Fast Liveness Checking for SSA-Form Programs

Benoit Boissinot
E.N.S Lyon

Sebastian Hack
E.N.S Lyon

Daniel Grund
Saarland University

Fabrice Rastello
E.N.S. Lyon

Benoît Dupont de Dinechin
STMicroelectronics

April 7, 2008

Outline

Outline

1. Liveness checking: what & why
2. Foundations
3. Algorithm
4. Experimental Results
5. Conclusion

Why do we need liveness analysis?

Resources analysis

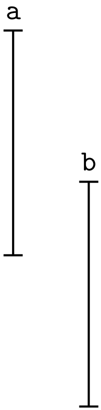
- Scheduling
- Coalescing/Register-allocation
- PRE sensitive to register pressure

```
a =

b =

= a

= b
```



Two approaches

Classical Approach: Liveness Sets (LS)

For *every* block boundary, the set of *all* live variables

- Expensive precomputation (space & time), fast query
- Usually, not all computed information is needed
- Adding, (re-)moving instructions $\Rightarrow$ recompute information

Our Approach: Liveness Checking (LC)

Answer *on demand:* Is variable live at program point?

- Faster precomputation, slower queries
- Information depends only on CFG and def-use chains
- Information invariant to adding, (re-) moving instructions

Outline

Dominance

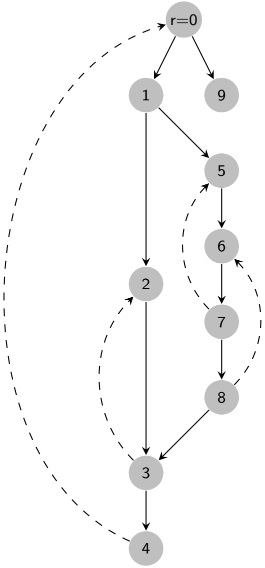
Control Flow Graph

Intermediate representation is a Control Flow Graph (CFG):

- one entry node r
- every node reachable from r

Definition

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



Dominance

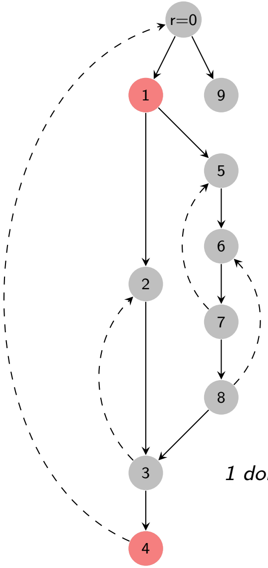
Control Flow Graph

Intermediate representation is a Control Flow Graph (CFG):

- one entry node r
- every node reachable from r

Definition

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



1 dominates 4?

Dominance

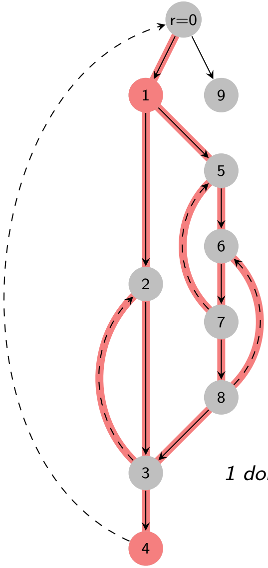
Control Flow Graph

Intermediate representation is a Control Flow Graph (CFG):

- one entry node r
- every node reachable from r

Definition

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



1 dominates 4

Dominance

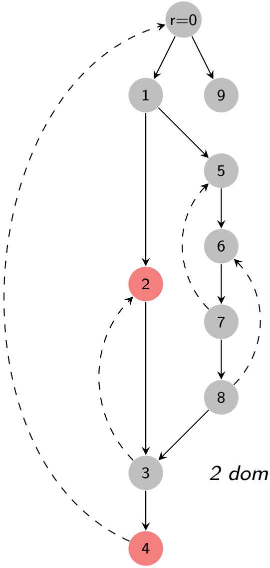
Control Flow Graph

Intermediate representation is a Control Flow Graph (CFG):

- one entry node r
- every node reachable from r

Definition

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



2 dominate 4?

