Revisiting Out-of-SSA Translation for Correctness, Code Quality, and Efficiency

For both Static and JIT Compilation

Benoit Boissinot (LIP), **Alain Darte** (LIP), Benoît Dupont de Dinechin (STMicro), Christophe Guillon (STMicro), Fabrice Rastello (LIP)

Compsys Team
Laboratoire de l'Informatique du Parallélisme (LIP)
École normale supérieure de Lyon

SSA Seminar, April 27, 2009, Autrans

Context

Collaboration between Compsys team and STMicroelectronics compiler group.

- Back-end compiler: Open64 + LAO (linear assembly optimizer) with use of SSA and ψ-SSA.
- Double role: both static and just-in-time compilation for STMicro processor family.
- Applications: multimedia applications.
- Optimization objectives: compiler robustness, ease of implementation, portability, code quality, speed of compiler.
- Other collaboration themes: instruction cache optimization, register allocation.

Outline

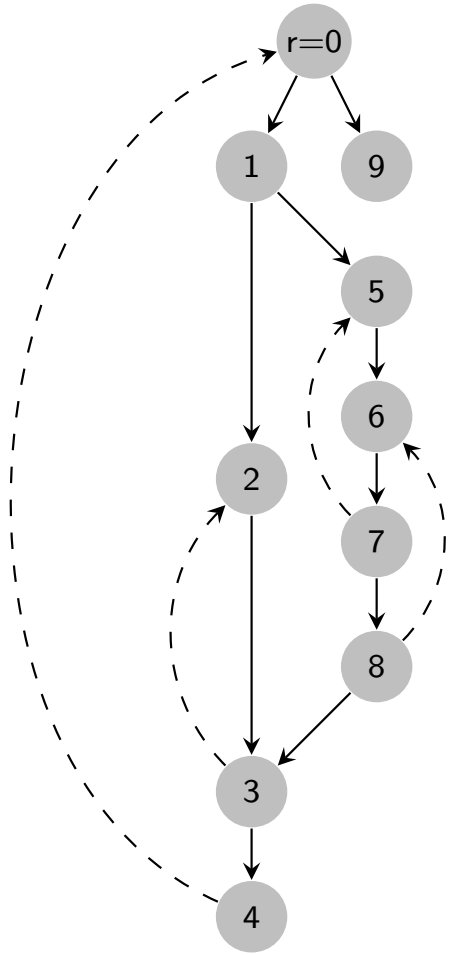
Dominance

Control Flow Graph

- one entry node r;
- every node reachable from r.

Definition (dominance)

a dominates b if every path from the root r to b contains a.



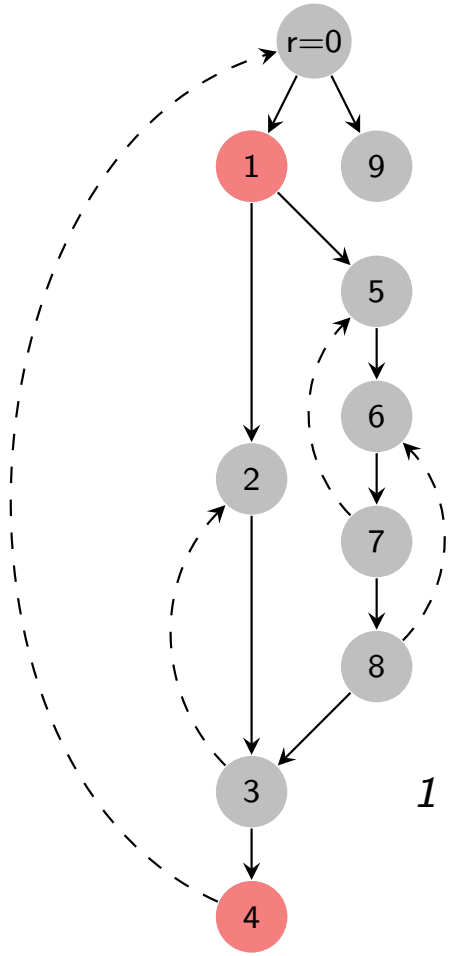
Dominance

Control Flow Graph

- one entry node r;
- every node reachable from r.

Definition (dominance)

a dominates b if every path from the root r to b contains a.



1 dominates 4?

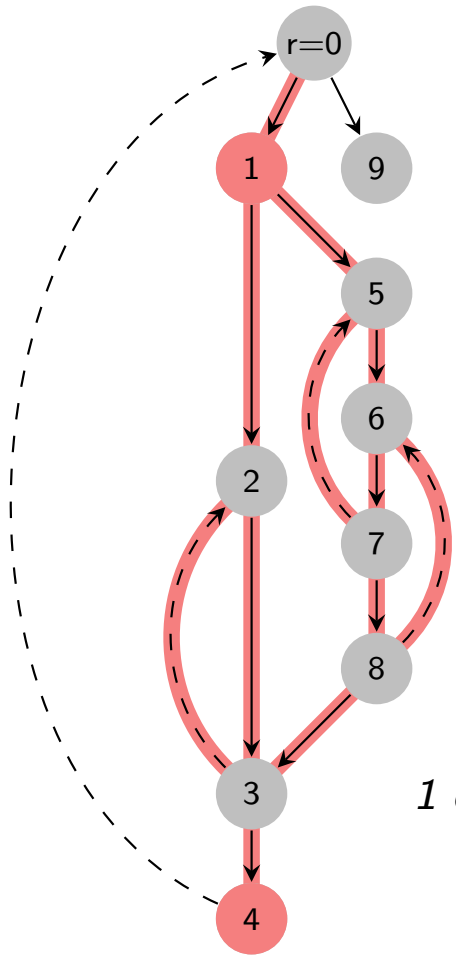
Dominance

Control Flow Graph

- one entry node r;
- every node reachable from r.

Definition (dominance)

a dominates b if every path from the root r to b contains a.



1 dominates 4? YES

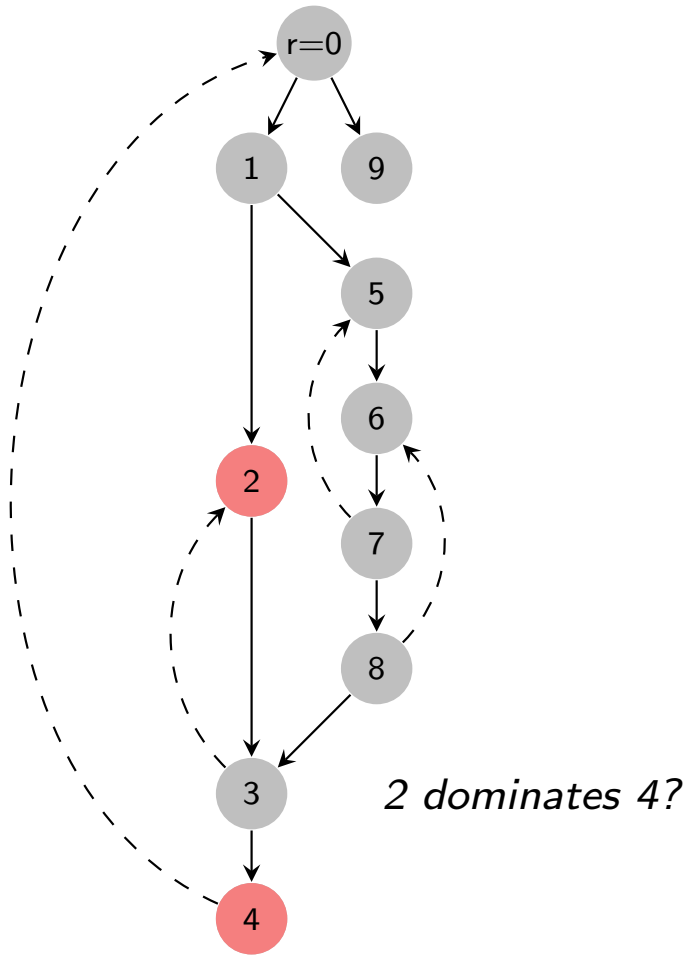
Dominance

Control Flow Graph

- one entry node r;
- every node reachable from r.

Definition (dominance)

a dominates b if every path from the root r to b contains a.



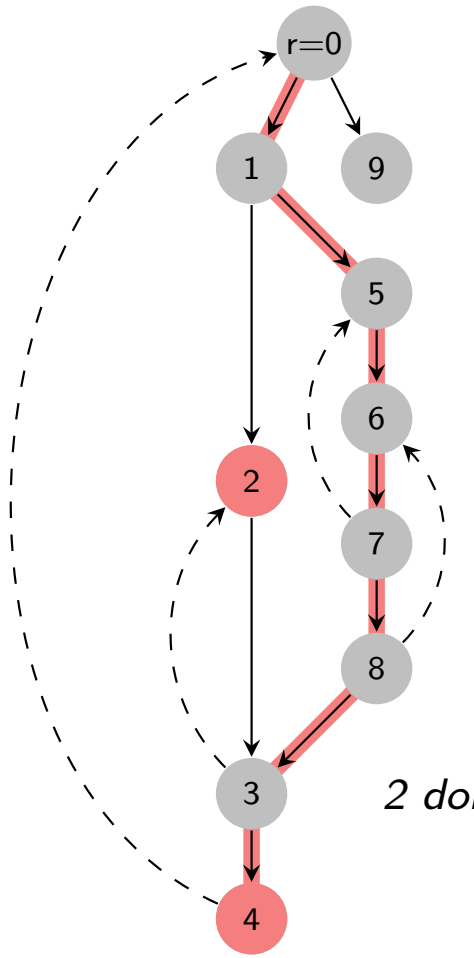
Dominance

Control Flow Graph

- one entry node r;
- every node reachable from r.

Definition (dominance)

a dominates b if every path from the root r to b contains a.



2 dominates 4? NO

Dominance

Control Flow Graph

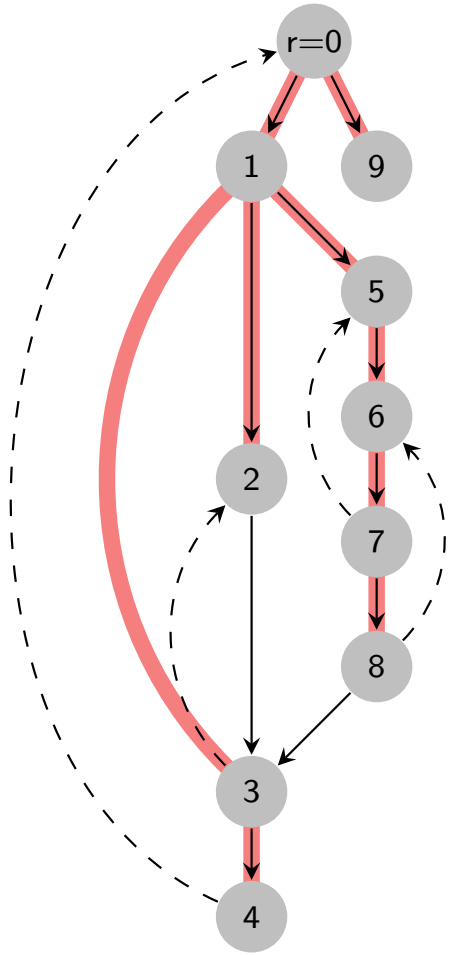
- one entry node r;
- every node reachable from r.

Definition (dominance)

a dominates b if every path from the root r to b contains a.

Property

The dominance relation induces a tree. ☛ With classical tree labeling, testing if a dominates b is an $O(1)$ operation.



Static single assignment (SSA)

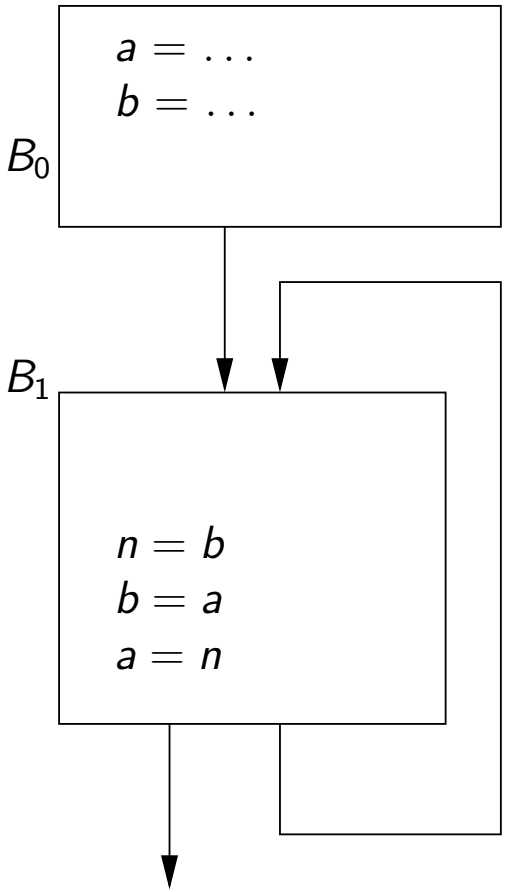
SSA with dominance property

- Unique definition for each variable;
- Each definition dominates its uses.

Static single assignment (SSA)

SSA with dominance property

- Unique definition for each variable;
- Each definition dominates its uses.



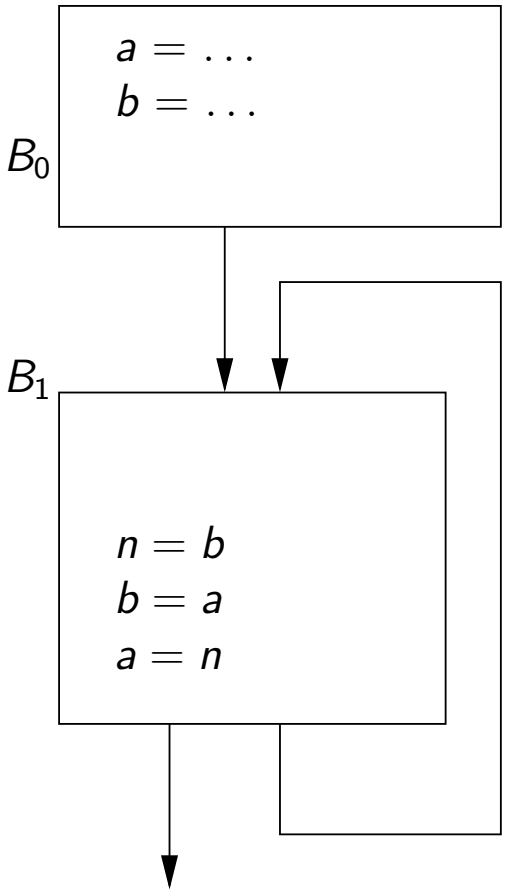
Static single assignment (SSA)

SSA with dominance property

- Unique definition for each variable;
- Each definition dominates its uses.

Conversion into SSA

- Need to introduce ϕ-functions at dominance frontier.



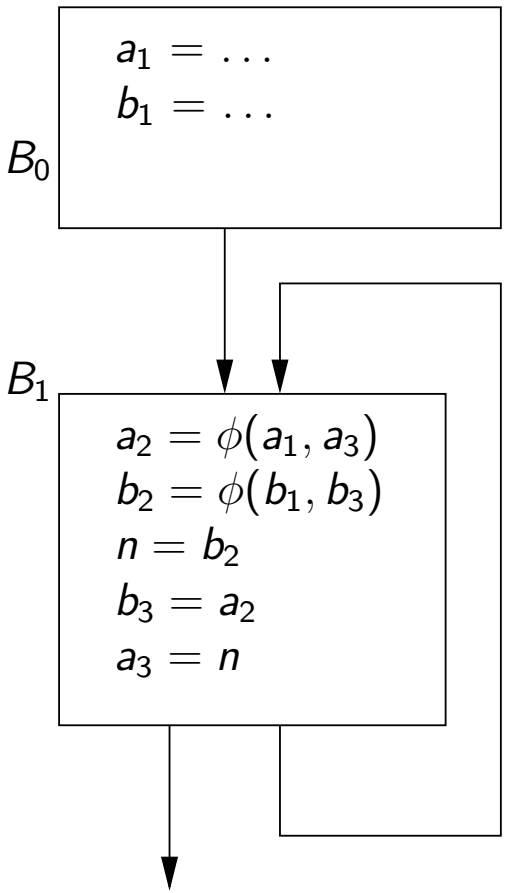
Static single assignment (SSA)

SSA with dominance property

- Unique definition for each variable;
- Each definition dominates its uses.

Conversion into SSA

- Need to introduce ϕ-functions at dominance frontier.



Static single assignment (SSA)

SSA with dominance property

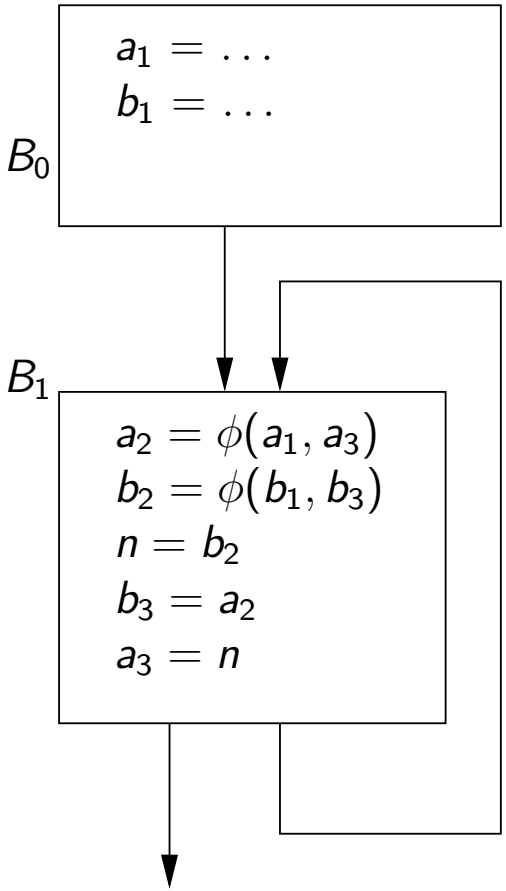
- Unique definition for each variable;
- Each definition dominates its uses.

