Towards an SSA based compiler back-end: some interesting properties of SSA and its extensions

Benoit Boissinot

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

Membres du jury :	Albert Cohen (rapporteur)	INRIA Saclay
	Anton Ertl (examinateur)	TU Wien
	David Monniaux (rapporteur)	Verimag
	Fabrice Rastello (directeur de thèse)	ENS de Lyon
	Yves Robert (examinateur)	ENS de Lyon

Compiling, compilers

```c
int max() {
        int x = 1, y = 2, r;
        if (x > y)
                r = x;
        else
                r = y;
        return r;
}
```

```
        .file       "foo.c"
        .text
.globl max
        .type       max, @function
max:
        pushl       %ebp
        movl        %esp, %ebp
        subl        $16, %esp
        movl        $1, -4(%ebp)
        movl        $2, -8(%ebp)
        movl        -4(%ebp), %eax
        cmpl        -8(%ebp), %eax
        jle         .L2
        movl        -4(%ebp), %eax
        movl        %eax, -12(%ebp)
        jmp         .L3
.L2:
        movl        -8(%ebp), %eax
        movl        %eax, -12(%ebp)
.L3:
        movl        -12(%ebp), %eax
        leave
        ret
        .size       max, .-max
        .ident      "GCC: (Ubuntu 4.4.3-4ubuntu5) 4.4.3"
        .section    .note.GNU-stack,"",@progbits
```

Compiling, compilers

Two compilation phases

1. On a workstation: compile to architecture independant bytecode
2. On the target processor, compile bytecode to native code

Compiling, compilers

Two compilation phases

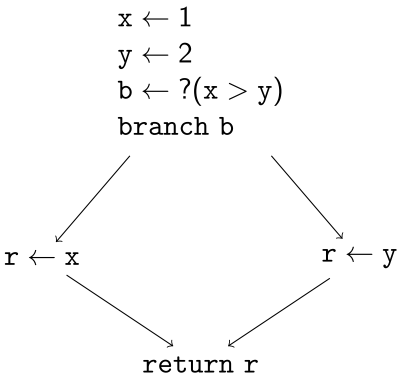
1. On a workstation: compile to architecture independant bytecode
2. On the target processor, compile bytecode to native code

Challenges

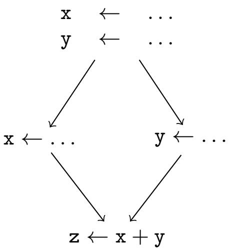
Reduce compilation time (speed of compilation),
while keeping a high code quality (speed of execution).

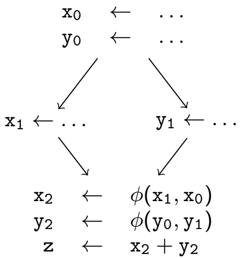
Control flow graph (CFG)

- Program is represented as a graph
- Node: sequence of instruction without branch
- Edge: possible execution flow (branch instructions)



Static Single Assignment (SSA)





For each variable:

- Only one textual definition ($\Rightarrow$ renaming)
- No use of undefined variable (dominance property)

Add ϕ-function at merge points, acts as parallel switch

Use of SSA

Goal

One static definition per variable ⇒ attach properties to variable (sparse representation)

Properties

- Dominance property
- Unique definition

⇒ simpler and more efficient algorithms

Use of SSA

Goal

One static definition per variable $\Rightarrow$ attach properties to variable (sparse representation)