Dominance

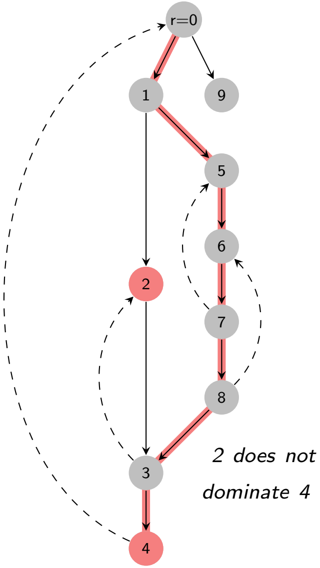
Control Flow Graph

Intermediate representation is a Control Flow Graph (CFG):

- one entry node r
- every node reachable from r

Definition

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

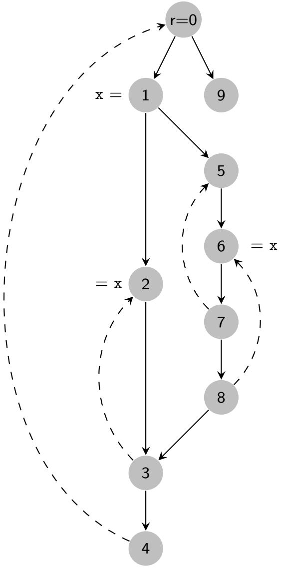


SSA with dominance property

Static Single Assignement (SSA)

Static Single Assignement (SSA) with dominance property:

- (textually) only *one* definition per variable
- the definition d dominates all its uses u_i

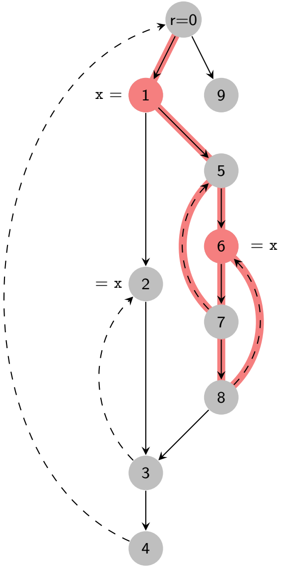


SSA with dominance property

Static Single Assignement (SSA)

Static Single Assignement (SSA) with dominance property:

- (textually) only *one* definition per variable
- the definition d dominates all its uses u_i

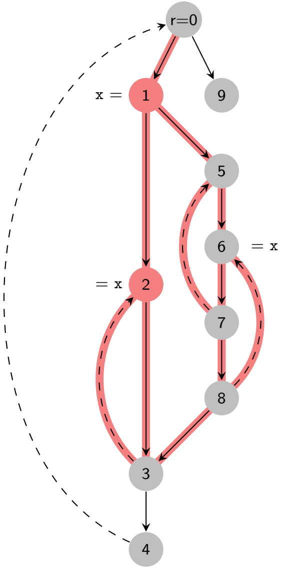


SSA with dominance property

Static Single Assignement (SSA)

Static Single Assignement (SSA) with dominance property:

- (textually) only *one* definition per variable
- the definition d dominates all its uses u_i



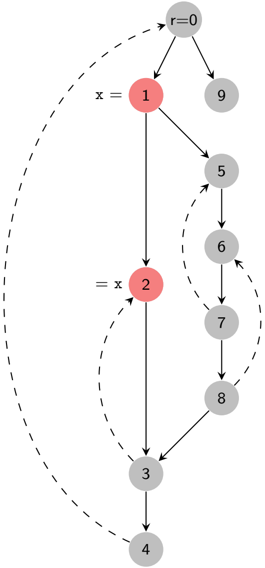
Liveness

Concept

- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)

A variable a is live-in at a node q if there exists a path from q to a node u where a is used and that path does not contain its definition d



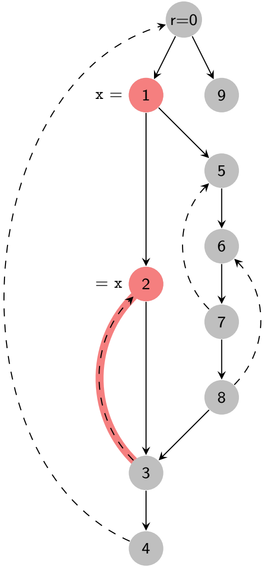
Liveness

Concept

- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)