Conversion into SSA

- Need to introduce ϕ-functions at dominance frontier.

Interests of SSA

- Code optimizations: efficient, easy-to-implement, fast;
- Two-phases register allocation;
- Program analysis/verification.



Static single assignment (SSA)

SSA with dominance property

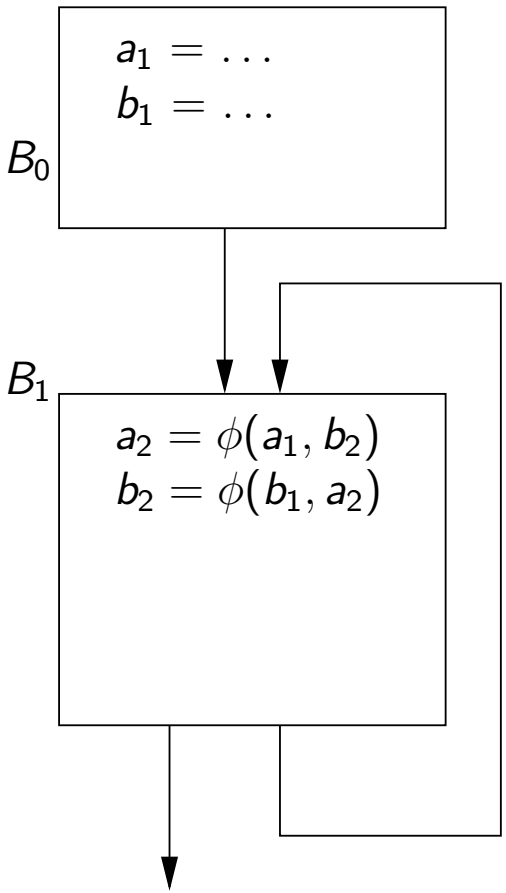
- Unique definition for each variable;
- Each definition dominates its uses.

Conversion into SSA

- Need to introduce ϕ-functions at dominance frontier.

Interests of SSA

- Code optimizations: efficient, easy-to-implement, fast;
- Two-phases register allocation;
- Program analysis/verification.



Basic dominance and SSA properties for interferences

SSA with dominance property and live range intersection

If two variables are simultaneously live at a given program point, then the definition of one dominates the definition of the other.

Basic dominance and SSA properties for interferences

SSA with dominance property and live range intersection

If two variables are simultaneously live at a given program point, then the definition of one dominates the definition of the other (also the first is live at the definition of the second). ☛ We will use this to replace interference graph by queries for intersection check.

Basic dominance and SSA properties for interferences

SSA with dominance property and live range intersection

If two variables are simultaneously live at a given program point, then the definition of one dominates the definition of the other (also the first is live at the definition of the second). ☛ We will use this to replace interference graph by queries for intersection check.

Chordal interference graph

SSA live-ranges $\equiv$ subtrees of the dominance tree ☛ chordal.

Basic dominance and SSA properties for interferences

SSA with dominance property and live range intersection

If two variables are simultaneously live at a given program point, then the definition of one dominates the definition of the other (also the first is live at the definition of the second). ☛ We will use this to replace interference graph by queries for intersection check.

Chordal interference graph

SSA live-ranges $\equiv$ subtrees of the dominance tree ☛ chordal.

- Forget Chaitin NP-completeness proof.
- Two-phases register allocation: spill to reduce "maxlive" then coalesce. Spill is the main issue.
- Don't be afraid to split, better coalescing schemes.

Basic dominance and SSA properties for interferences

SSA with dominance property and live range intersection

If two variables are simultaneously live at a given program point, then the definition of one dominates the definition of the other (also the first is live at the definition of the second). ☛ We will use this to replace interference graph by queries for intersection check.

Chordal interference graph

SSA live-ranges $\equiv$ subtrees of the dominance tree ☛ chordal.

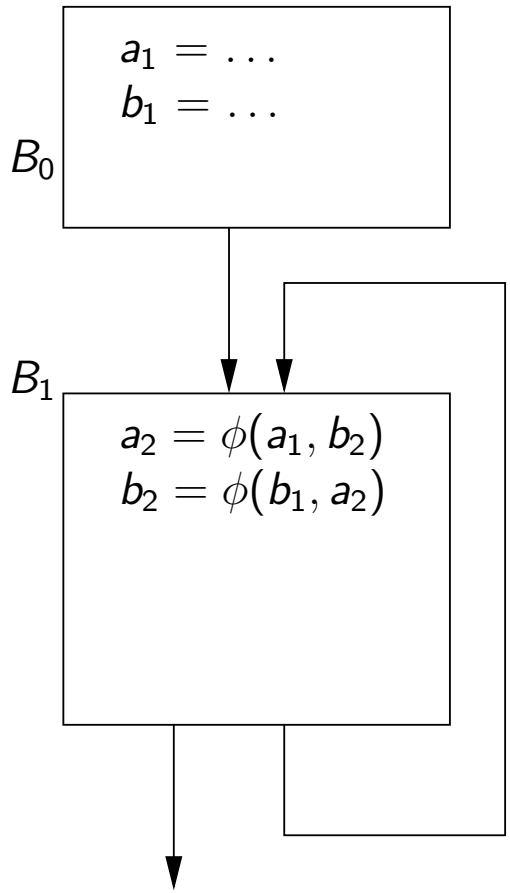
- Forget Chaitin NP-completeness proof.
- Two-phases register allocation: spill to reduce "maxlive" then coalesce. Spill is the main issue.
- Don't be afraid to split, better coalescing schemes.

See work of F. Bouchez, P. Brisk, S. Hack, J. Palsberg, F. Pereira

Why is out-of-SSA translation difficult?

- Cytron et al. (1991): copies in predecessor basic blocks.

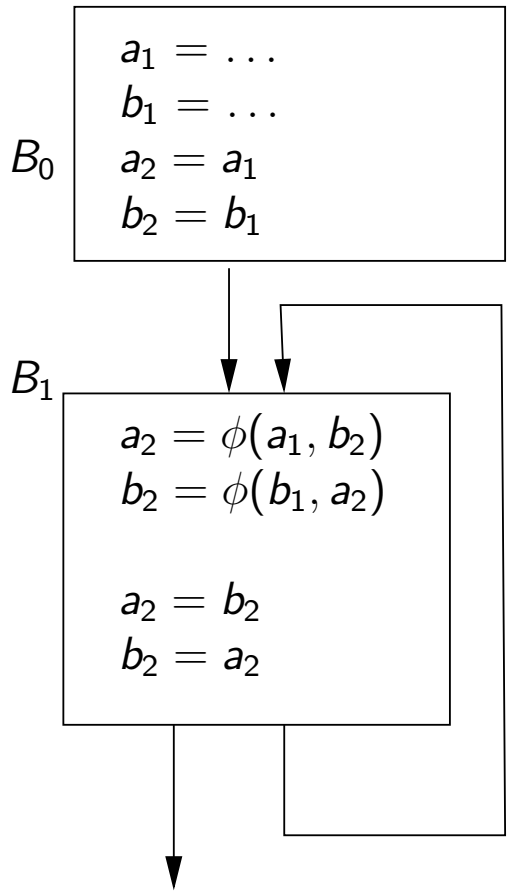
Swap problem



Why is out-of-SSA translation difficult?

- Cytron et al. (1991): copies in predecessor basic blocks.

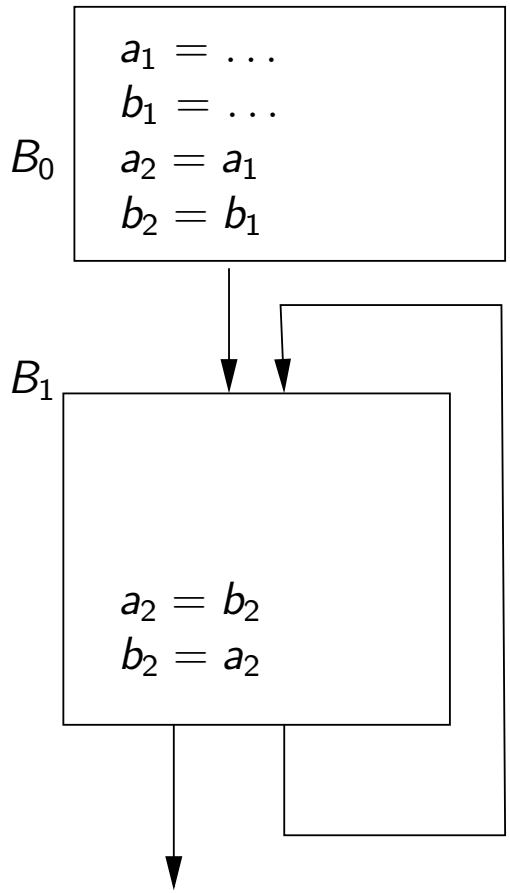
Swap problem



Why is out-of-SSA translation difficult?

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies.

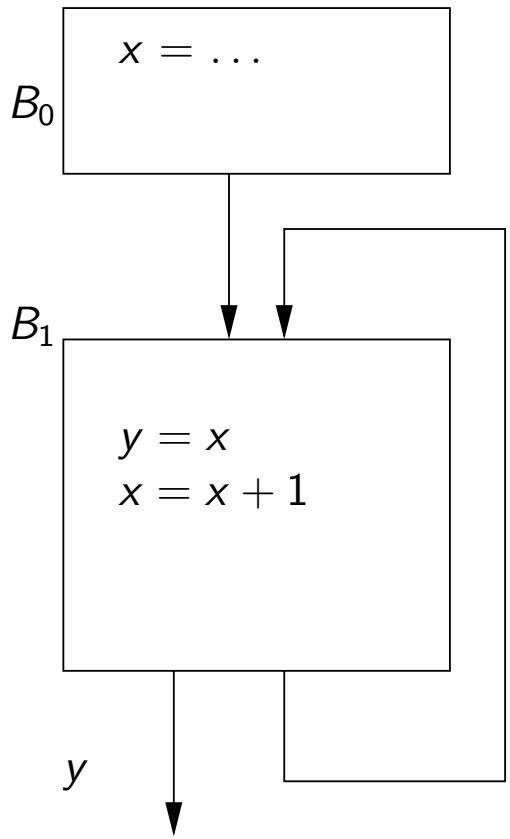
Swap problem



Why is out-of-SSA translation difficult?

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies.

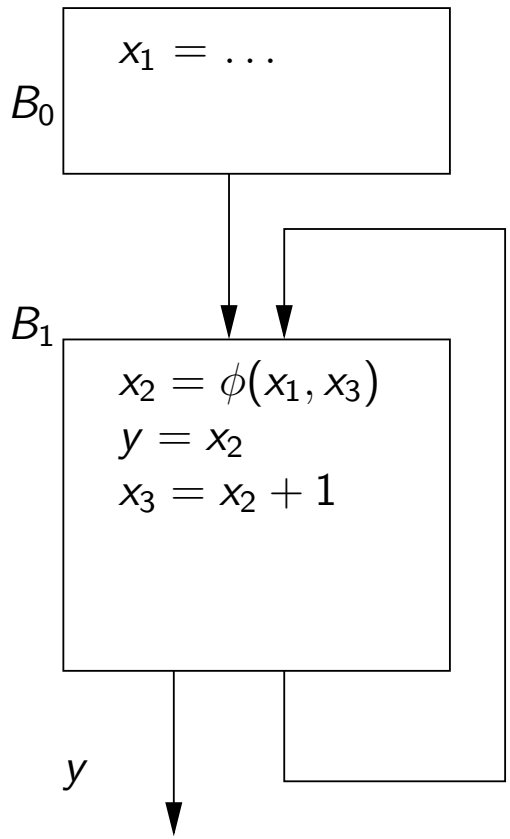
Lost copy problem



Why is out-of-SSA translation difficult?

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies.

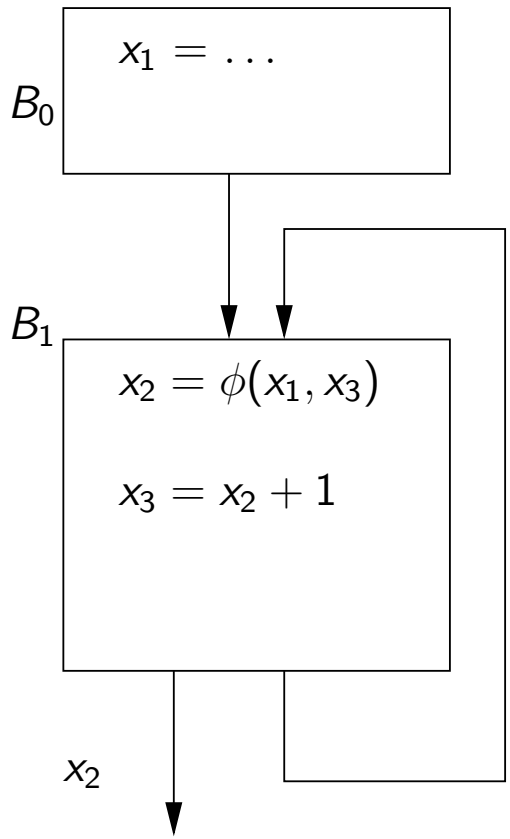
Lost copy problem



Why is out-of-SSA translation difficult?

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies.

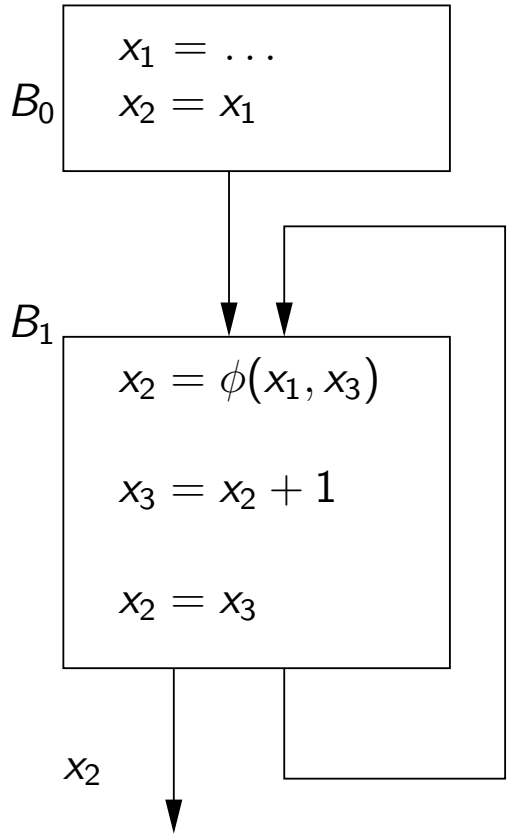
Lost copy problem



Why is out-of-SSA translation difficult?

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies.

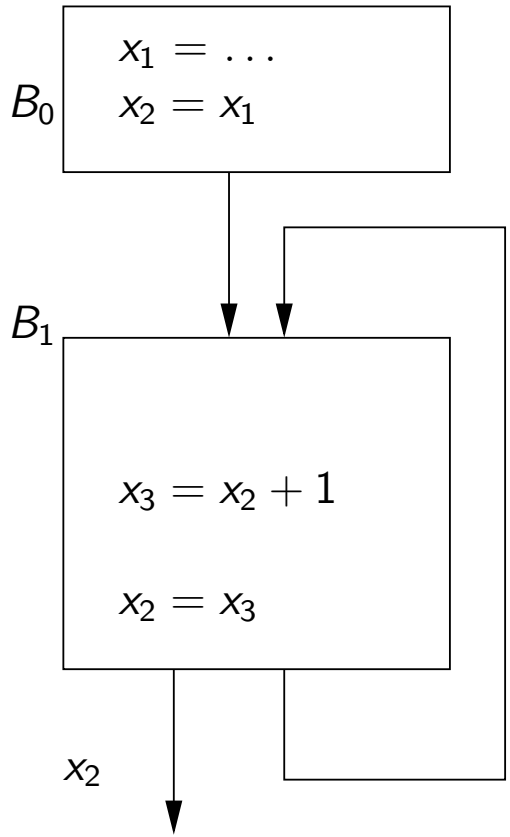
Lost copy problem



Why is out-of-SSA translation difficult?

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies;
  - Bad understanding of critical edges and interferences.

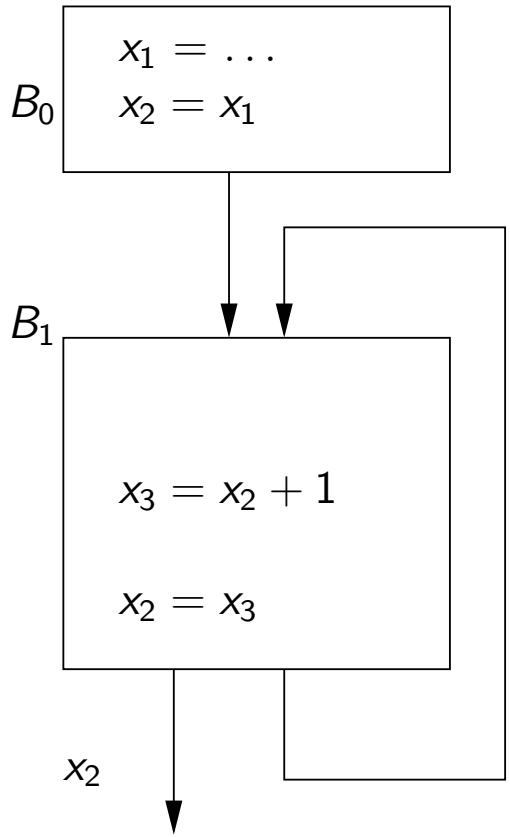
Lost copy problem



Why is out-of-SSA translation difficult?

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies;
  - Bad understanding of critical edges and interferences.
- Briggs et al. (1998): both problems identified. General correctness unclear.