Properties

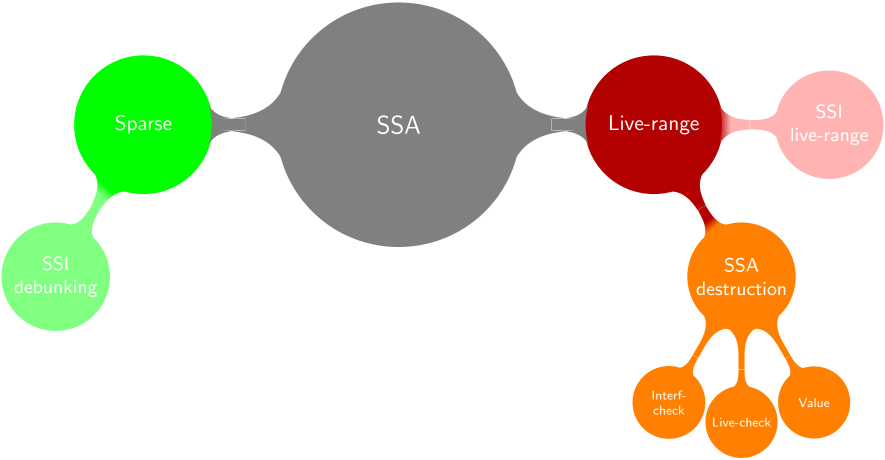
- Dominance property
- Unique definition

$\Rightarrow$ simpler and more efficient algorithms

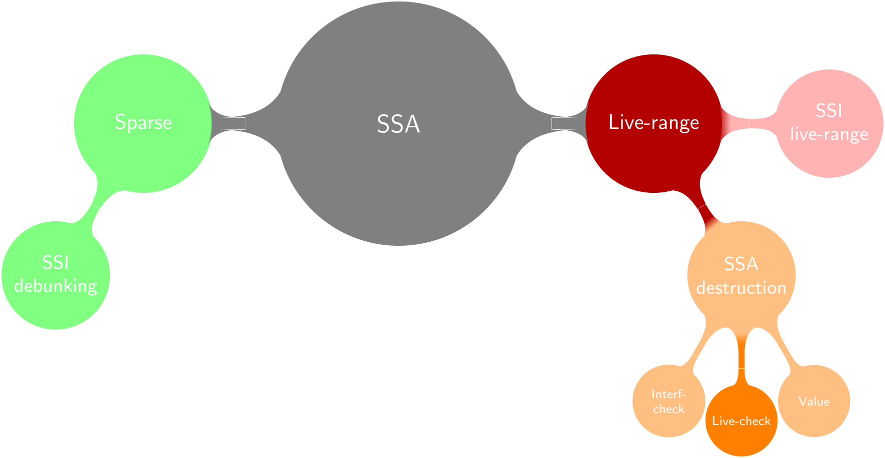
Drawbacks

- Create more variables (memory)
- Cost of maintainance
- Need to get rid of ϕ-functions

Outline


Sparse
SSA
Live-range
SSI
live-range
SSI
debunking
SSA
destruction
Interf-
check
Live-check
Value

Outline


Sparse
SSA
Live-range
SSI
live-range
SSI
debunking
SSA
destruction
Interf-
check
Live-check
Value

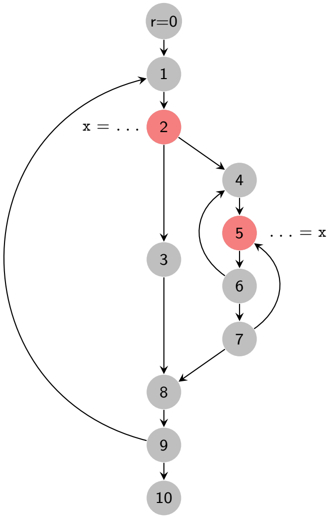
Liveness

Definition (live-in)

Existence of a path to a use that does not contain the definition.

Definition (live-range)

Set of program points where a variable is live-in.



What to compute?

Classical Approach: Liveness Sets

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

What to compute?

Classical Approach: Liveness Sets

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 ⇒ recompute information

Alternative approach: Liveness Checking

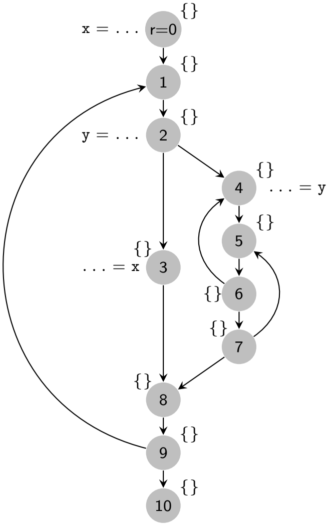
Answer on demand: Is a variable live at program point?