A variable a is live-in at a node q if there exists a path from q to a node u where a is used and that path does not contain its definition d



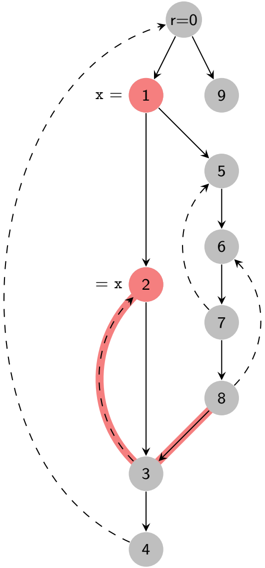
Liveness

Concept

- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)

A variable a is live-in at a node q if there exists a path from q to a node u where a is used and that path does not contain its definition d



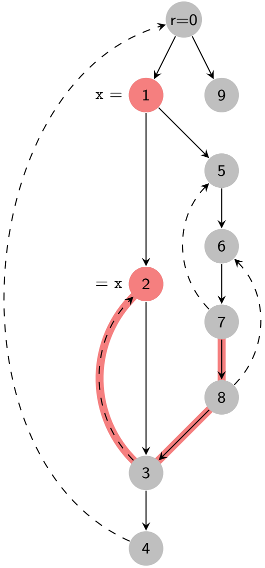
Liveness

Concept

- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)

A variable a is live-in at a node q if there exists a path from q to a node u where a is used and that path does not contain its definition d



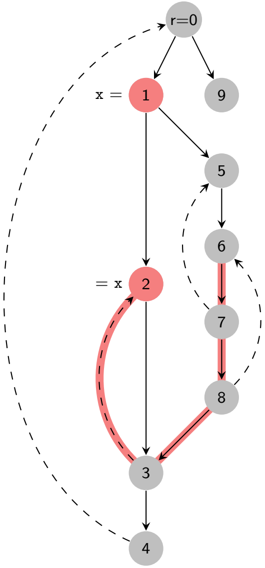
Liveness

Concept

- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)

A variable a is live-in at a node q if there exists a path from q to a node u where a is used and that path does not contain its definition d



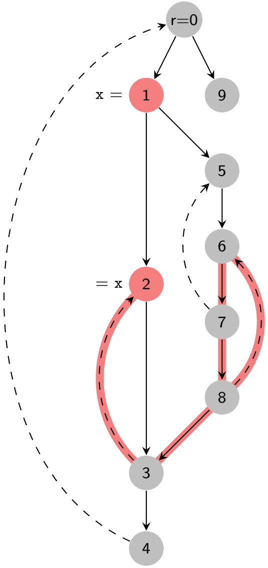
Liveness

Concept

- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)

A variable a is live-in at a node q if there exists a path from q to a node u where a is used and that path does not contain its definition d



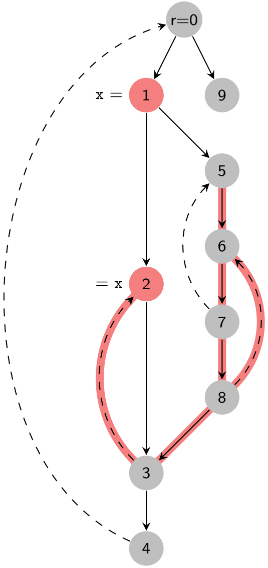
Liveness

Concept

- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)

A variable a is live-in at a node q if there exists a path from q to a node u where a is used and that path does not contain its definition d



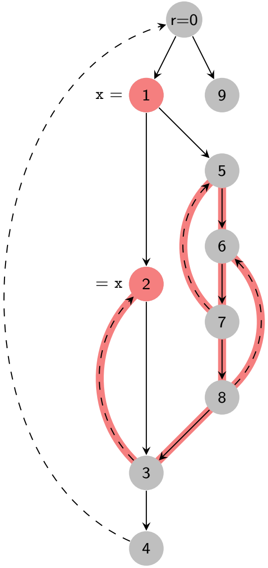
Liveness

Concept

- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)