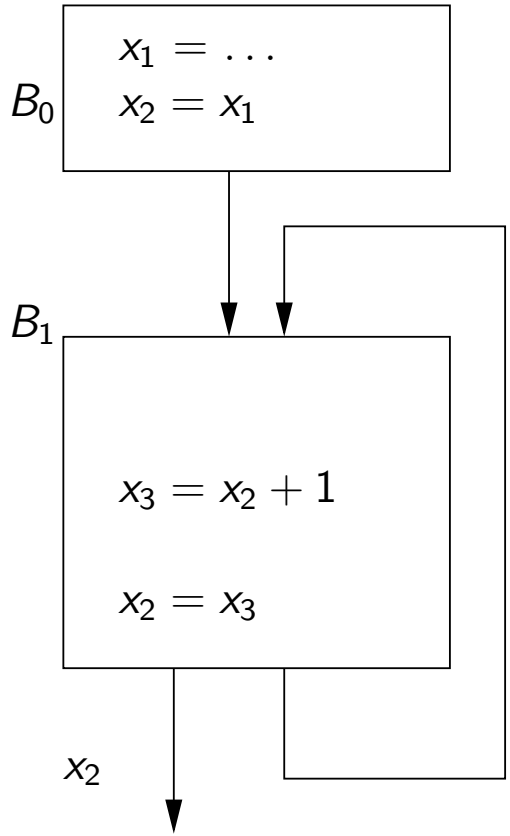
Lost copy problem



Why is out-of-SSA translation difficult?

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies;
  - Bad understanding of critical edges and interferences.
- Briggs et al. (1998): both problems identified. General correctness unclear.
- Sreedhar et al. (1999): correct but
  - handling of complex branching instructions unclear;
  - interplay with coalescing unclear;
  - "virtualization" hard to implement.

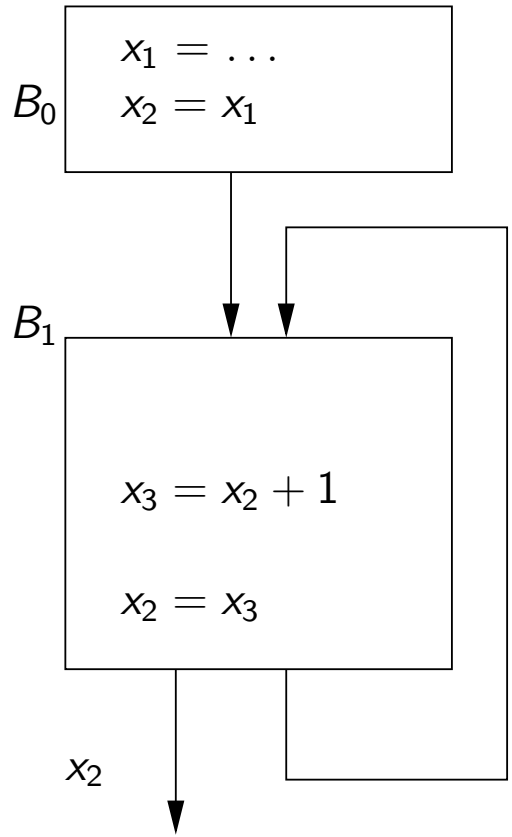
Lost copy problem



Why is out-of-SSA translation difficult?

- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect!
  - Bad understanding of parallel copies;
  - Bad understanding of critical edges and interferences.
- Briggs et al. (1998): both problems identified. General correctness unclear.
- Sreedhar et al. (1999): correct but
  - handling of complex branching instructions unclear;
  - interplay with coalescing unclear;
  - "virtualization" hard to implement.

Lost copy problem



☛ Many SSA optimizations turned off in gcc and Jikes.

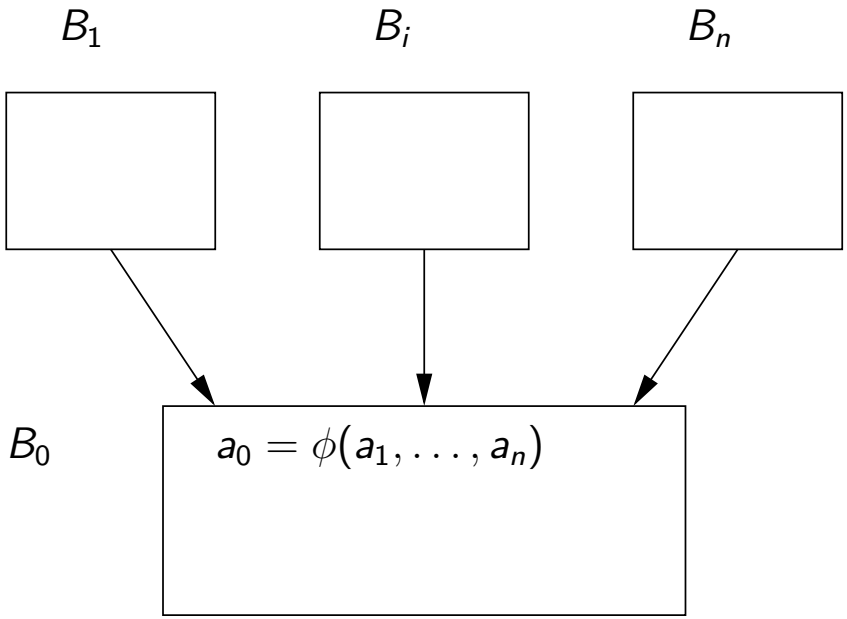
Outline

Going to CSSA (conventional SSA): Sreedhar et al.

Definition (conventional SSA)

CSSA: if variables can be renamed, without changing program semantics, so that, for all ϕ-function $a_0 = \phi(a_1, \ldots, a_n)$, $a_0, \ldots, a_n$ have the same name.

From SSA to CSSA



Going to CSSA (conventional SSA): Sreedhar et al.

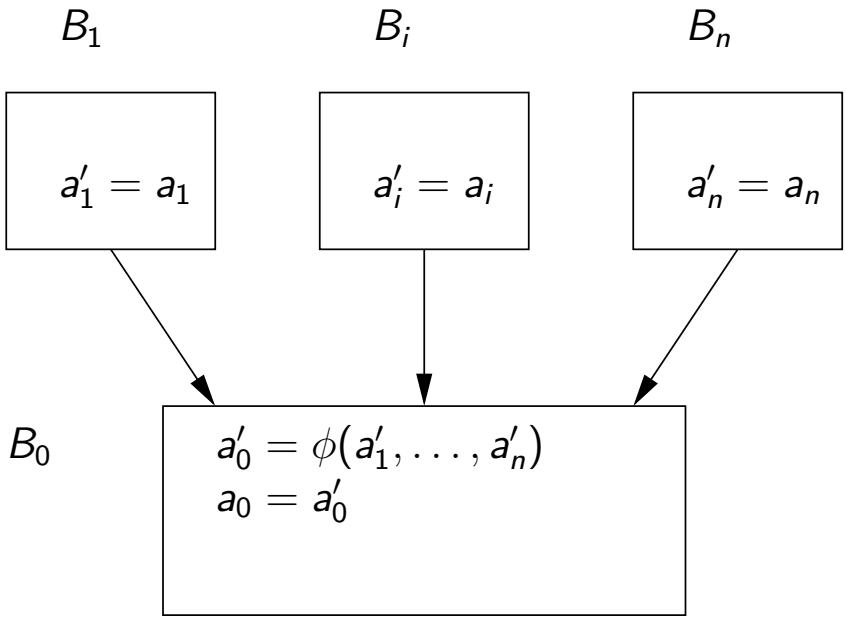
Definition (conventional SSA)

CSSA: if variables can be renamed, without changing program semantics, so that, for all ϕ-function $a_0 = \phi(a_1, \ldots, a_n)$, $a_0, \ldots, a_n$ have the same name.

Correctness

After introduction of variables a'_i and copies, the code is in CSSA.

From SSA to CSSA



Going to CSSA (conventional SSA): Sreedhar et al.

Definition (conventional SSA)

CSSA: if variables can be renamed, without changing program semantics, so that, for all ϕ-function $a_0 = \phi(a_1, \ldots, a_n)$, $a_0, \ldots, a_n$ have the same name.

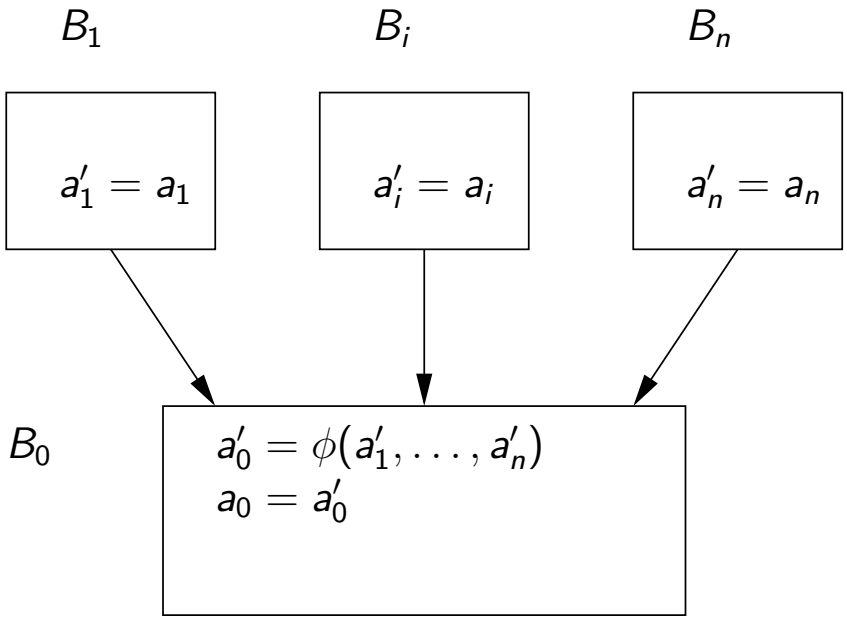
Correctness

After introduction of variables a'_i and copies, the code is in CSSA.

Code quality

Aggressive coalescing can remove useless copies. But better use accurate notion of interferences.

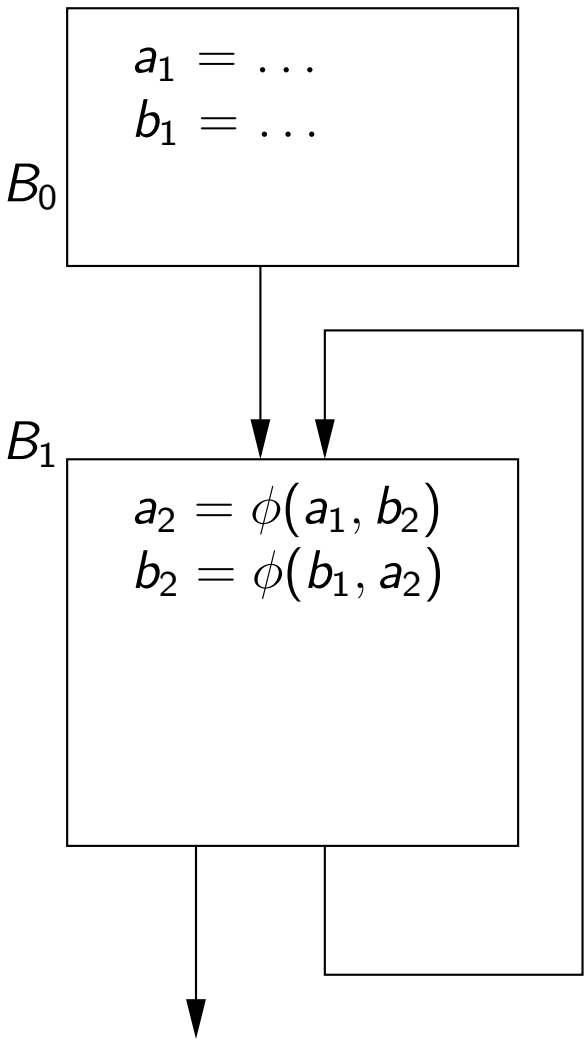
From SSA to CSSA

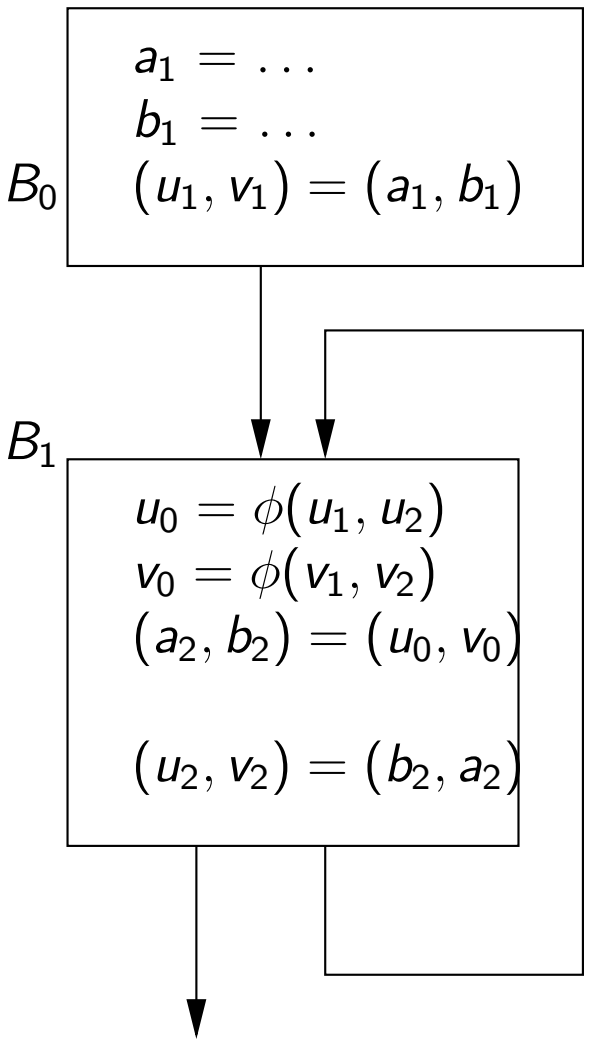


"Liveness of ϕ" defined by the a'_i.

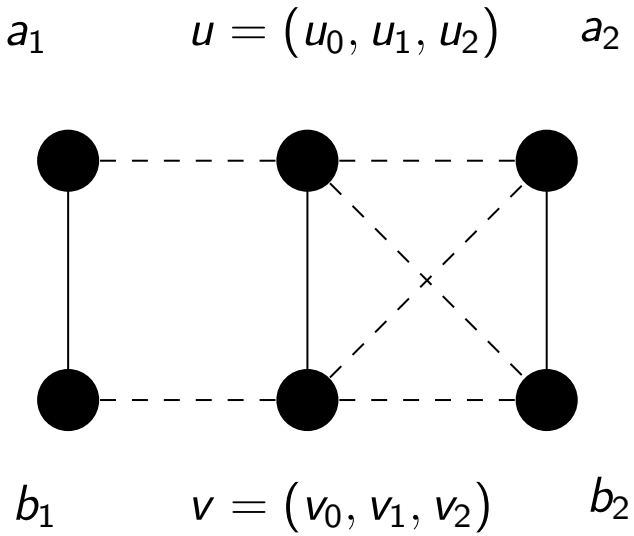
✝ Be careful with potential bugs due to conditional branches that use or define variables.