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

Data-flow

Traditional approach (not SSA specific)

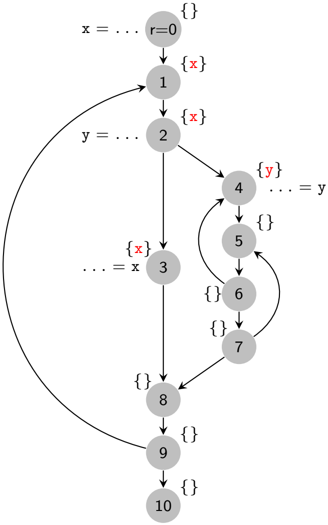
Backward data-flow propagation: propagates the information about every variable inside the CFG.
Wait for a fixed-point.
⇒ number of iterations depends on the cyclic structure.



Data-flow

Traditional approach (not SSA specific)

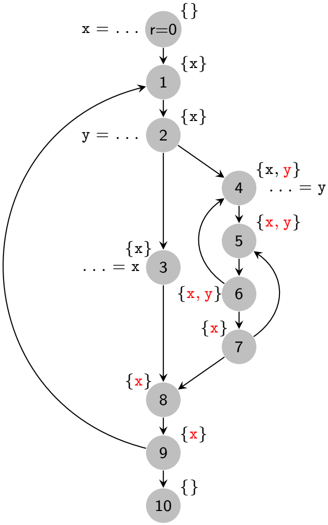
Backward data-flow propagation: propagates the information about every variable inside the CFG.
Wait for a fixed-point.
⇒ number of iterations depends on the cyclic structure.



Data-flow

Traditional approach (not SSA specific)

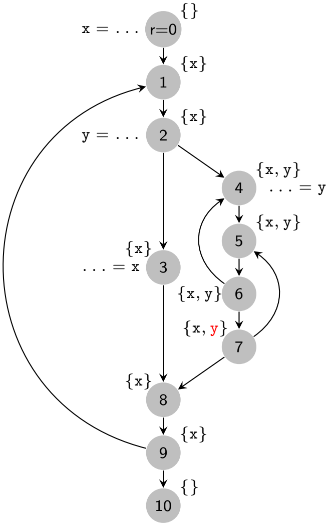
Backward data-flow propagation: propagates the information about every variable inside the CFG.
Wait for a fixed-point.
⇒ number of iterations depends on the cyclic structure.



Data-flow

Traditional approach (not SSA specific)

Backward data-flow propagation: propagates the information about every variable inside the CFG.
Wait for a fixed-point.
$\Rightarrow$ number of iterations depends on the cyclic structure.



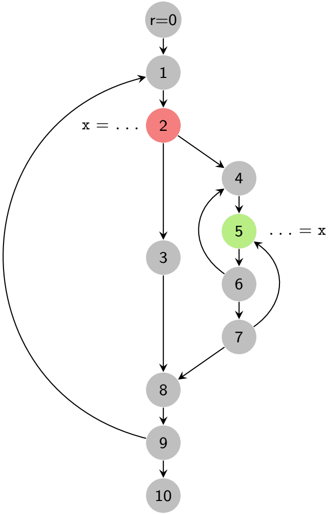
Path exploration (principle)

Context

- Found in some modern compiler textbooks, requires or builds use and def sets.
- Discover live-range: starting from the uses, mark the ancestors in the CFG, stop when a definition is found.

Efficiency

- Few uses per variables.
- Under SSA: short live-ranges, use-def chains and def-use chains available.



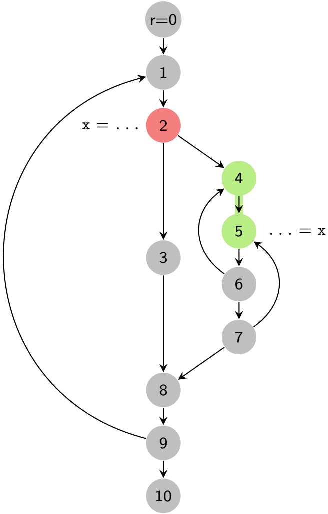
Path exploration (principle)

Context

- Found in some modern compiler textbooks, requires or builds use and def sets.
- Discover live-range: starting from the uses, mark the ancestors in the CFG, stop when a definition is found.