A variable a is live-in at a node q if there exists a path from q to a node u where a is used and that path does not contain its definition d



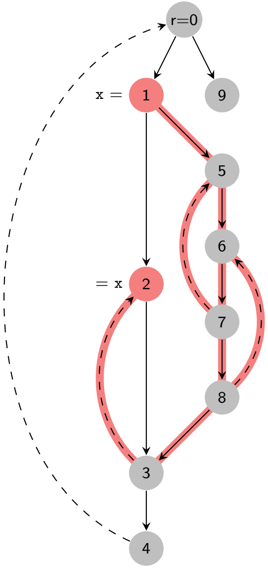
Liveness

Concept

- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)

A variable a is live-in at a node q if there exists a path from q to a node u where a is used and that path does not contain its definition d



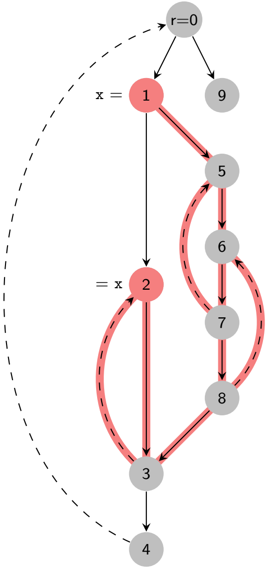
Liveness

Concept

- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)

A variable a is live-in at a node q if there exists a path from q to a node u where a is used and that path does not contain its definition d



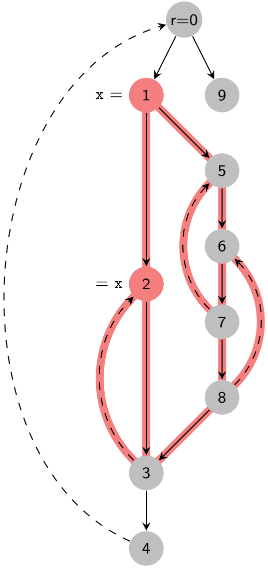
Liveness

Concept

- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)

A variable a is live-in at a node q if there exists a path from q to a node u where a is used and that path does not contain its definition d



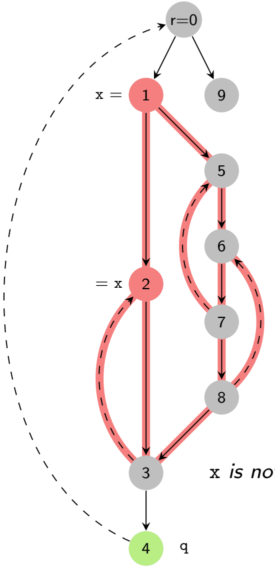
Liveness

Concept

- Defined in the past: reaching definition
- Used in the future: upward exposed use

Definition (live-in)

A variable a is live-in at a node q if there exists a path from q to a node u where a is used and that path does not contain its definition d



x *is not live at* q

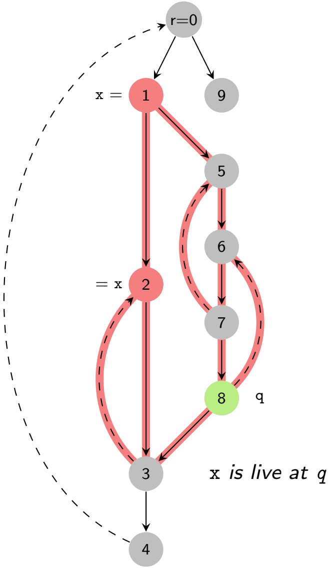
Liveness

Concept

- Defined in the past: reaching definition
- Used in the future: upward exposed use

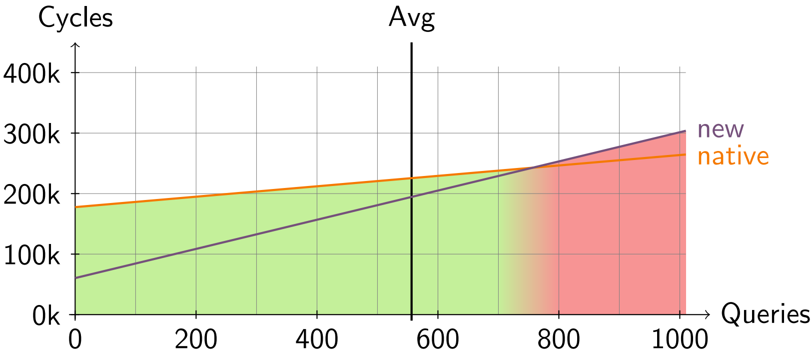
Definition (live-in)

