^\circ < k$).
- A post order walk of the dominance tree gives such an elimination order.
- A pre order walk of the dominance tree directly yields a coloring sequence.
- No need to build the interference graph itself.
Chordal $k$-colorable: greedy-$k$-colorable & tree-scan

If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.

- Greedy-$k$-colorable: all vertices can be successively “simplified” ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.
- A pre order walk of the dominance tree directly yields a coloring sequence.
- No need to build the interference graph itself.
Chordal $k$-colorable: greedy-$k$-colorable & tree-scan

- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-$k$-colorable: all vertices can be successively “simplified” ($d^\circ < k$).
- A post order walk of the dominance tree gives such an elimination order.
- A pre order walk of the dominance tree directly yields a coloring sequence.
- No need to build the interference graph itself.
Chordal $k$-colorable: greedy-$k$-colorable & tree-scan

- If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
- Greedy-$k$-colorable: all vertices can be successively “simplified” ($d^o < k$).
- A post order walk of the dominance tree gives such an elimination order.
- A pre order walk of the dominance tree directly yields a coloring sequence.
- No need to build the interference graph itself.

If register pressure $\leq k$, no spill is necessary. Here only 2 registers needed.
Greedy-$k$-colorable: all vertices can be successively “simplified” ($d^o < k$).
A post order walk of the dominance tree gives such an elimination order.
A pre order walk of the dominance tree directly yields a coloring sequence.
No need to build the interference graph itself.
Outline

1. Code representations
   - Control-flow graph
   - Loop-nesting forest
   - Static single assignment

2. Out-of-SSA translation
   - Translation with copy insertions: pitfalls and solution
   - Improving code quality and ease of implementation
   - Fast implementation with reduced memory footprint

3. SSA properties and liveness
   - Dominance, liveness, interferences, and chordal graphs
   - Construction of liveness sets in reducible CFGs for strict SSA
   - Extensions to irreducible CFGs and for checking liveness
Traditional fixed-point data-flow analysis

Equations

\[
\text{LiveIn}(B) = \text{PhiDefs}(B) \cup \text{UpwardExposed}(B) \cup (\text{LiveOut}(B) \setminus \text{Defs}(B))
\]

\[
\text{LiveOut}(B) = \bigcup_{S \in \text{succs}(B)} (\text{LiveIn}(S) \setminus \text{PhiDefs}(S)) \cup \text{PhiUses}(B)
\]

- \text{PhiDefs}(B): variables defined by \(\phi\)-operations at entry of \(B\).
- \text{PhiUses}(B): used by \(\phi\)-operations at a successor block of \(B\).
- \text{UpwardExposed}(B): used in \(B\) but not defined earlier in \(B\).
Traditional fixed-point data-flow analysis

Equations

\[
\text{LiveIn}(B) = \Phi\text{Defs}(B) \cup \text{UpwardExposed}(B) \cup (\text{LiveOut}(B) \setminus \text{Defs}(B))
\]

\[
\text{LiveOut}(B) = \bigcup_{S \in \text{succs}(B)} (\text{LiveIn}(S) \setminus \Phi\text{Defs}(S)) \cup \Phi\text{Uses}(B)
\]

- \(\Phi\text{Defs}(B)\): variables defined by \(\phi\)-operations at entry of \(B\).
- \(\Phi\text{Uses}(B)\): used by \(\phi\)-operations at a successor block of \(B\).
- \(\text{UpwardExposed}(B)\): used in \(B\) but not defined earlier in \(B\).

Complexity

- \(W\): non-local variables (i.e., not fully in a block), \(P\): program.
- \(G = (V, E)\): CFG with \(|V| - 1 \leq |E| \leq |V|^2\).

\(O(|P|) + O(|W|) \times \text{number of iterations, i.e., } O(|E||W|)\) for worklist algorithms and \(O(|E|(d(G, T) + 3))\) for round robin.

\(d(G, T)\): max. number of back edges (for a DFS tree \(T\)), in a cycle-free path of \(G\).
Exploiting loop structure

- \( G = (V, E) \): reducible CFG with strict SSA.
- \( \mathcal{F}_L(G) \): DAG obtained by removing loop-edges.

Bad case for iterative data-flow analysis:

Principles to avoid iteration:

- Compute liveness information, traversing \( \mathcal{F}_L(G) \) bottom-up.
- Refine liveness by exploiting loop structure.
Key lemmas related to loop structure

Lemma 1

Let $G$ be a reducible CFG, $\nu$ an SSA variable, and $d$ its definition. If $L$ is a maximal loop not containing $d$, then $\nu$ is live-in at the loop-header $h$ of $L$ iff there is a path in $F_L(G)$, not containing $d$, from $h$ to a use of $\nu$.

Lemma 2

Let $G$ be a reducible CFG, $\nu$ an SSA variable, and $d$ its definition. Let $p$ be a node of $G$ such that all loops containing $p$ also contain $d$. Then $\nu$ is live-in at $p$ iff there is a path in $F_L(G)$, from $p$ to a use of $\nu$, not containing $d$. 
Key lemmas related to loop structure (cont’d)

- Propagating liveness along $\mathcal{F}_L(G)$ can only mark live-in variables that are indeed live-in.
- If, after this propagation, $v$ is missing at $p$, $p$ belongs to a loop that does not contain the definition of $v$ (Lemma 2).
- If $L$ is such a maximal loop, $v$ is correctly marked as live-in at the header of $L$ (Lemma 1).