Efficiency

- Few uses per variables.
- Under SSA: short live-ranges, use-def chains and def-use chains available.



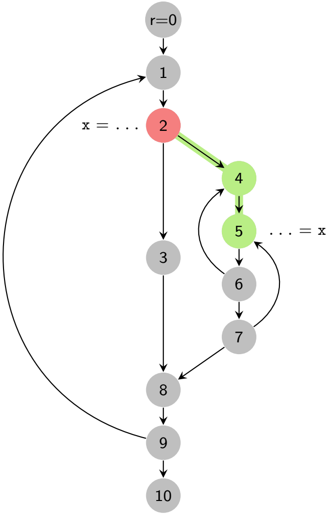
Path exploration (principle)

Context

- Found in some modern compiler textbooks, requires or builds use and def sets.
- Discover live-range: starting from the uses, mark the ancestors in the CFG, stop when a definition is found.

Efficiency

- Few uses per variables.
- Under SSA: short live-ranges, use-def chains and def-use chains available.



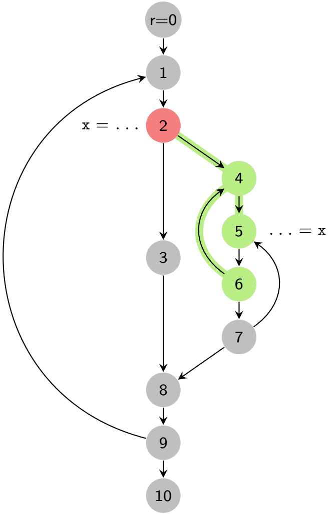
Path exploration (principle)

Context

- Found in some modern compiler textbooks, requires or builds use and def sets.
- Discover live-range: starting from the uses, mark the ancestors in the CFG, stop when a definition is found.

Efficiency

- Few uses per variables.
- Under SSA: short live-ranges, use-def chains and def-use chains available.



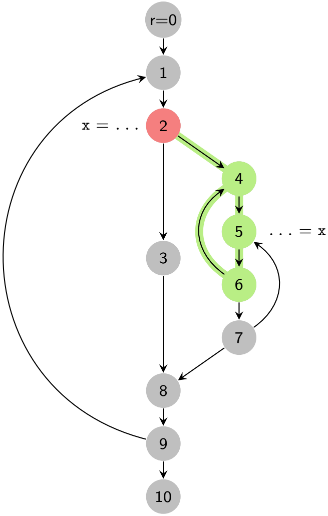
Path exploration (principle)

Context

- Found in some modern compiler textbooks, requires or builds use and def sets.
- Discover live-range: starting from the uses, mark the ancestors in the CFG, stop when a definition is found.

Efficiency

- Few uses per variables.
- Under SSA: short live-ranges, use-def chains and def-use chains available.



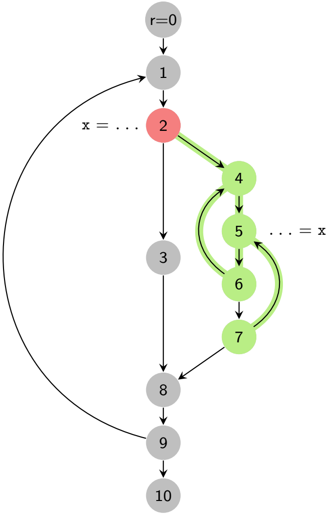
Path exploration (principle)

Context

- Found in some modern compiler textbooks, requires or builds use and def sets.
- Discover live-range: starting from the uses, mark the ancestors in the CFG, stop when a definition is found.

Efficiency

- Few uses per variables.
- Under SSA: short live-ranges, use-def chains and def-use chains available.