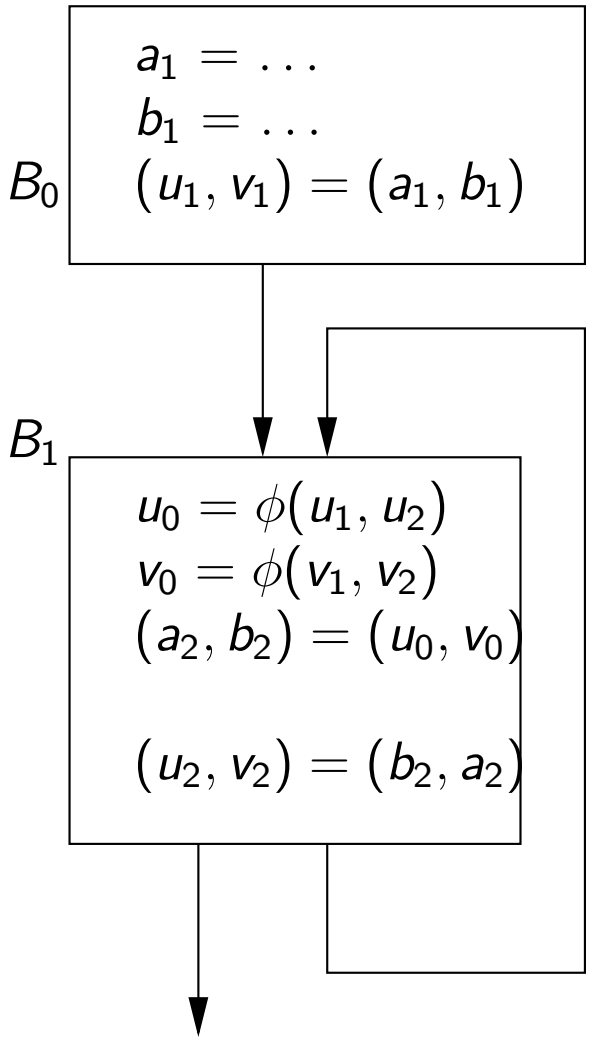
Coalesced example: the swap problem



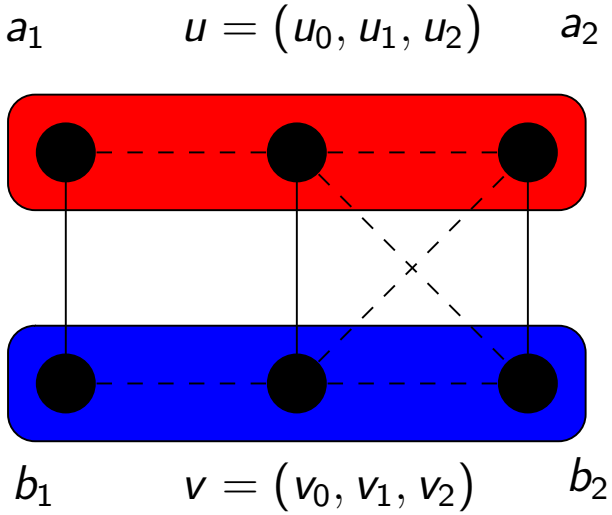


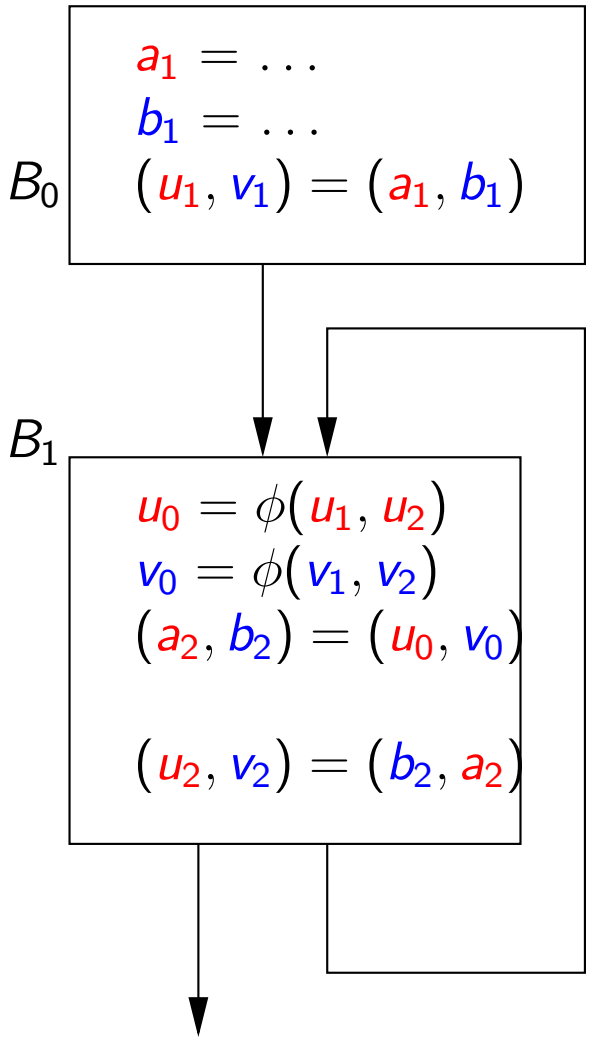
Coalesced example: the swap problem



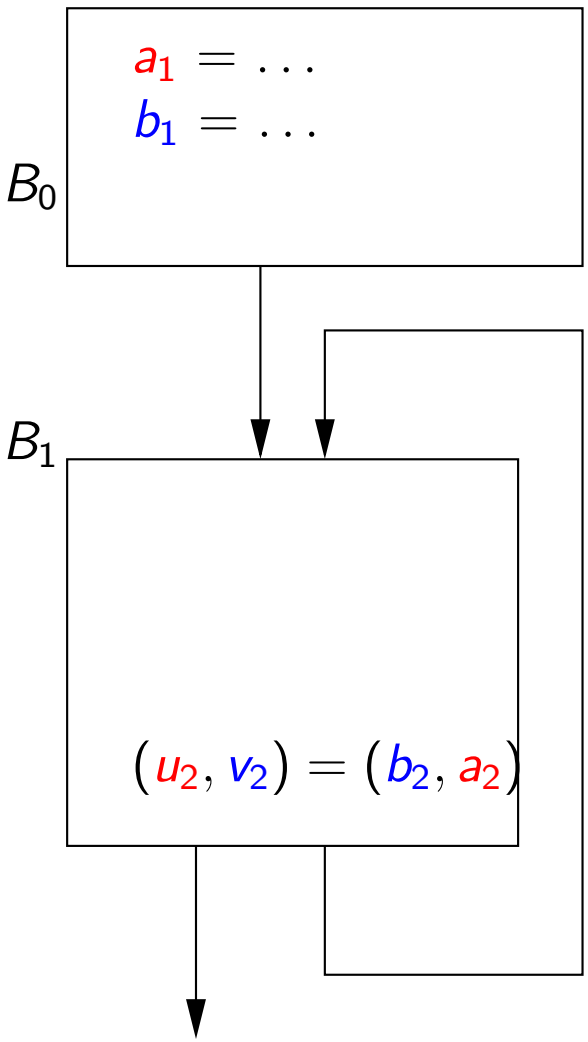


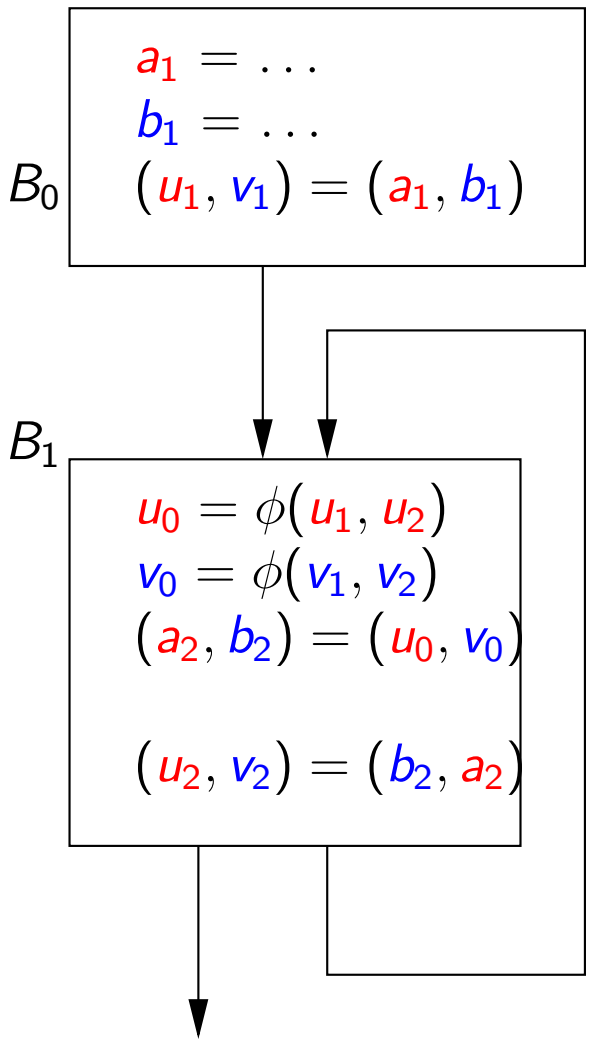
Coalesced example: the swap problem



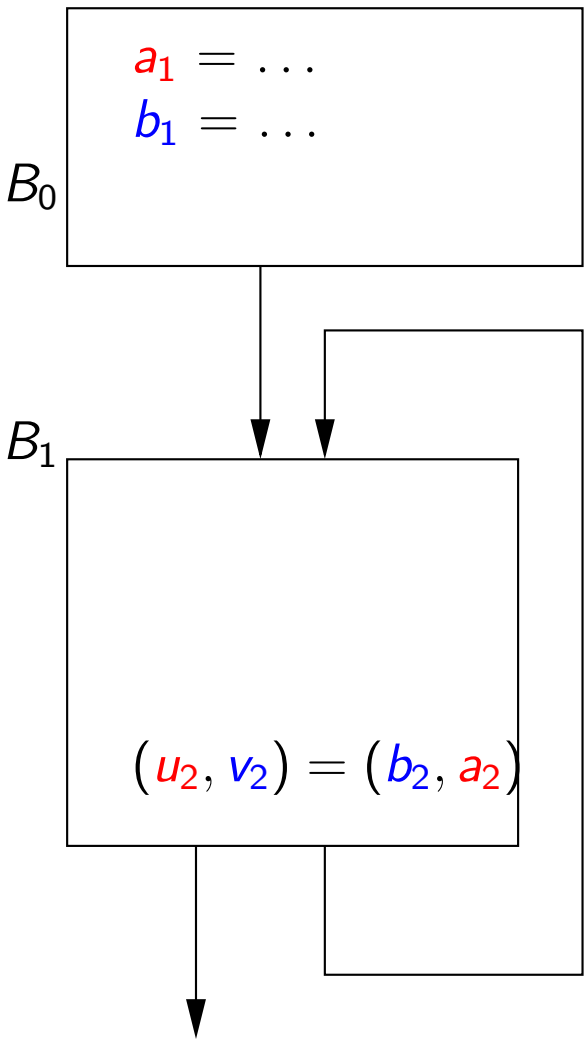


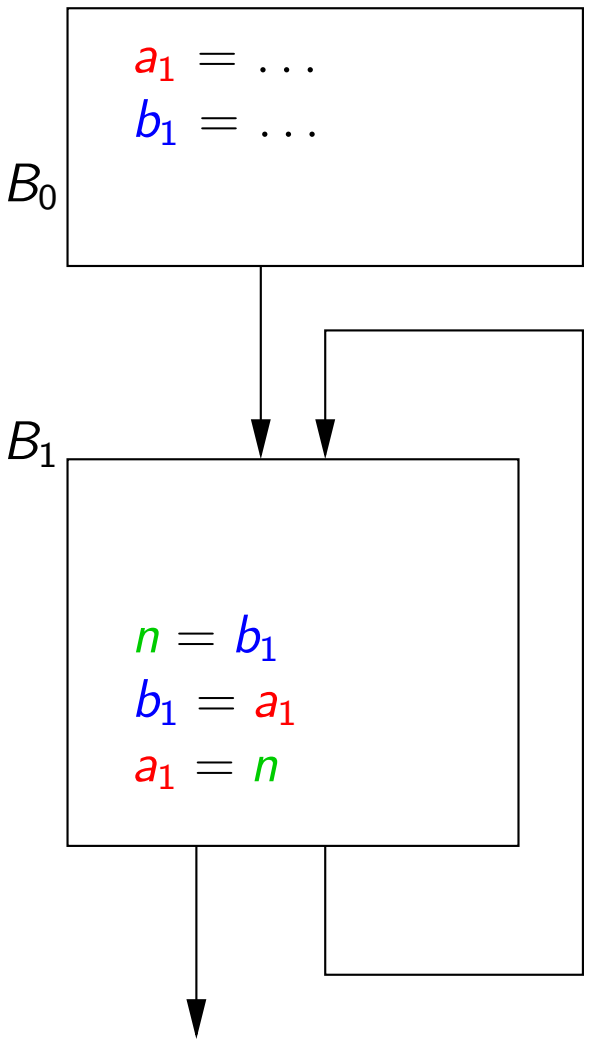
Coalesced example: the swap problem



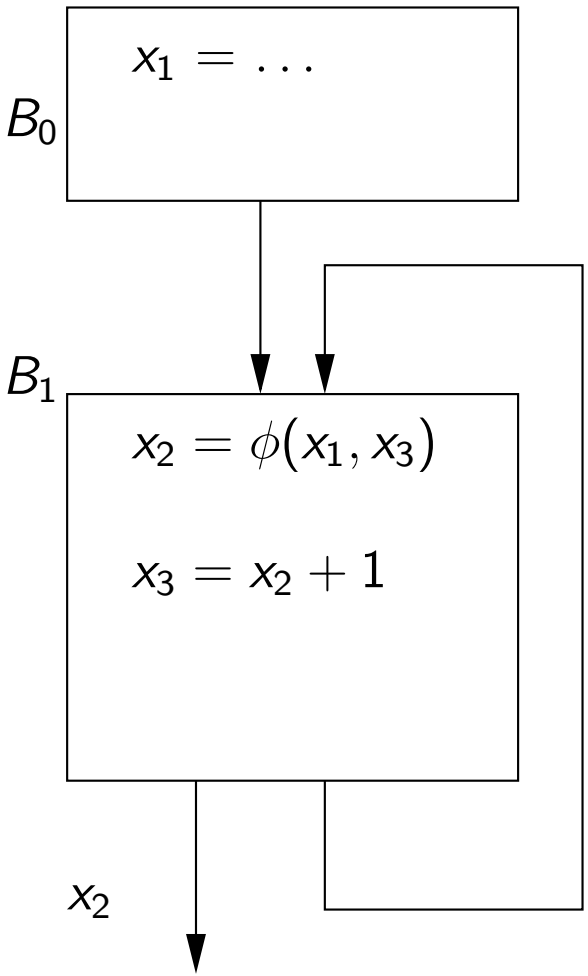


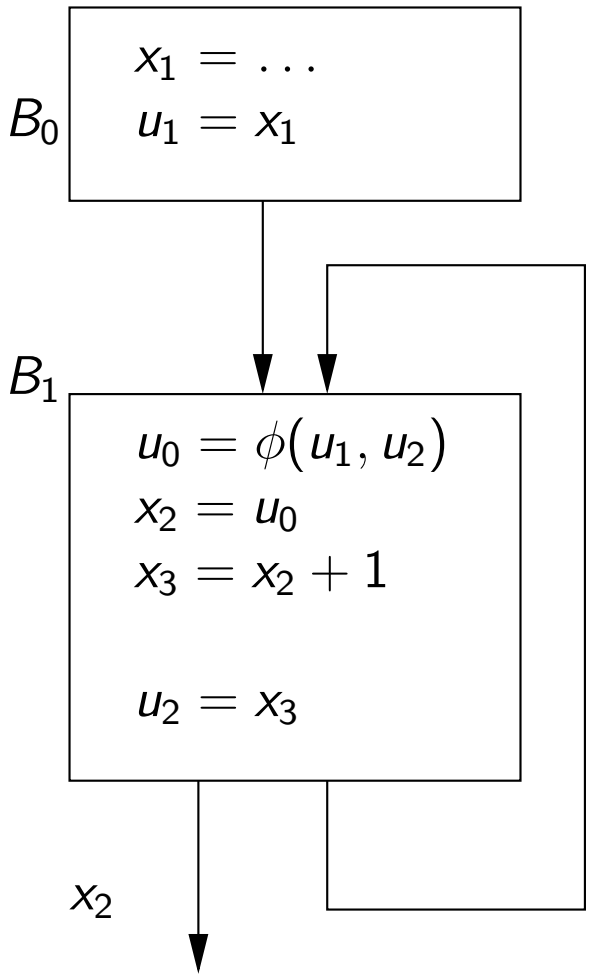
Coalesced example: the swap problem



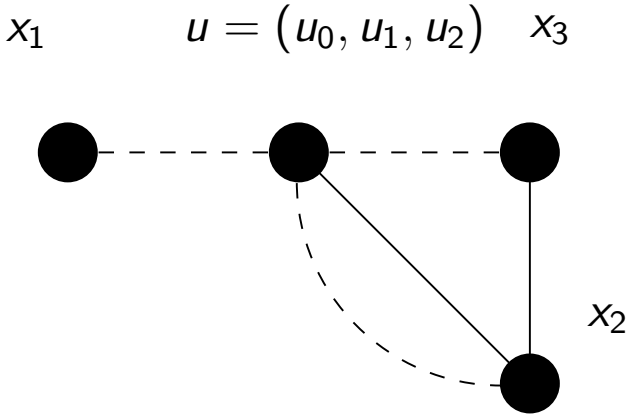


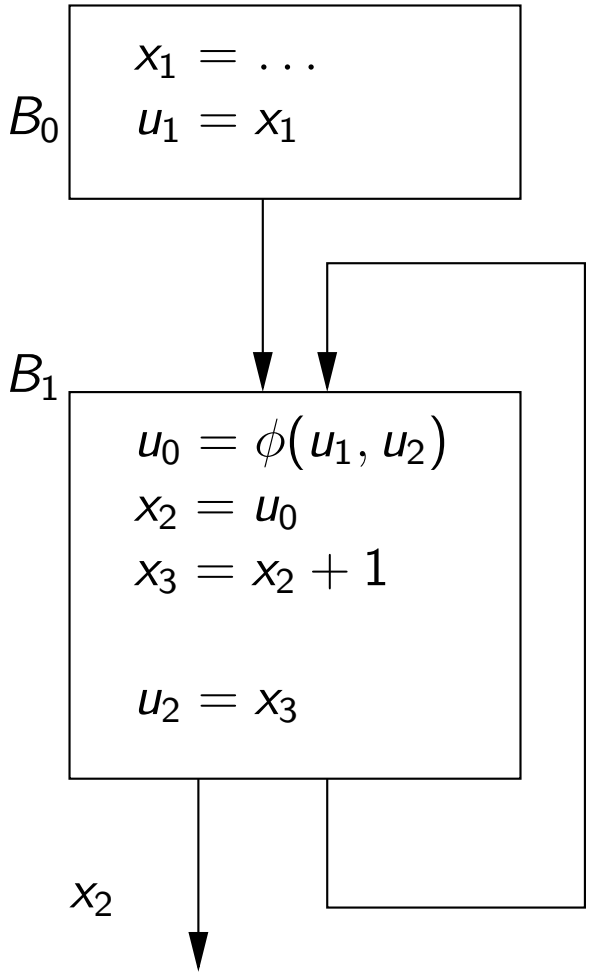
Coalesced example: the lost copy problem