A variable a is live-in at a node q if there exists a path from q to a node u where a is used and that path does not contain its definition d



Liveness: precomputation versus queries

- Classical liveness (data-flow):
  - Costly precomputation
  - Almost constant queries
- Our solution:
  - Fast precomputation
  - Queries almost linear in the number of uses


Cycles
Avg
400k
300k
200k
100k
0k
0
200
400
600
800
1000
Queries
new
native

Outline

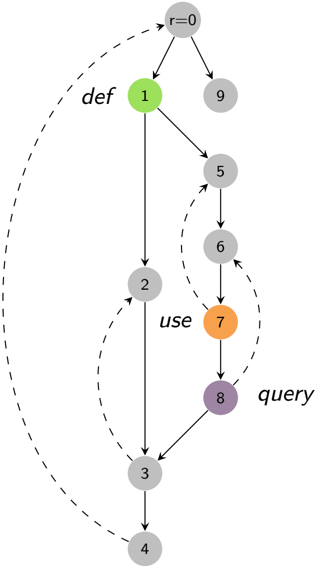
Principle

Goal:
From all the paths from *query* to *use*, remove those going through *def* .

Highest point

Last point of the path such that all the following points are below.

If the highest point is dominated by *def* then the whole path is.



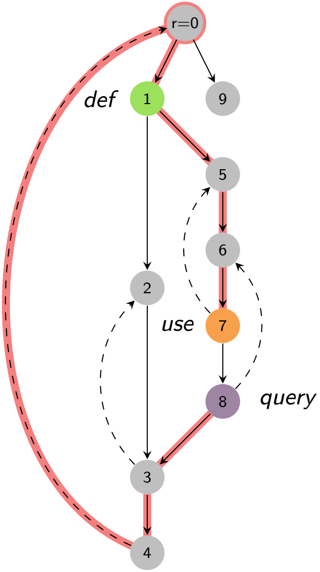
Principle

Goal:
From all the paths from *query* to *use*, remove those going through *def*.

Highest point

Last point of the path such that all the following points are below.

If the highest point is dominated by *def* then the whole path is.



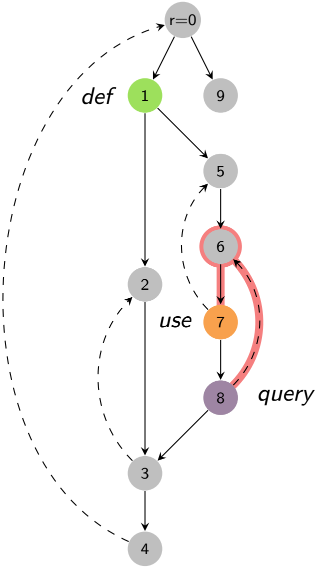
Principle

Goal:
From all the paths from *query* to *use*, remove those going through *def* .

Highest point

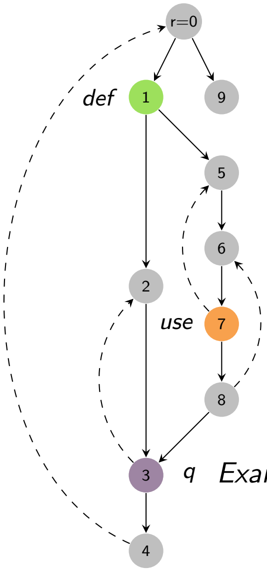
Last point of the path such that all the following points are below.

If the highest point is dominated by *def* then the whole path is.