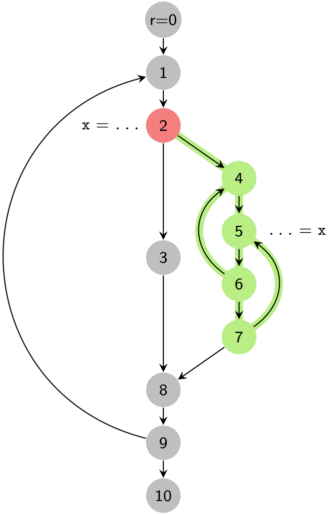
Path exploration (principle)

Context

- Found in some modern compiler textbooks, requires or builds use and def sets.
- Discover live-range: starting from the uses, mark the ancestors in the CFG, stop when a definition is found.

Efficiency

- Few uses per variables.
- Under SSA: short live-ranges, use-def chains and def-use chains available.



Path exploration (algorithm)

Live-sets

For each variable:

1. build the live-range
2. update the live-sets

Complexity: size of resulting sets (cost of insertions)

Live-check

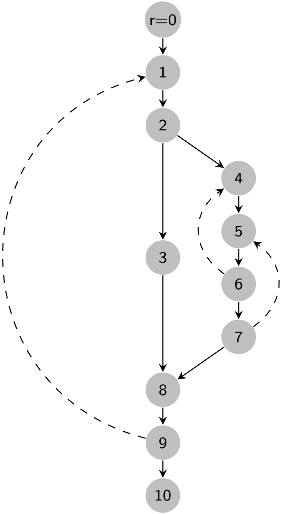
Given a variable `x` and a program point p, build the live-range of `x`, test membership of p.

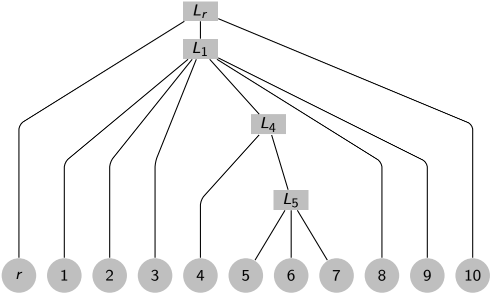
Loops

Loop

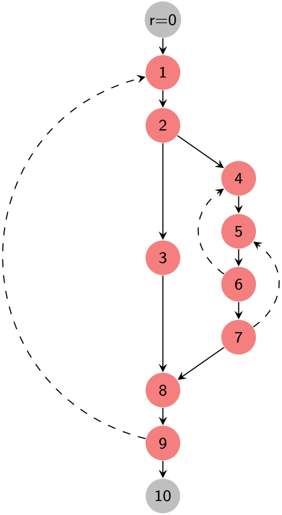
- Decomposition of the CFG in strongly connected components
- Headers: distinguished nodes (usually entry points)
- Form a tree structure

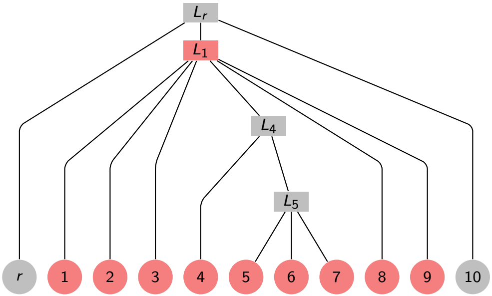
Loops (example)



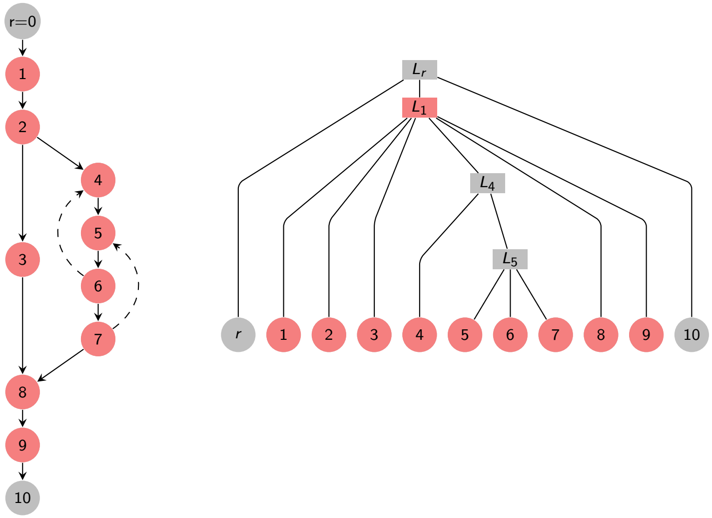


Loops (example)