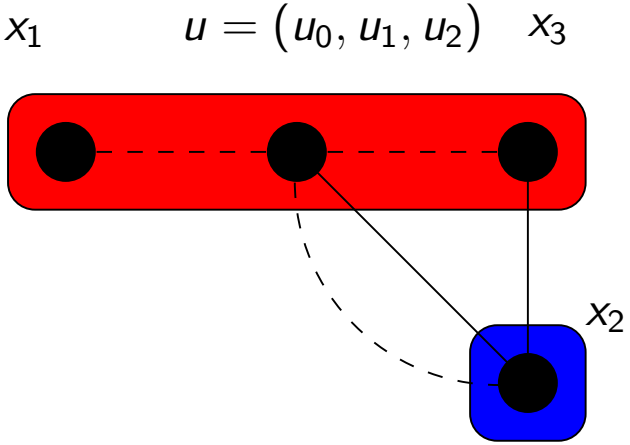


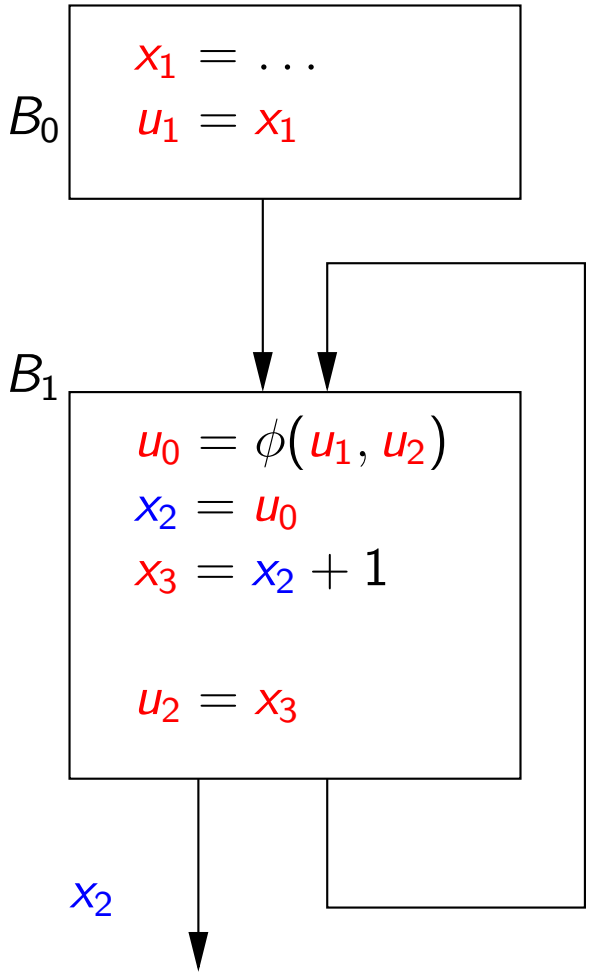
Coalesced example: the lost copy problem



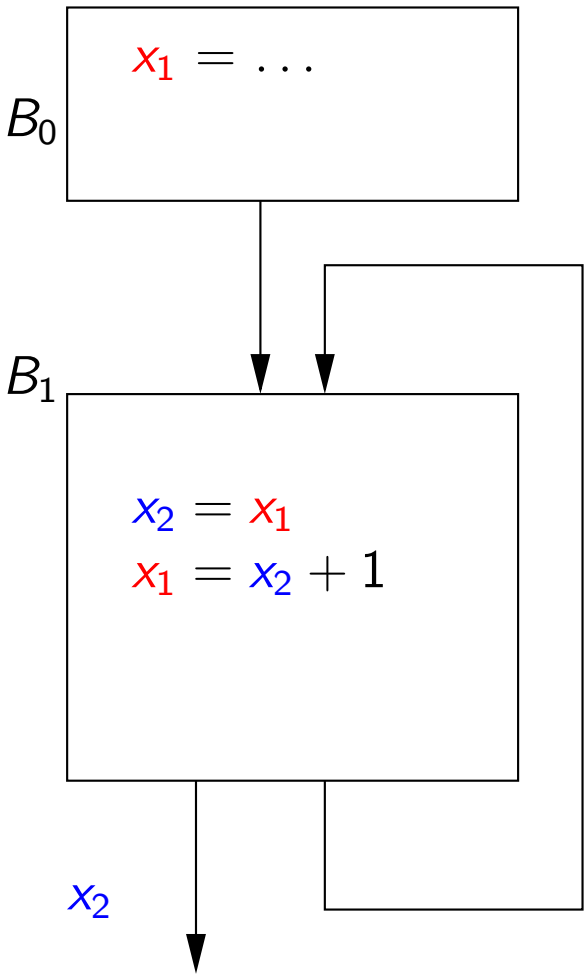


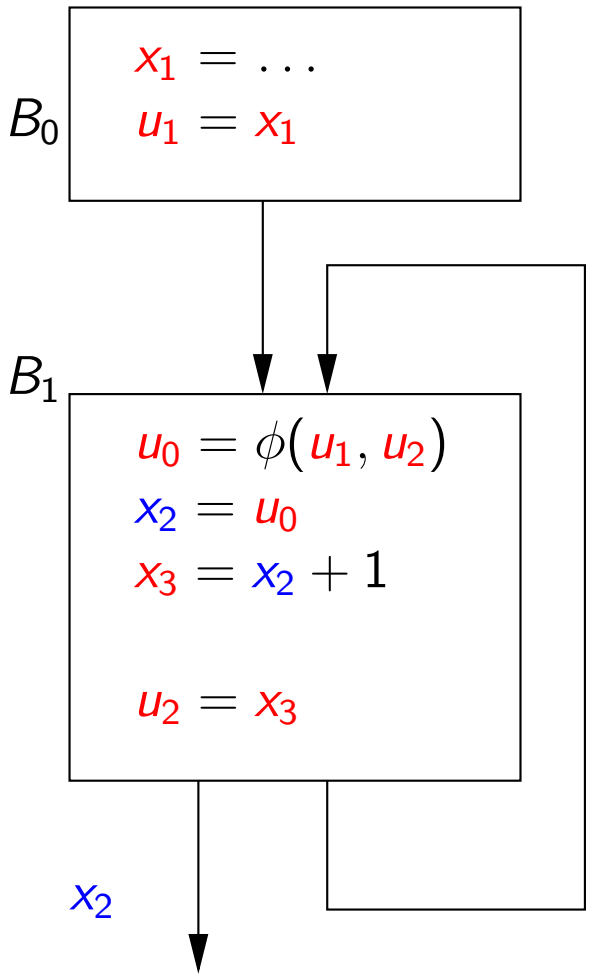
Coalesced example: the lost copy problem





Coalesced example: the lost copy problem

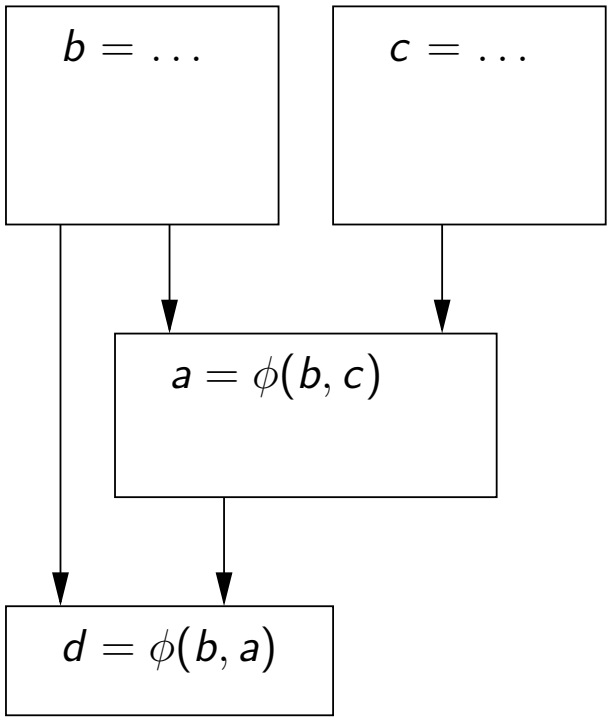




Exploiting SSA: value-based interferences

Definition (Chaitin interference)

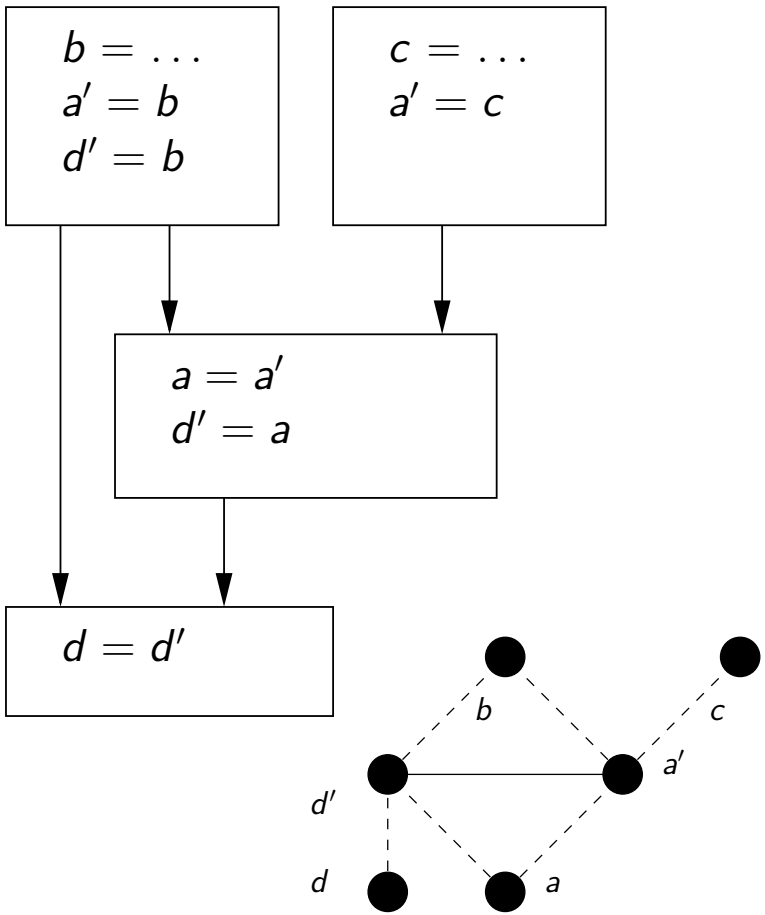
Two variables interfere if one is live at the definition of the other, which is not a copy of the first.



Exploiting SSA: value-based interferences

Definition (Chaitin interference)

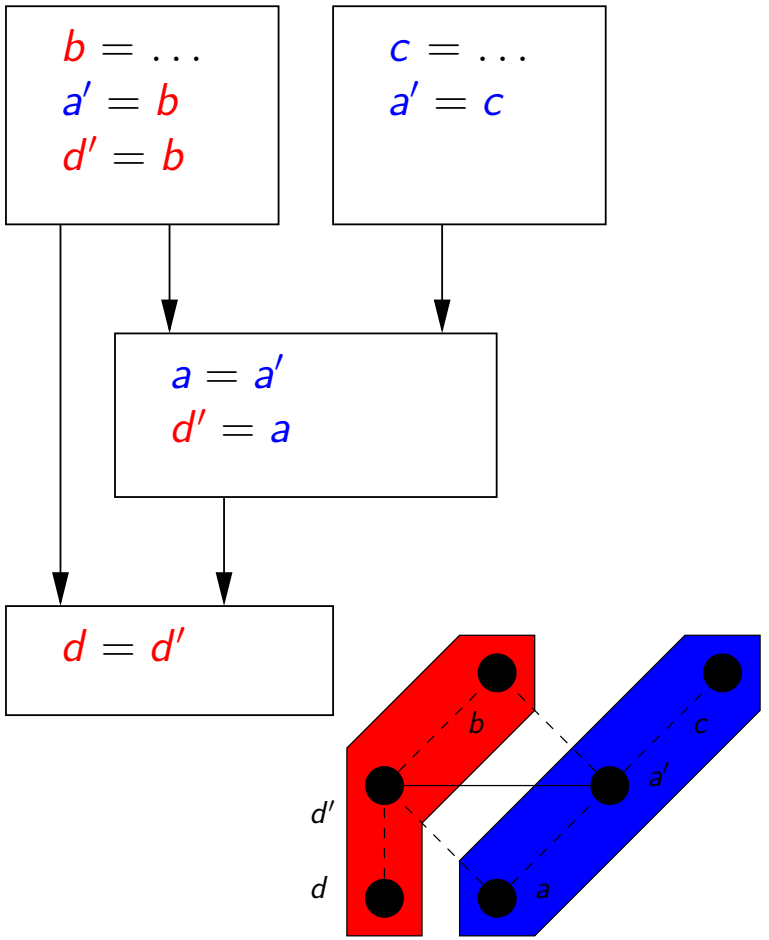
Two variables interfere if one is live at the definition of the other, which is not a copy of the first.



Exploiting SSA: value-based interferences

Definition (Chaitin interference)

Two variables interfere if one is live at the definition of the other, which is not a copy of the first.

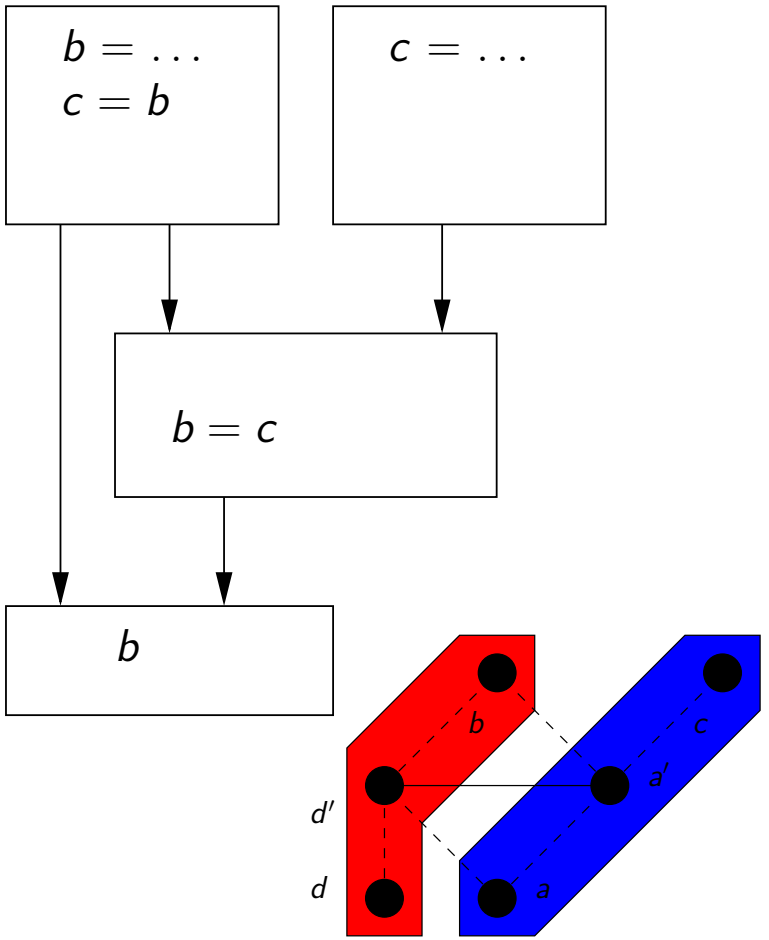


Exploiting SSA: value-based interferences

Definition (Chaitin interference)

Two variables interfere if one is live at the definition of the other, which is not a copy of the first.

☛ Need to update interference graph after coalescing.



Exploiting SSA: value-based interferences

Definition (Chaitin interference)

Two variables interfere if one is live at the definition of the other, which is not a copy of the first.

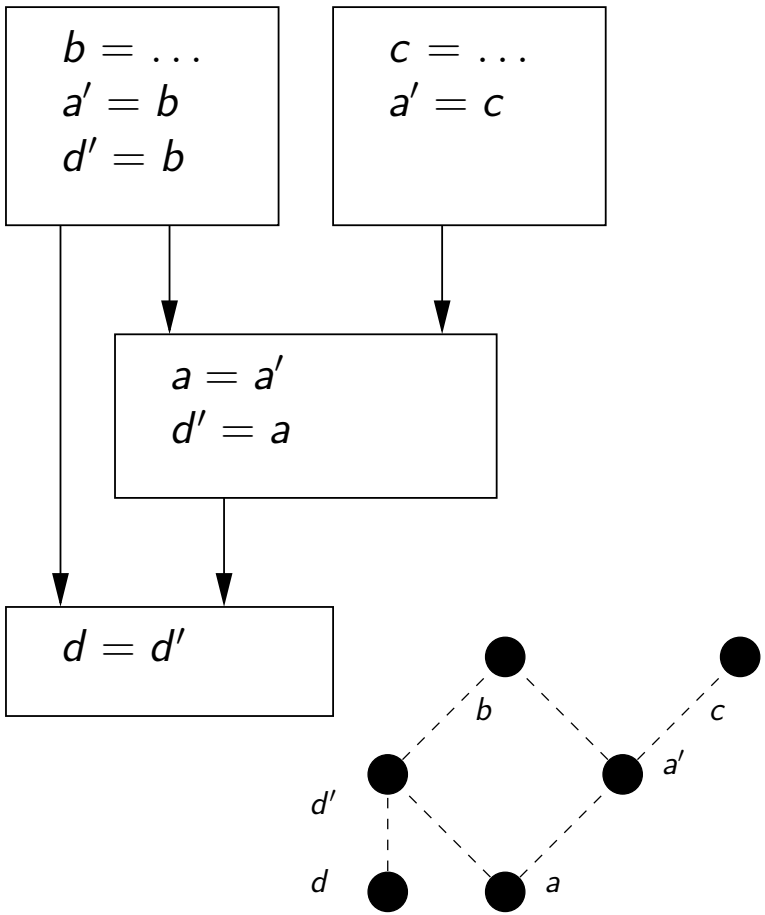
☛ Need to update interference graph after coalescing.

Unique value V of a SSA variable

For a copy $\mathtt{b} = \mathtt{a}$, $V(\mathtt{b}) = V(\mathtt{a})$ (traversal of dominance tree).

Value-based interference

a and b interfere if $V(\mathtt{a}) \neq V(\mathtt{b})$ and Live-range(a) $\cap$ Live-range(b) $\neq \emptyset$.



Exploiting SSA: value-based interferences

Definition (Chaitin interference)

Two variables interfere if one is live at the definition of the other, which is not a copy of the first.