Principle

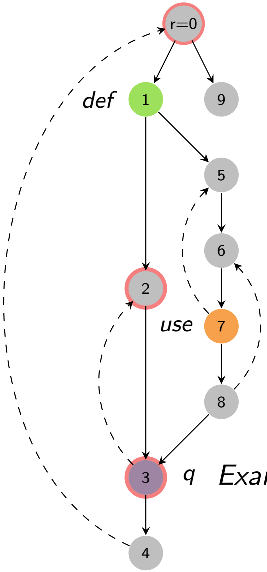
- For each node *q* of the CFG, compute the set of potential *highest points* of every path starting at *q*.
- From this set, remove the points *above def* (not dominated by *def*).
- From the remaining *highest points*, test the *descending* reachability to *use*.



Example 1

Principle

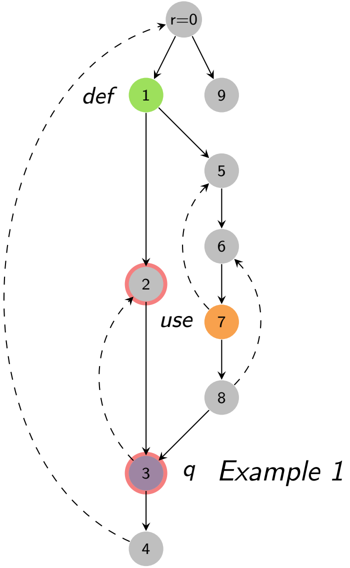
- For each node q of the CFG, compute the set of potential *highest points* of every path starting at q.
- From this set, remove the points *above def* (not dominated by *def*).
- From the remaining *highest points*, test the *descending* reachability to *use*.



Example 1

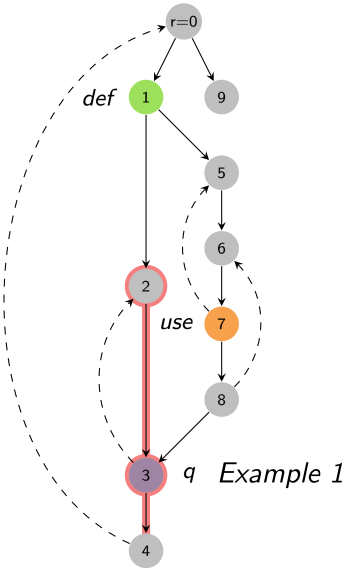
Principle

- For each node q of the CFG, compute the set of potential *highest points* of every path starting at q.
- From this set, remove the points *above def* (not dominated by *def*).
- From the remaining *highest points*, test the *descending* reachability to *use*.



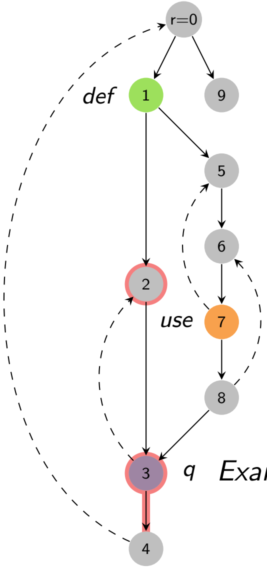
Principle

- For each node q of the CFG, compute the set of potential *highest points* of every path starting at q.
- From this set, remove the points *above def* (not dominated by *def*).
- From the remaining *highest points*, test the *descending* reachability to *use*.



Principle

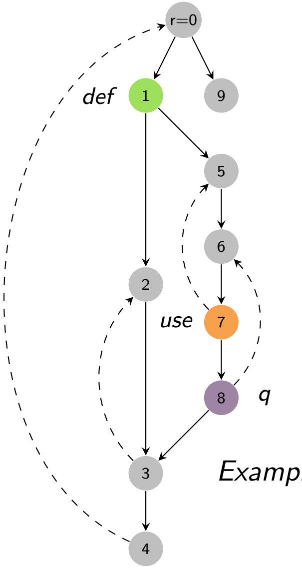
- For each node q of the CFG, compute the set of potential *highest points* of every path starting at q.
- From this set, remove the points *above def* (not dominated by *def*).
- From the remaining *highest points*, test the *descending* reachability to *use*.