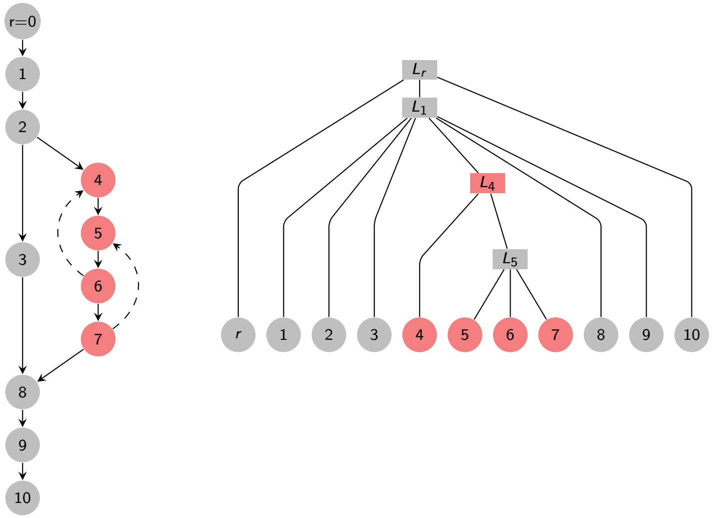




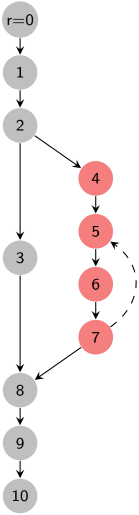
Loops (example)

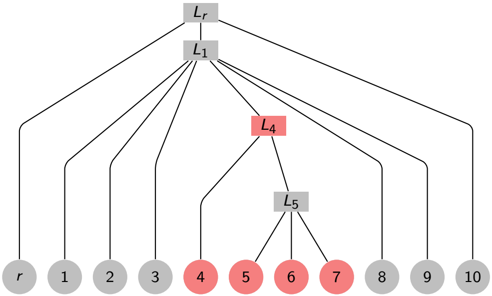


Loops (example)

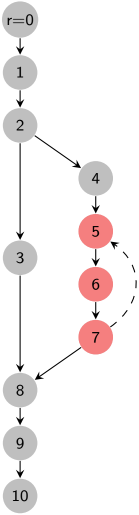


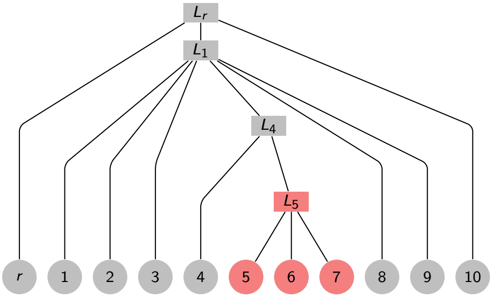
Loops (example)



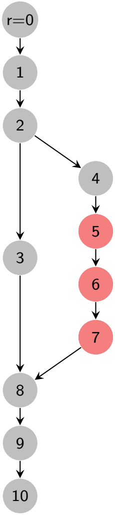


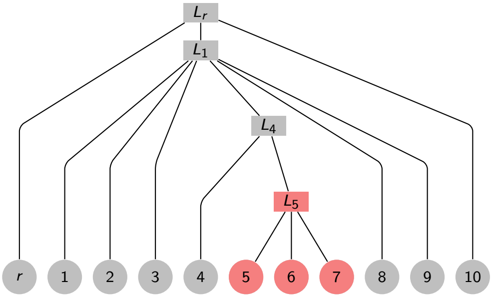
Loops (example)





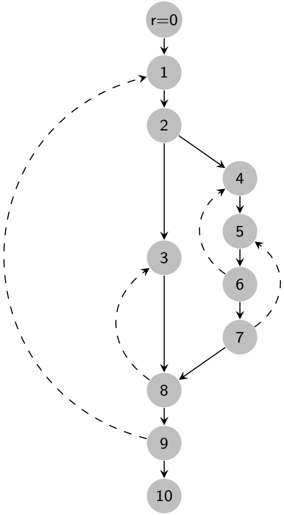
Loops (example)