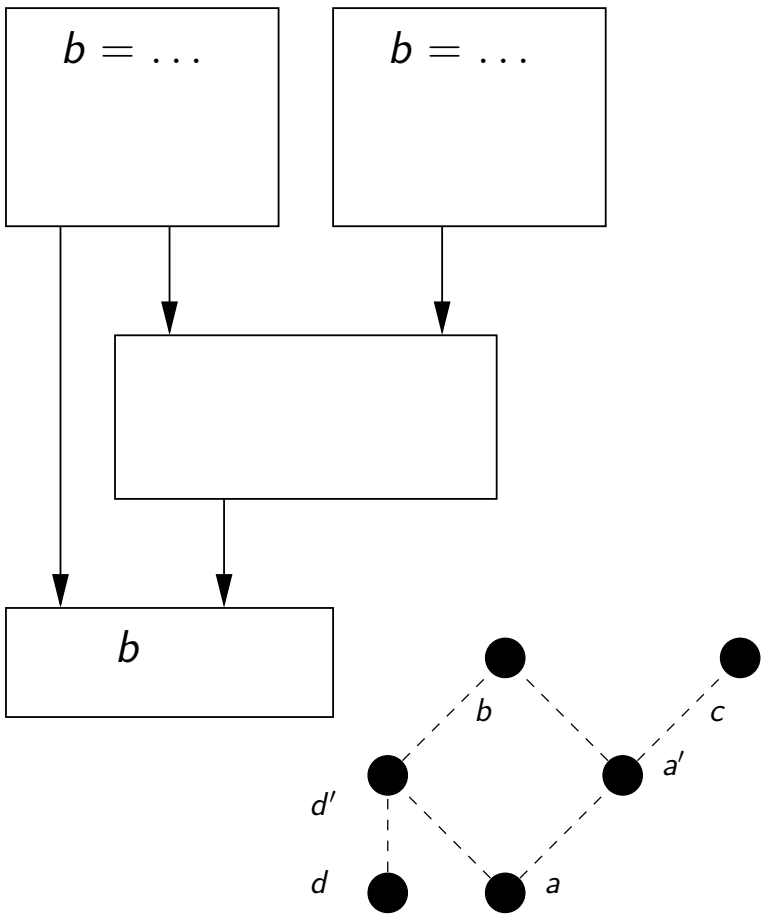
☛ Need to update interference graph after coalescing.

Unique value V of a SSA variable

For a copy $\mathtt{b} = \mathtt{a}$, $V(\mathtt{b}) = V(\mathtt{a})$ (traversal of dominance tree).

Value-based interference

a and b interfere if $V(\mathtt{a}) \neq V(\mathtt{b})$ and Live-range(a) $\cap$ Live-range(b) $\neq \emptyset$.



Using parallel copies instead of sequential copies

Parallel copy semantics

In $(a_1, \ldots, a_n) = (b_1, \ldots, b_n)$, all copies $a_i = b_i$ are simultaneous.

- Fewer interferences than with sequential copies.
- Easier insertion & liveness updates.
- But need to sequentialize.

Using parallel copies instead of sequential copies

Parallel copy semantics

In $(a_1, \ldots, a_n) = (b_1, \ldots, b_n)$, all copies $a_i = b_i$ are simultaneous.

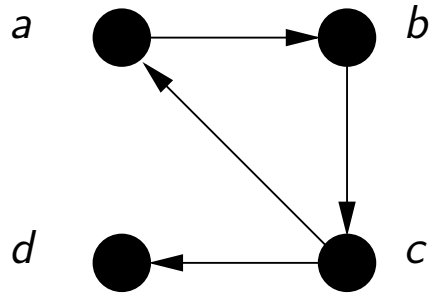
- Fewer interferences than with sequential copies.
- Easier insertion & liveness updates.
- But need to sequentialize.

Particular copy structure

Directed graph with edges $b_i \rightarrow a_i$.

- Directed trees with roots=circuits.
- Insert copies for the leaves first.

$$(a, b, c, d) = (c, a, b, c)$$



Using parallel copies instead of sequential copies

Parallel copy semantics

In $(a_1, \ldots, a_n) = (b_1, \ldots, b_n)$, all copies $a_i = b_i$ are simultaneous.

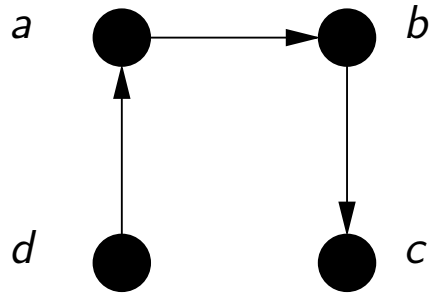
- Fewer interferences than with sequential copies.
- Easier insertion & liveness updates.
- But need to sequentialize.

Particular copy structure

Directed graph with edges $b_i \rightarrow a_i$.

- Directed trees with roots=circuits.
- Insert copies for the leaves first.

$d = c$
$(a, b, c) = (d, a, b)$



Using parallel copies instead of sequential copies

Parallel copy semantics

In $(a_1, \ldots, a_n) = (b_1, \ldots, b_n)$, all copies $a_i = b_i$ are simultaneous.

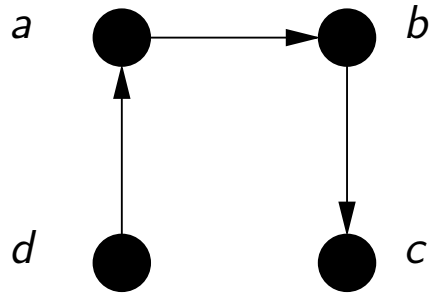
- Fewer interferences than with sequential copies.
- Easier insertion & liveness updates.
- But need to sequentialize.

Particular copy structure

Directed graph with edges $b_i \rightarrow a_i$.

- Directed trees with roots=circuits.
- Insert copies for the leaves first.

$$
\begin{aligned}
d &= c \\
c &= b \\
b &= a \\
a &= d
\end{aligned}
$$



Using parallel copies instead of sequential copies

Parallel copy semantics

In $(a_1, \ldots, a_n) = (b_1, \ldots, b_n)$, all copies $a_i = b_i$ are simultaneous.

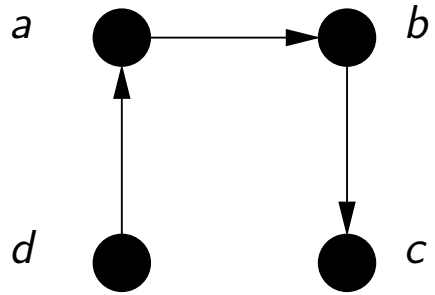
- Fewer interferences than with sequential copies.
- Easier insertion & liveness updates.
- But need to sequentialize.

Particular copy structure

Directed graph with edges $b_i \rightarrow a_i$.

- Directed trees with roots=circuits.
- Insert copies for the leaves first.
- Simple circuit: one more copy.

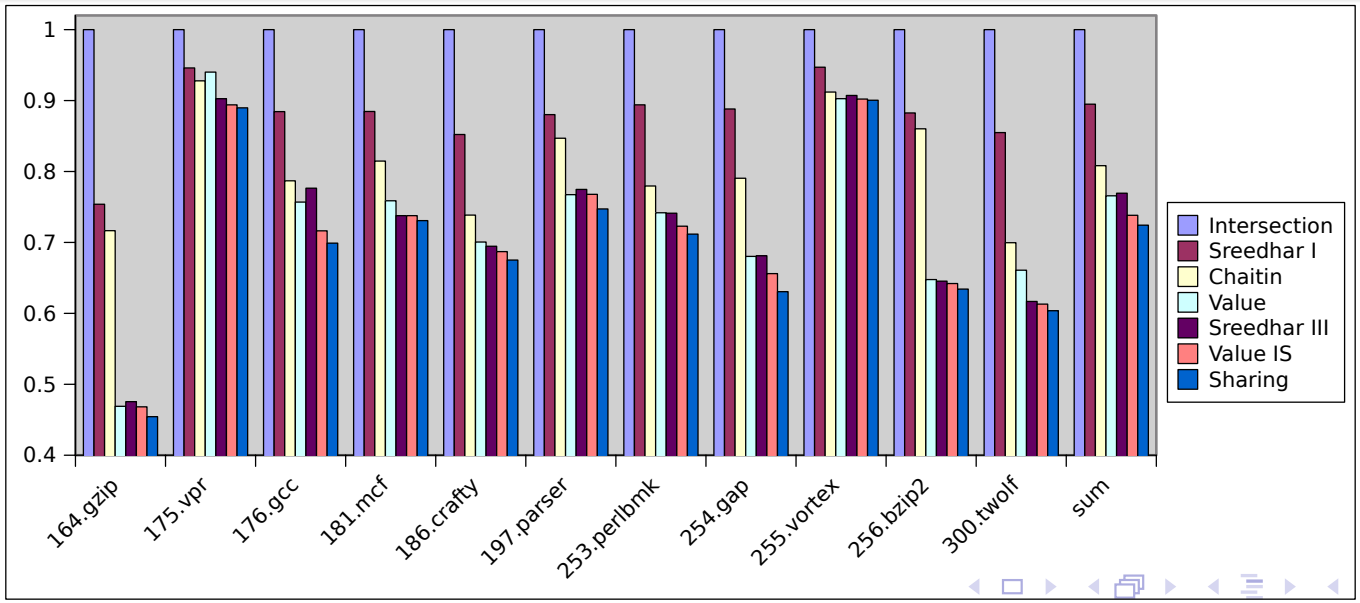
$$
\begin{aligned}
d &= c \\
c &= b \\
b &= a \\
a &= d
\end{aligned}
$$



Qualitative experiments with SPEC CINT2000

Key points of the out-of-SSA translation

- Copy insertion (to go to CSSA and to handle register renaming constraints) followed by coalescing.
- Value-based interferences ☛ coalescing is improved and independent of virtualization (i.e., as in Sreedhar III).
- Parallel copies followed by sequentialization.



Outline

How to coalesce variables?

Two alternatives

- Use a working interference graph where, in case of coalescing, corresponding vertices 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

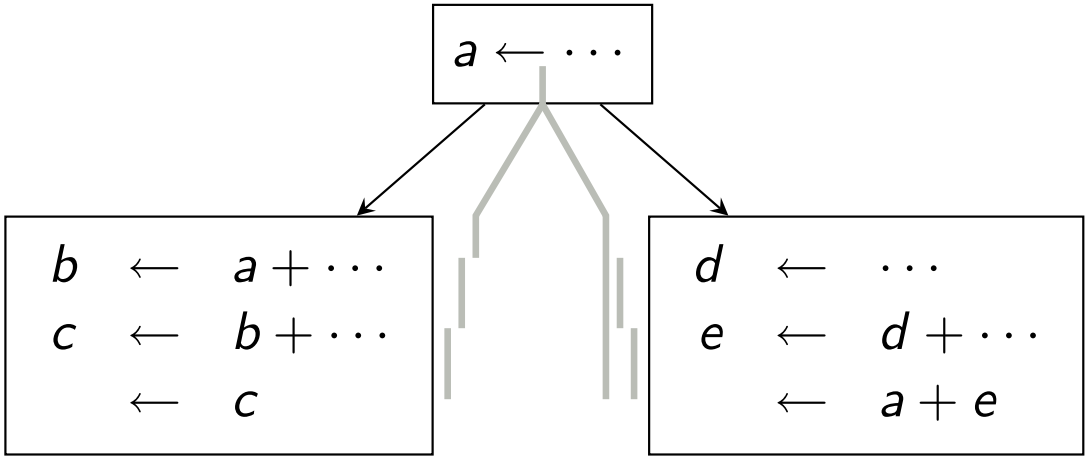
- Use a working interference graph where, in case of coalescing, corresponding vertices are merged. $O(1)$ interference query.
- Manipulate congruence classes, i.e., sets of coalesced variables. Interferences must be tested between sets.

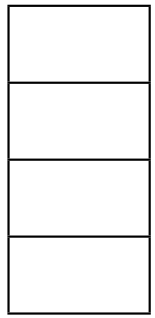
Chaitin, Sreedhar, Budimlić use congruence classes. Also useful to avoid interference graph. Naive algorithm: quadratic complexity.

Key properties for linear-complexity live range intersection

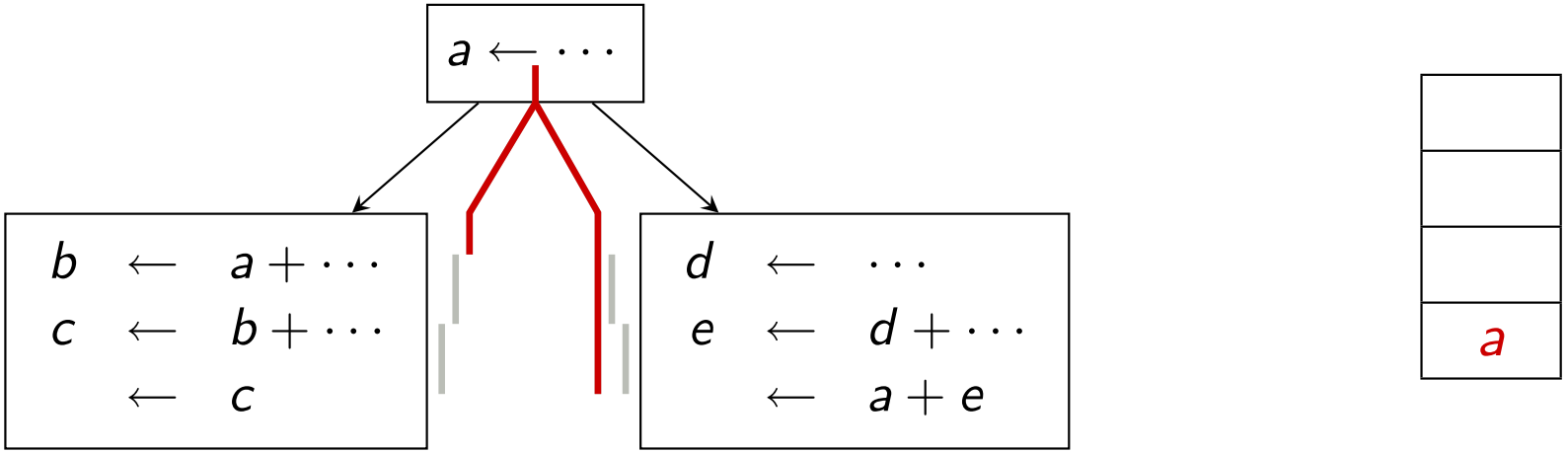
- 2 variables intersect if one is live at the definition of the other.
- In this case, the first definition dominates the second one.
- Budimlić: a set contains 2 intersecting variables if it contains a variable that intersects its "parent dominating" variable.

Fast interference test for a set of variables

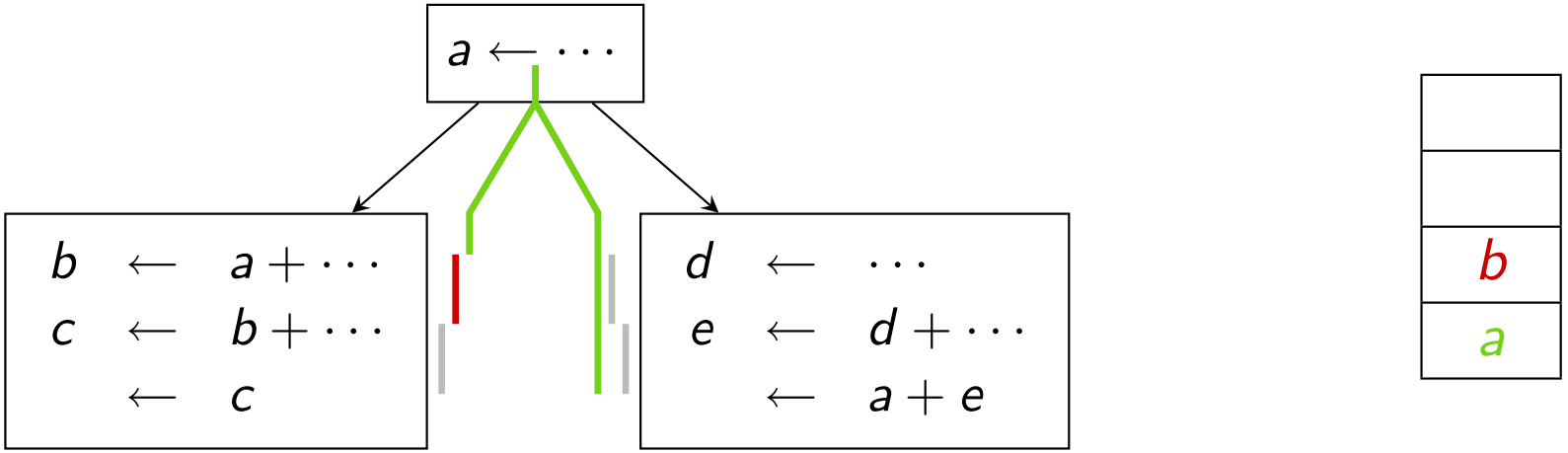




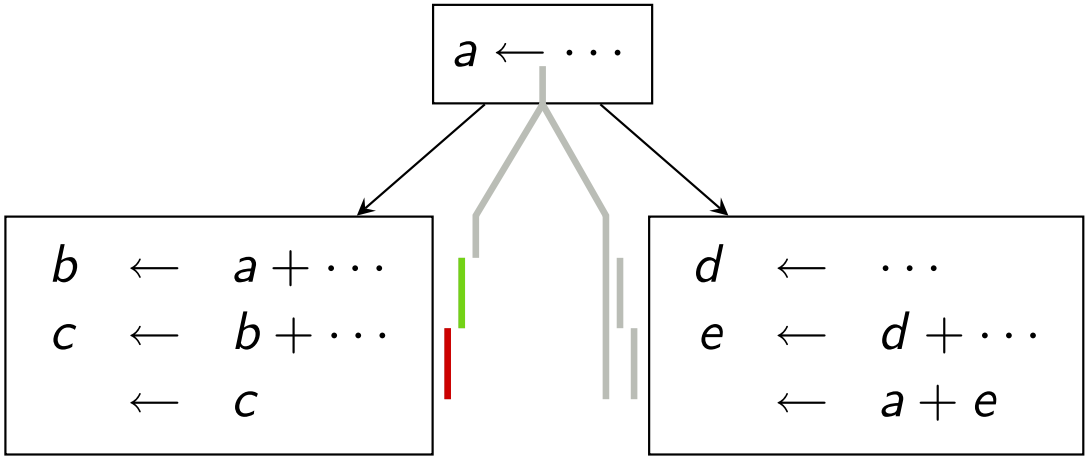
Fast interference test for a set of variables