Example 1

Principle

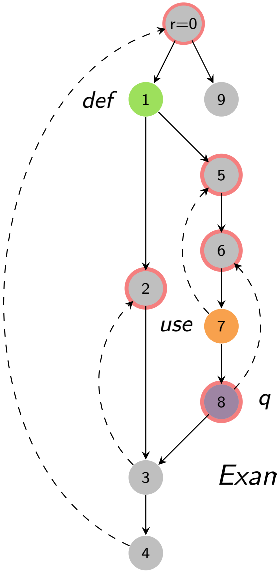
- For each node q of the CFG, compute the set of potential *highest points* of every path starting at q.
- From this set, remove the points *above def* (not dominated by *def*).
- From the remaining *highest points*, test the *descending* reachability to *use*.



Example 2

Principle

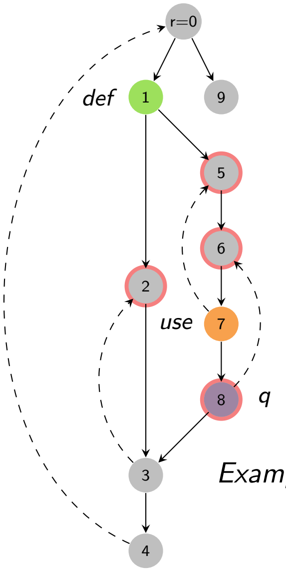
- For each node *q* of the CFG, compute the set of potential *highest points* of every path starting at *q*.
- From this set, remove the points *above def* (not dominated by *def*).
- From the remaining *highest points*, test the *descending* reachability to *use*.



Example 2

Principle

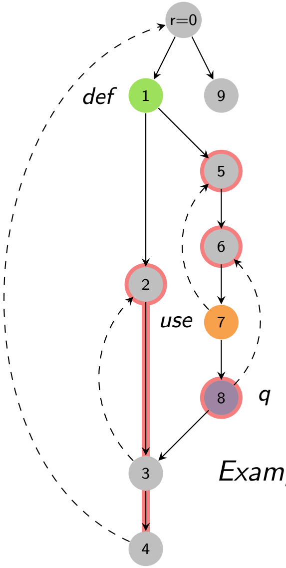
- For each node q of the CFG, compute the set of potential *highest points* of every path starting at q.
- From this set, remove the points *above def* (not dominated by *def*).
- From the remaining *highest points*, test the *descending* reachability to *use*.



Example 2

Principle

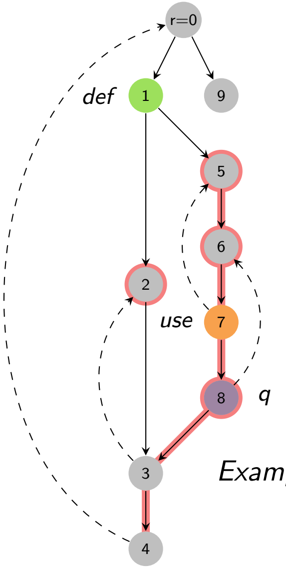
- For each node q of the CFG, compute the set of potential *highest points* of every path starting at q.
- From this set, remove the points *above def* (not dominated by *def*).
- From the remaining *highest points*, test the *descending* reachability to *use*.



Example 2

Principle

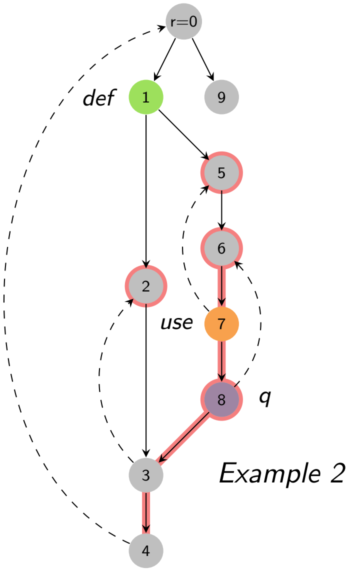
- For each node *q* of the CFG, compute the set of potential *highest points* of every path starting at *q*.
- From this set, remove the points *above def* (not dominated by *def*).
- From the remaining *highest points*, test the *descending* reachability to *use*.