Irreducible CFG

- Strongly connected components with several entry points
- Unstructured original code (goto)
- More complex algorithms



Loop-based liveness (principle)

Build partial liveness

Liveness is easy to compute on directed acyclic graphs (DAG)
$\Rightarrow$ partial liveness on the CFG without the loop-edges (reduced DAG)

Loop-based liveness (principle)

Build partial liveness

Liveness is easy to compute on directed acyclic graphs (DAG)
$\Rightarrow$ partial liveness on the CFG without the loop-edges (reduced DAG)

Fix liveness using loops

For every SSA variable `x` and program point p such that `x` is live-in at p:

- either p is found live-in with partial liveness,
- or there exists a header of a loop containing p: h, such that h is found live-in with partial liveness.

Reciprocally, if `x` is live-in of a loop header h, it is also live-in of every node contained in the loop.

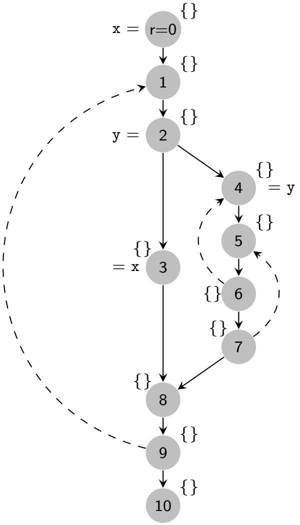
Two-pass data-flow (live-sets)

First pass (build partial liveness)

One pass of data-flow in reverse topological order of the reduced acyclic graph.

Second pass (fix liveness)

Add the missing variables to the live-sets with a top-down pass in the loop forest.



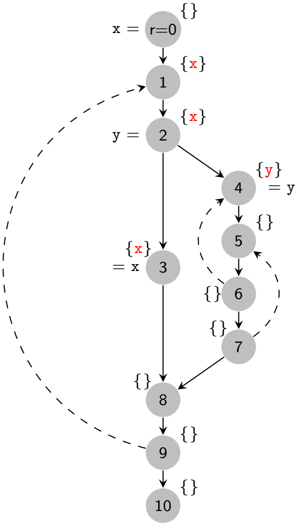
Two-pass data-flow (live-sets)

First pass (build partial liveness)

One pass of data-flow in reverse topological order of the reduced acyclic graph.

Second pass (fix liveness)

Add the missing variables to the live-sets with a top-down pass in the loop forest.



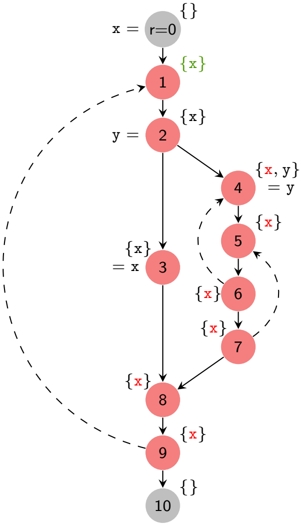
Two-pass data-flow (live-sets)

First pass (build partial liveness)

One pass of data-flow in reverse topological order of the reduced acyclic graph.

Second pass (fix liveness)

Add the missing variables to the live-sets with a top-down pass in the loop forest.



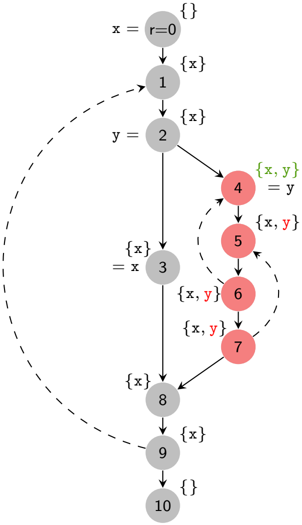
Two-pass data-flow (live-sets)

First pass (build partial liveness)

One pass of data-flow in reverse topological order of the reduced acyclic graph.

Second pass (fix liveness)

Add the missing variables to the live-sets with a top-down pass in the loop forest.