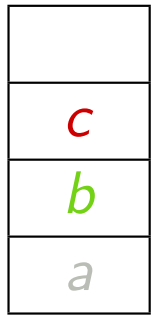


Fast interference test for a set of variables

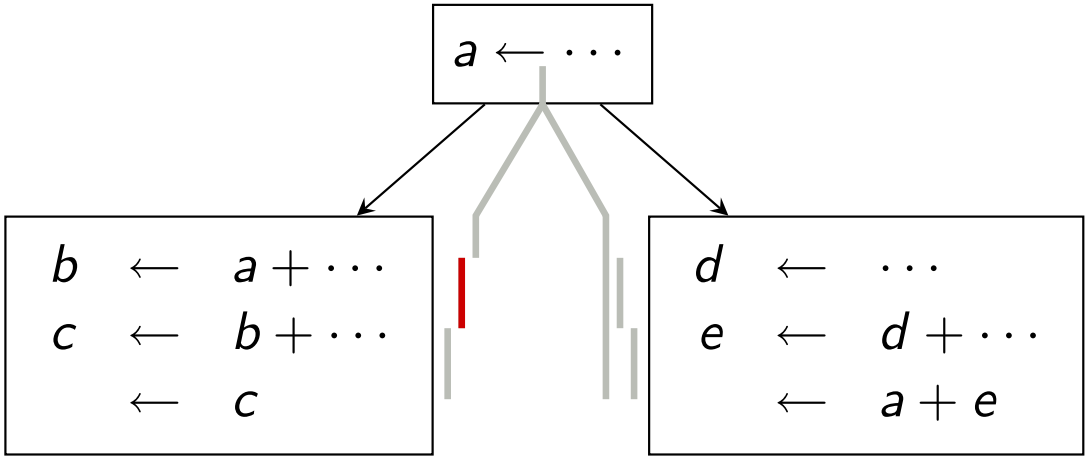


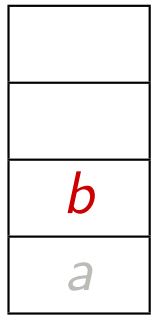
Fast interference test for a set of variables



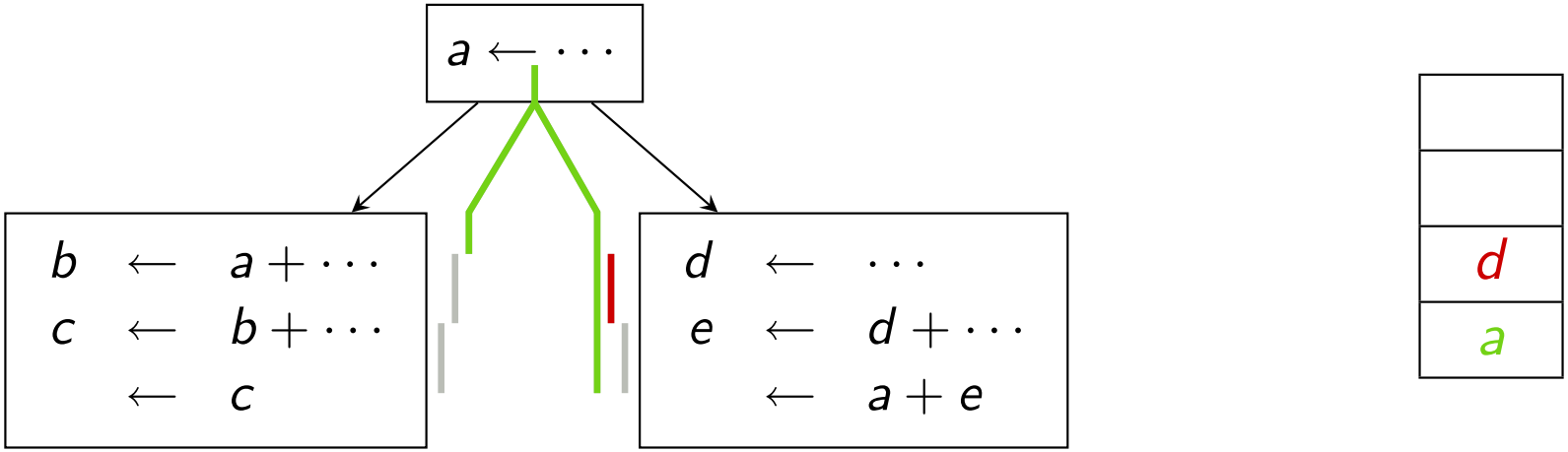


Fast interference test for a set of variables

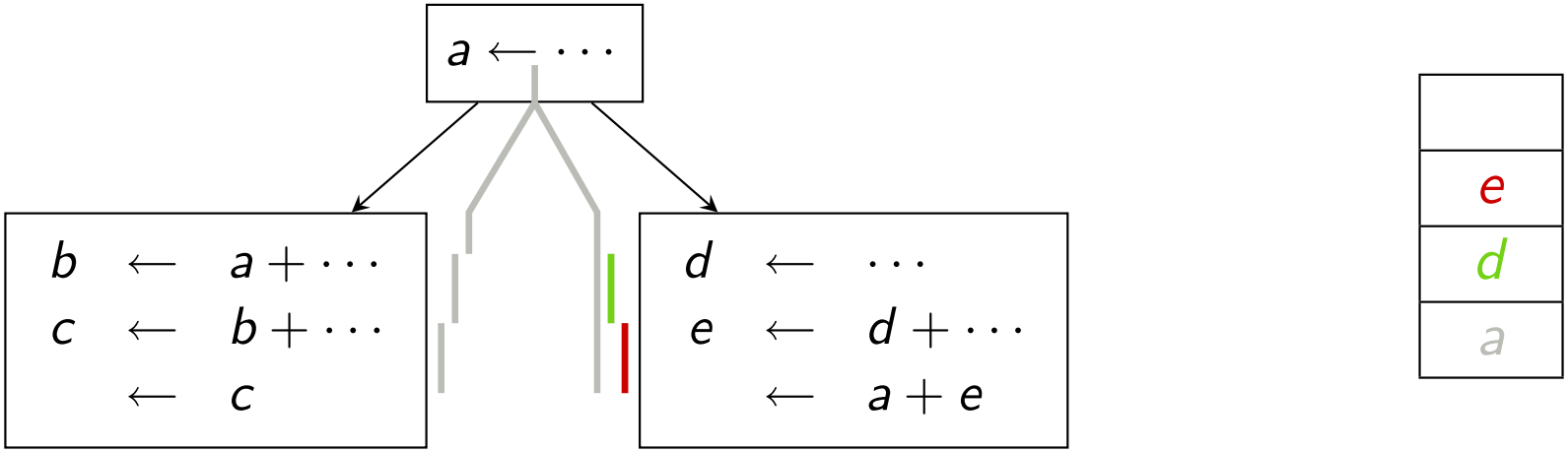




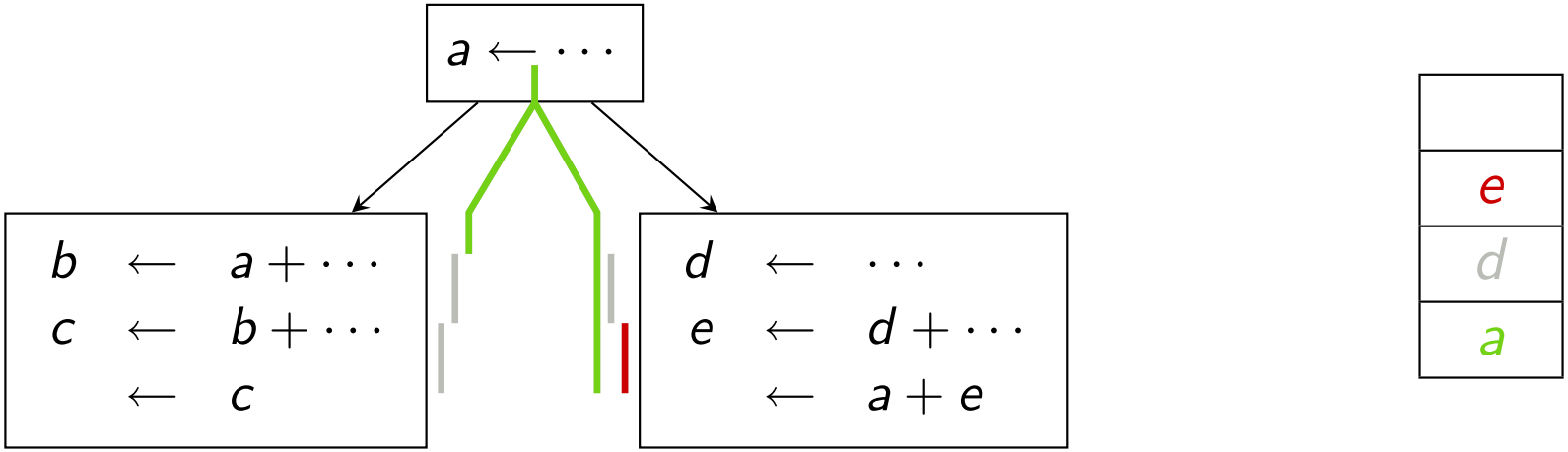
Fast interference test for a set of variables



Fast interference test for a set of variables



Fast interference test for a set of variables



Algorithm 1: 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  dom ← empty_stack ; i ← 0  ;                          /* stack of the traversal */
2  while i < list.size() do
3      current ← list(i++) ;
4      other ← dom.top() ;                               /* NULL if dom is empty */
5      while (other ≠ NULL) and dominate(other, current) = FALSE do
6          dom.pop() ;                                   /* not the desired parent, remove */
7          other ← dom.top() ;                           /* consider next one */
8      parent ← other ;
9      if (parent ≠ NULL) and (intersect(current, parent) = TRUE) then
       return TRUE ;                                     /* intersection detected */
10     dom.push(current) ;                               /* otherwise, keep checking */
11 return 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. Also 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. Also 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.

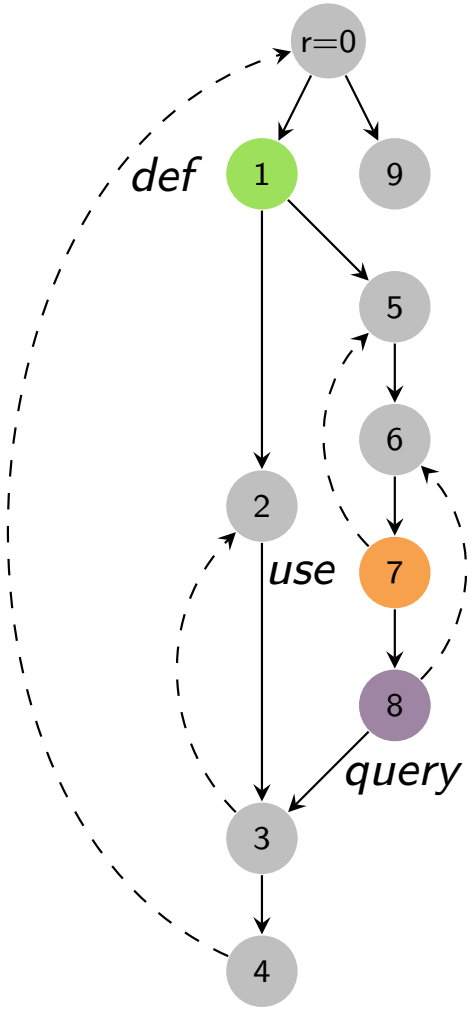
Fewer intersection tests ☛ possible now to use more expensive queries for intersection and liveness and avoid interference graph:

- Budimlić intersection test, still using liveness sets.
- Fast liveness checking of Boissinot et al. (CGO'08).

Fast liveness checking (Boissinot et al. CGO'08)

Definition (Liveness)

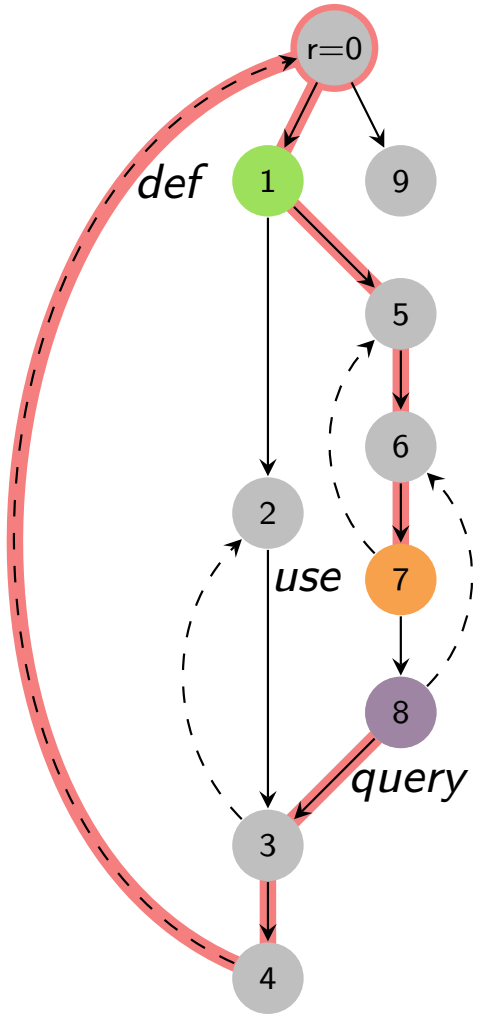
Variable a is live-in at q if there is a path from q to a *use* of a, that does not contain its *def*.



Fast liveness checking (Boissinot et al. CGO'08)

Definition (Liveness)

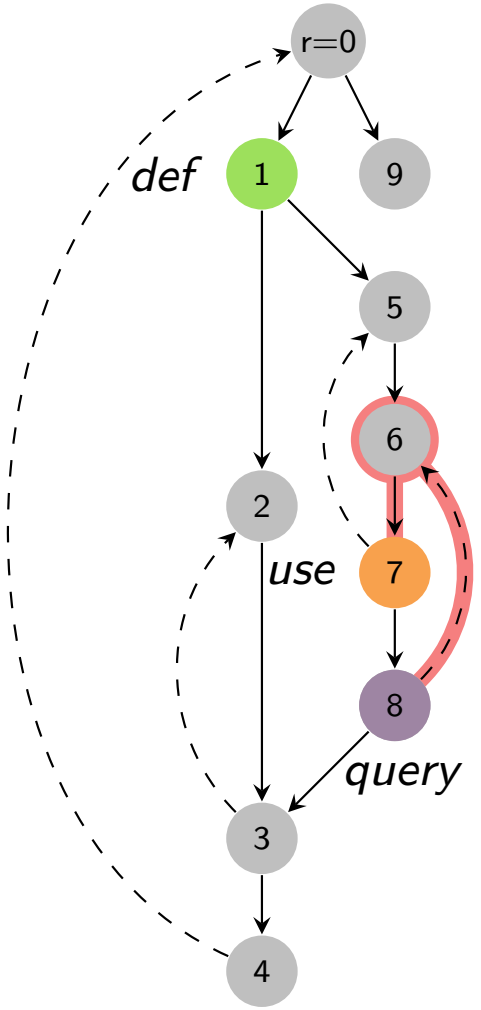
Variable a is live-in at q if there is a path from q to a *use* of a, that does not contain its *def*.



Fast liveness checking (Boissinot et al. CGO'08)

Definition (Liveness)

Variable a is live-in at q if there is a path from q to a *use* of a, that does not contain its *def*.



Fast liveness checking (Boissinot et al. CGO'08)

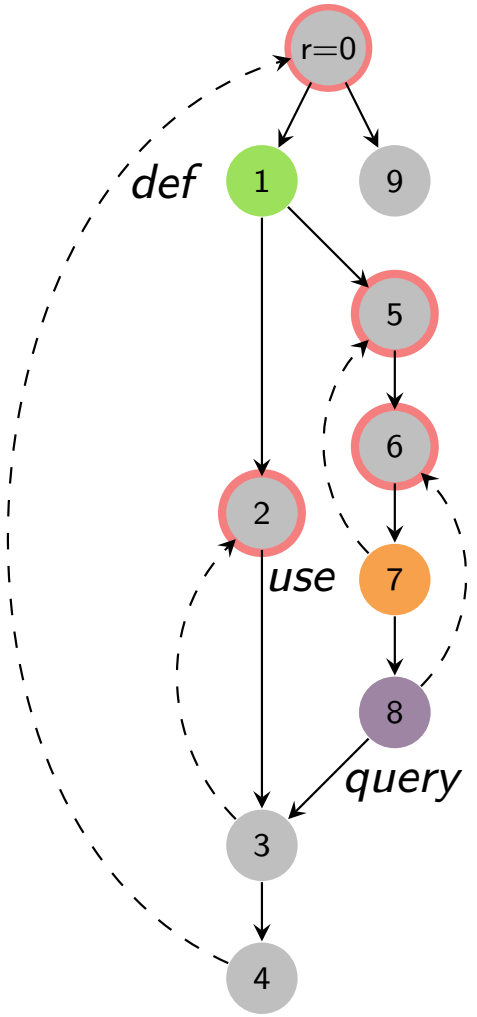
Definition (Liveness)

Variable a is live-in at q if there is a path from q to a *use* of a, that does not contain its *def*.

Algorithm (see CGO'08 paper)

Precomputation:

- Compute transitive closure of G', the CFG without DFS back edges;
- For each q, compute a set T_q of back-edge targets reached from q.



Fast liveness checking (Boissinot et al. CGO'08)

Definition (Liveness)

Variable a is live-in at q if there is a path from q to a *use* of a, that does not contain its *def*.