Lemma 3

Consider $L$ a loop and $v$ an SSA variable. If $v$ is live-in at the loop-header of $L$, it is live-in and live-out at every node in $L$.

Propagating inside loops is enough to patch the liveness sets.
Partial liveness, with postorder traversal

Algorithm 6: DAG_DFS(block B)

1. for each \( S \in \text{succs}(B) \) if \((B, S)\) is not a loop-edge do
2.     if \( S \) is unprocessed then
3.         DAG_DFS(S)
4.     \( \text{Live} = \text{PhiUses}(B) \) /* used by \( \phi \)-functions in B’s successors */
5. for each \( S \in \text{succs}(B) \) if \((B, S)\) is not a loop-edge do
6.      \( \text{Live} = \text{Live} \cup (\text{LiveIn}(S) \setminus \text{PhiDefs}(S)) \)
7. \( \text{LiveOut}(B) = \text{Live} \);
8. for each program point \( p \) in \( B \), backward do
9.      remove variables defined at \( p \) from \( \text{Live} \);
10.     add uses at \( p \) in \( \text{Live} \)
11. \( \text{LiveIn}(B) = \text{Live} \cup \text{PhiDefs}(B) \);  
12. mark \( B \) as processed
Propagate live variables within loop bodies

**Algorithm 7:** LoopTree\_DFS(node $N$ of the loop forest)

1. **if** $N$ is a loop node **then**
2. \hspace{1em} Let $B_N = \text{Block}(N)$ /* the loop-header of $N$ */
3. \hspace{1em} Let $\text{LiveLoop} = \text{LiveIn}(B_N) \setminus \Phi\text{Defs}(B_N)$;
4. \hspace{1em} **for** each $M \in \text{LoopTree\_children}(N)$ **do**
5. \hspace{2em} Let $B_M = \text{Block}(M)$ /* loop-header or block */
6. \hspace{2em} $\text{LiveIn}(B_M) = \text{LiveIn}(B_M) \cup \text{LiveLoop}$;
7. \hspace{2em} $\text{LiveOut}(B_M) = \text{LiveOut}(B_M) \cup \text{LiveLoop}$;
8. \hspace{2em} LoopTree\_DFS($M$)

**Algorithm 8:** Compute\_LiveSets\_SSA\_Reducible(CFG)

1. **for** each basic block $B$ **do**
2. \hspace{1em} mark $B$ as unprocessed
3. $\text{DAG\_DFS}(R)$ /* $R$ is the CFG root node */
4. **for** each root node $L$ of the loop-nesting forest **do**
5. \hspace{1em} LoopTree\_DFS($L$)
Transformation of an irreducible CFG into a reducible one

\[ E' = E \setminus \text{LoopEdges}(L) \setminus \text{EntryEdges}(L) \cup \{(s, \delta_L) \mid s \in \text{PreEntries}(L)\} \]
\[ \cup \{(s, \delta_L) \mid \exists (s, h) \in \text{LoopEdges}(L)\} \cup \{(\delta_L, h) \mid h \in \text{LoopHeaders}(L)\} \]

G: Irreducible

ψ_L(G): Reducible
Key results to analyze liveness in irreducible CFGs

Lemma 4

If \( d \) dominates \( u \) in \( G \), \( d \) dominates \( u \) in \( \Psi_L(G) \).

Theorem 5

Let \( v \) be an SSA variable, \( G \) a CFG, transformed into \( \Psi_L(G) \) when considering a loop \( L \) of a loop forest of \( G \). Then, for each node \( q \) of \( G \), \( v \) is live-in (resp. live-out) at \( q \) in \( G \) iff \( v \) is live-in (resp. live-out) at \( q \) in \( \Psi_L(G) \).

\( \text{HnCA}(B,S) \): highest non common ancestor (in the loop forest) of \( B \) and \( S \), i.e., highest ancestor of \( S \) that is not ancestor of \( B \).
Algorithm 9: DAG_DFS(block B) /* if loops have one header */

1 for each $S \in \text{succs}(B)$ if $(B, S)$ is not a loop-edge do
2   if $S$ is unprocessed then
3     $T = \text{HnCA}(B, S)$;
4     $\text{DAG_DFS}(T)$
5 $Live = \Phi\text{Uses}(B)$ /* used by $\phi$-functions in $B$’s successors */
6 for each $S \in \text{succs}(B)$ if $(B, S)$ is not a loop-edge do
7     $T = \text{HnCA}(B, S)$;
8     $Live = Live \cup (\text{LiveIn}(T) \setminus \Phi\text{Defs}(T))$
9 $\text{LiveOut}(B) = Live$
10 for each program point $p$ in $B$, backward do
11    remove variables defined at $p$ from $Live$
12    add uses at $p$ in $Live$
13 $\text{LiveIn}(B) = Live \cup \Phi\text{Defs}(B)$ ;
14 mark $B$ as processed
Experimental results

Speed-up w.r.t. iterative data-flow, unoptimized programs, bitsets.

Speed-up w.r.t. iterative data-flow, optimized programs, bitsets.
Experimental results

Speed-up w.r.t iterative data-flow, for optimized programs, with bitsets.

Ratio of the different phases in the forest-based algorithm (forward & backward passes, computation of PhiUses & PhiDefs sets, initialization), bitsets, unoptimized & optimized programs.