Example 2

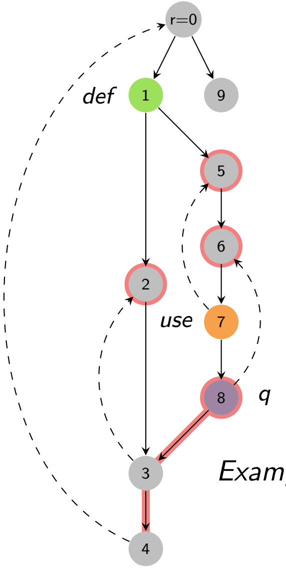
Principle

- For each node q of the CFG, compute the set of potential *highest points* of every path starting at q.
- From this set, remove the points *above def* (not dominated by *def*).
- From the remaining *highest points*, test the *descending* reachability to *use*.



Principle

- For each node q of the CFG, compute the set of potential *highest points* of every path starting at q.
- From this set, remove the points *above def* (not dominated by *def*).
- From the remaining *highest points*, test the *descending* reachability to *use*.



Example 2

Algorithm

Precomputation

1. Compute transitive closure on the reduced graph G'
   - G' = CFG without DFS back edges (cycle-free)
   - Simple to compute: post-order traversal
2. For each node q compute a set T_q of possible highest points (back-edge targets)
   - Also simple to compute: pre-order and post-order traversal

Query

- For each *use*:
  - For each $t \in T_q$ dominated by *def*:
    - Test reachability in the reduced graph

Implementation Tricks

- Reachability and T_q can be efficiently implemented as bitsets
- For reducible CFGs there is exactly one “highest” back-edge target
  - dominates all the other back-edge targets
  - sufficient to check from there
- Hence, order nodes according to dominance

  ☞ “highest” node is first set bit in T_q

Outline

Evaluation

Setup

- Implemented in LAO, code generator developed by STMicroelectronics
- Benchmarked with a subset of SPEC2000 (CINT)
- Liveness-analysis used during SSA deconstruction

The main factors influencing the speed of our algorithm are:

- the number of uses per variable ($\#uses$)
- the number of basic blocks ($\#BB$)
- the number of CFG edges ($\#edges$)

Quantitative Evaluation

	# of Uses per Variable			
Benchmark	Maximum	$\% \leq 1$	$\% \leq 2$	$\% \leq 3$
164.gzip	51	65.64	86.38	92.81
175.vpr	75	70.36	88.90	93.93
176.gcc	422	73.99	87.81	92.42
181.mcf	46	66.91	83.50	89.33
186.crafty	620	72.98	90.09	93.85
197.parser	96	65.12	86.75	94.26
254.gap	156	70.46	85.95	91.26
255.vortex	254	65.99	90.80	95.02
256.bzip2	36	69.89	89.89	94.47
300.twolf	165	69.71	87.59	93.23
Total	620	71.30	87.85	92.76

Quantitative Evaluation

	# of Basic Blocks		
Benchmark	Average	% $\leq$ 32	% $\leq$ 64
164.gzip	33.35	69.51	85.36
175.vpr	34.45	68.88	84.44
176.gcc	38.96	72.85	86.03
181.mcf	20.31	84.61	100.00
186.crafty	69.28	59.63	76.14
197.parser	23.60	84.82	93.49
254.gap	32.89	67.60	87.44
255.vortex	26.46	77.57	90.68
256.bzip2	22.97	78.37	91.89
300.twolf	56.97	59.47	77.36
Total	35.21	72.71	87.18

Runtime Experiments

	Speedup		
Benchmark	Precomputation	Queries	Both
164.gzip	3.12	0.53	1.16
175.vpr	2.17	0.48	1.41
176.gcc	3.03	0.26	1.00
181.mcf	1.85	0.44	1.39
186.crafty	2.78	0.49	0.73
197.parser	2.13	0.49	1.54
254.gap	3.45	0.52	2.08
255.vortex	1.67	0.45	1.32
256.bzip2	3.45	0.51	2.32
300.twolf	4.76	0.49	1.92
Total	2.94	0.36	1.16

Outline

Contributions

- Novel approach for liveness checking relying only on the CFG
- Fast construction algorithm
- Overall speedup in most cases

Future Work

- Dynamic update for CFG transformations
- Memory efficient reachability
- Use information available from the loop nesting forest

The End

Thank you!