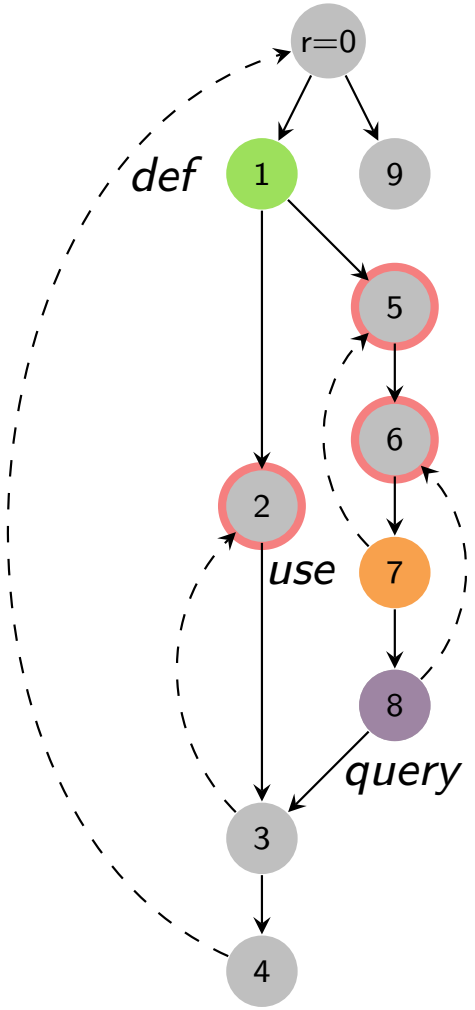
Algorithm (see CGO'08 paper)

Precomputation:

- Compute transitive closure of G', the CFG without DFS back edges;
- For each q, compute a set T_q of back-edge targets reached from q.

Query:

- For each *use*, for each $t \in T_q$ dominated by *def*, test reachability in G'



Fast liveness checking (Boissinot et al. CGO'08)

Definition (Liveness)

Variable a is live-in at q if there is a path from q to a *use* of a, that does not contain its *def*.

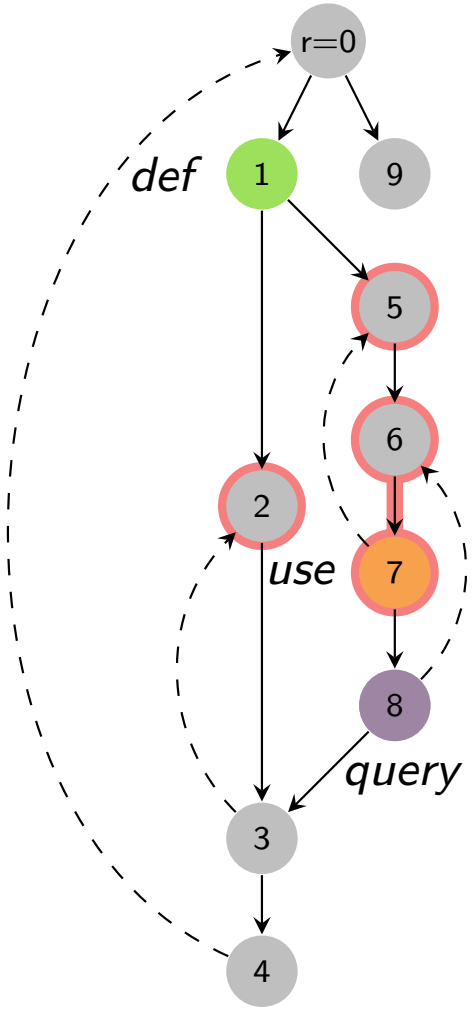
Algorithm (see CGO'08 paper)

Precomputation:

- Compute transitive closure of G', the CFG without DFS back edges;
- For each q, compute a set T_q of back-edge targets reached from q.

Query:

- For each *use*, for each $t \in T_q$ dominated by *def*, test reachability in G'



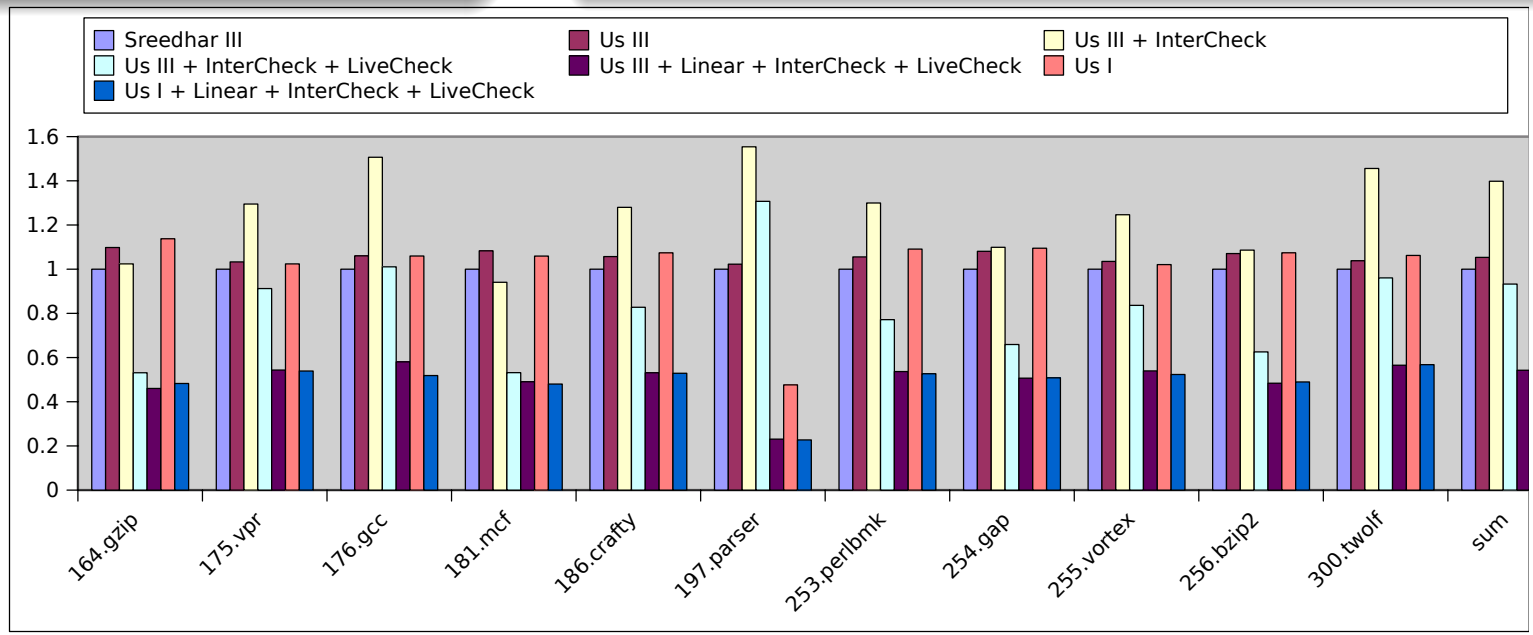
Speed-up for SPEC CINT2000: x2

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.

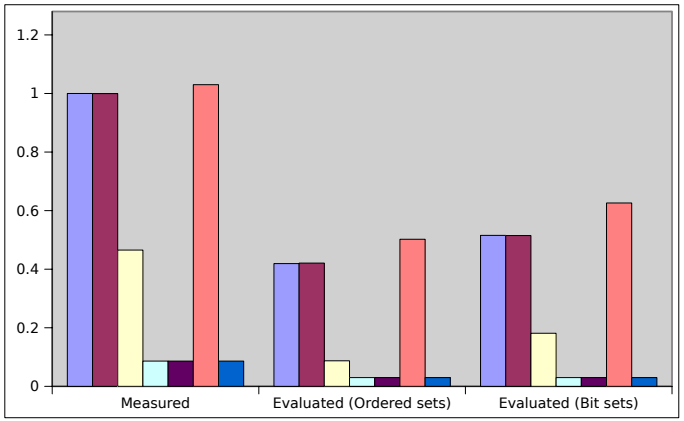


Memory footprint reduction for SPEC CINT2000: x10

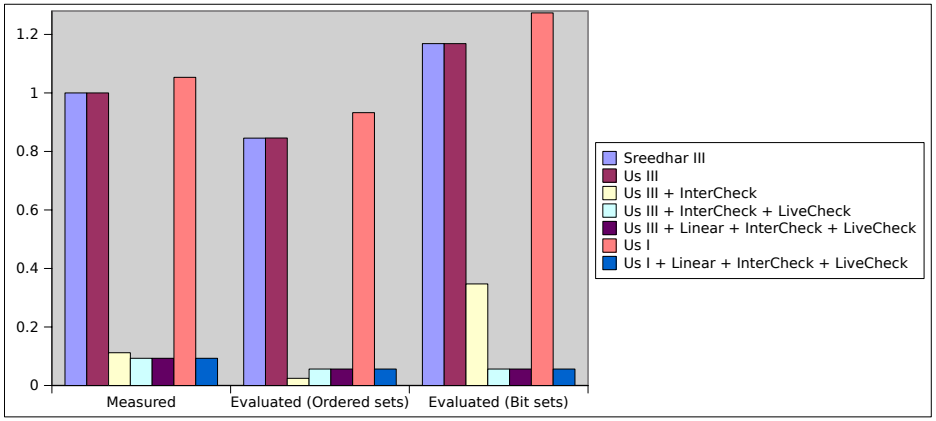
- Interference graph: half-size bit matrix.
- Liveness sets: enumerated sets. Does not count construction.
- Livenesss check: bit sets. Construction taken into account.

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

Sum of memory footprint



Max of memory footprint



Outline

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

The End

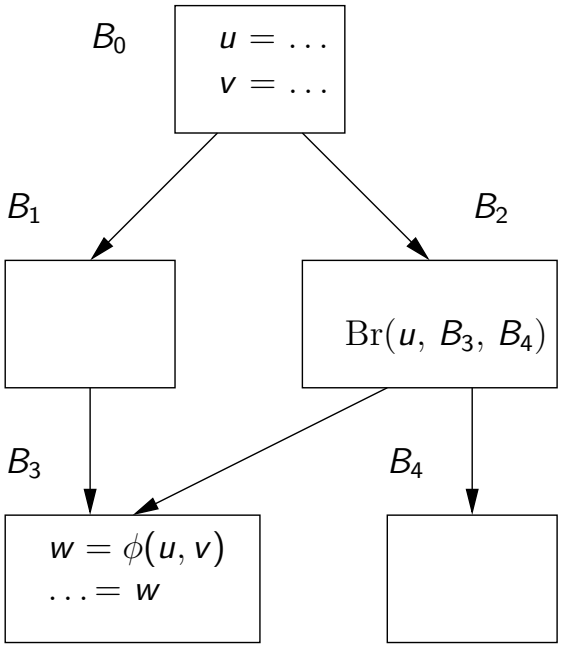
Thank you!

Bug tracking RVM-254 of Jikes RVM

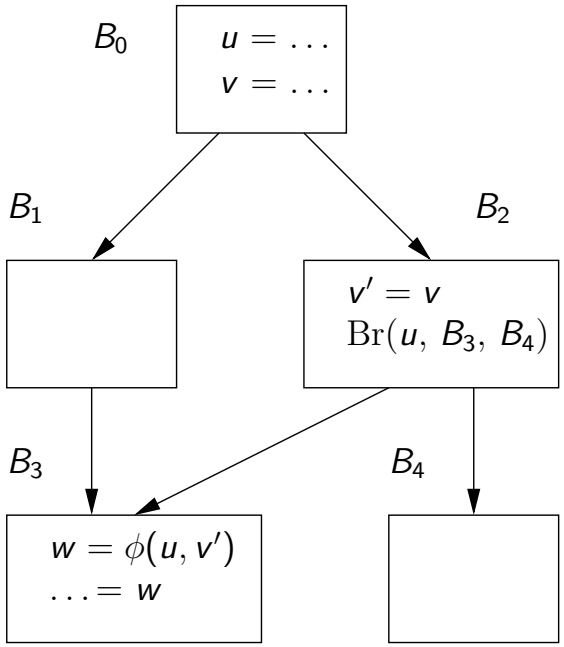
Problems with SSA form: lack of loop unrolling breaks VM

This problem is probably one of the most serious in the RVM currently. When loop unrolling is disabled and SSA enabled the created IR is corrupt. The error has in the past look like we were suffering from the "lost copy" problem, but implementing a naive solution to this didn't solve the problem. Their is sound logic behind the code so we need to identify a small test case where things are broken and then reason about what's wrong in leave SSA. This has been attempted once (with the code that removes an element from the live set) but the problem no longer appears to surface here. Currently these optimizations are disabled but by RVM 3.0 they should be re-enable and this bug cured.

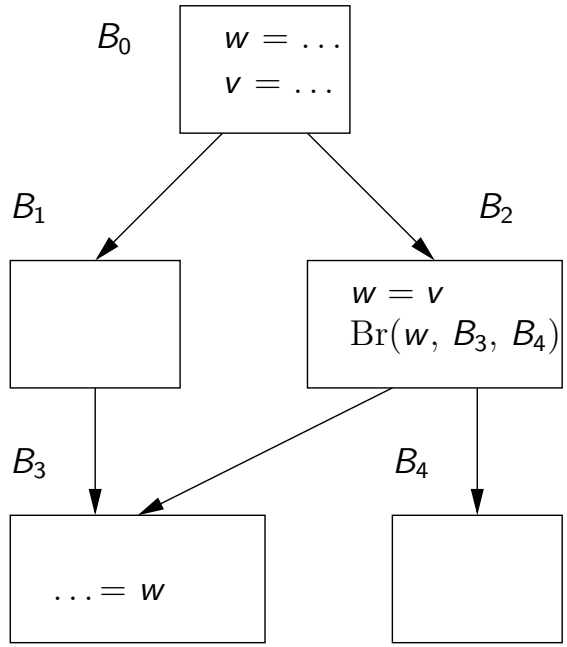
Potential bugs with conditional branches



Initial code

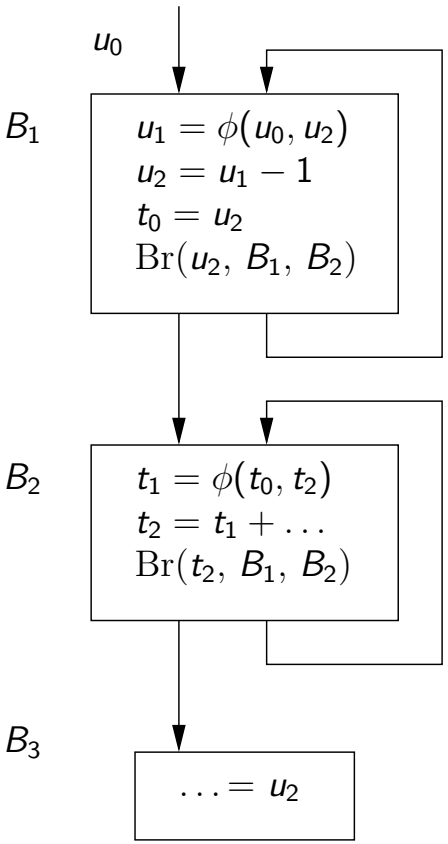


"Blind" Sreedhar III

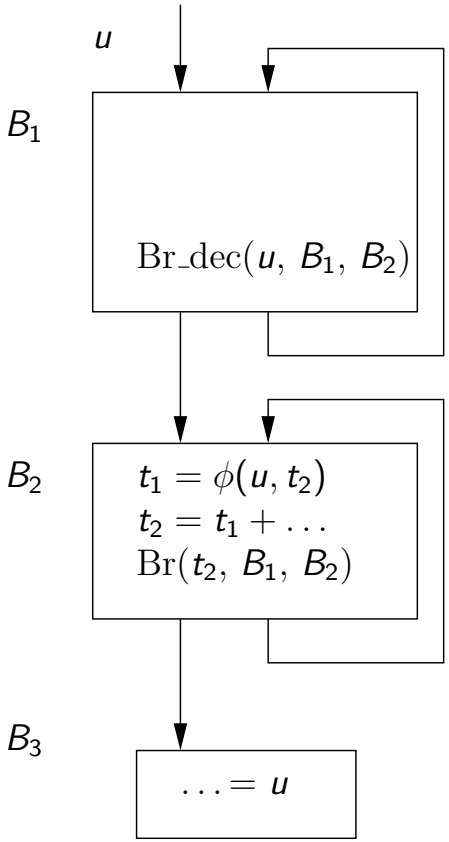


Wrong output code

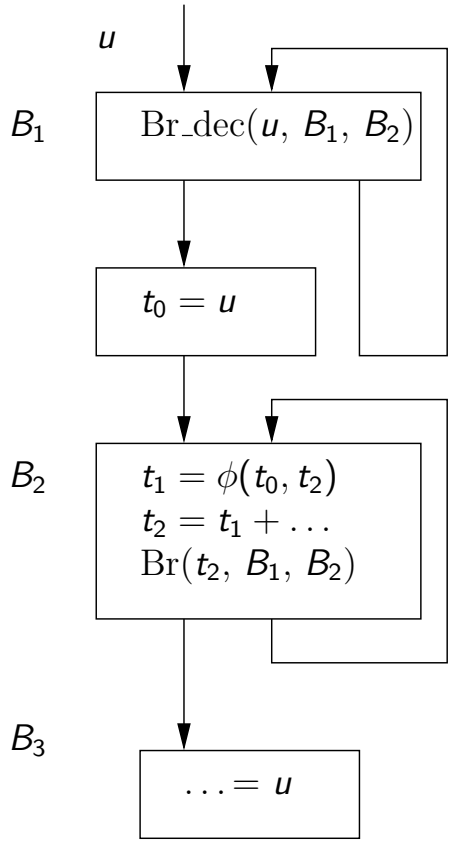
Unfeasible out-of-SSA translation example



Initial code



After optimization



Needs edge splitting