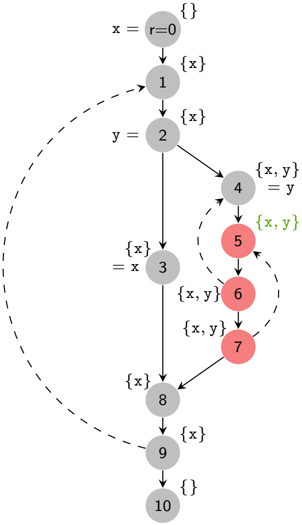
Two-pass data-flow (live-sets)

First pass (build partial liveness)

One pass of data-flow in reverse topological order of the reduced acyclic graph.

Second pass (fix liveness)

Add the missing variables to the live-sets with a top-down pass in the loop forest.



Loop-based liveness (live-check)

Query

Given a variable x, a program point p and a u a use of x:

1. Find h: header of largest loop containing p but not the definition of x.
2. Check existence of path from h to u in the reduced DAG.

Step 1 can be reduced to the least-common ancestor problem in the loop-nesting tree.

Loop-based liveness (irreducible graphs)

Graph transformation preserving liveness

Given a graph that is not reducible, we can transform it, while preserving liveness and loop structure, so that it becomes reducible:

1. For every loop, choose an entry node as unique header
2. For every edge entering the loop, redirect them to the chosen header

No need to actually modify the graph.

Summary

Algorithms	Live-sets	Live-check
Data-flow	✓	–
Path exploration	✓	✓
Loop	✓	✓

Summary

Algorithms	Live-sets	Live-check
Data-flow	✓	–
Path exploration	✓	✓
Loop	✓	✓

Requirements

Path exploration:

- Sets of use and definitions

Loop-based liveness:

- Loop-nesting tree
- SSA
- Def-use chains for live-check

Summary

Loop-based liveness

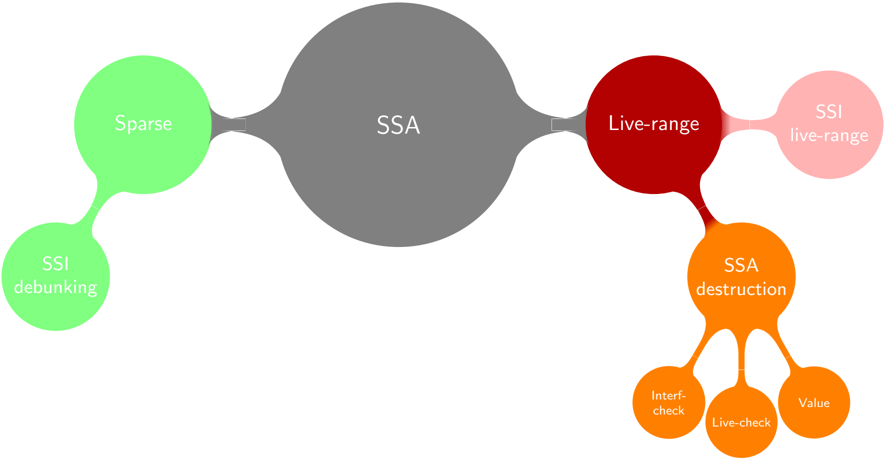
Use SSA properties to derive a new way to find liveness

Live-check

New way to use liveness information:

- easy to pre-compute
- no maintainance needed
- overall speedup depending on the number of queries

Outline


Sparse
SSA
Live-range
SSI live-range
SSI debunking
SSA destruction
Interf-check
Live-check
Value

Previous approaches

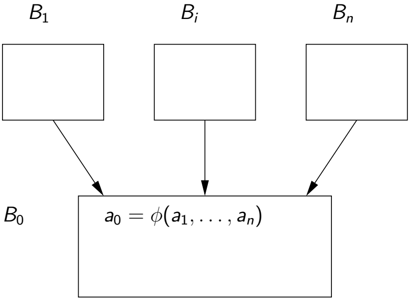
- Cytron et al. (1991): copies in predecessor basic blocks. Incorrect because of a bad understanding of:
  - parallel nature of ϕ-functions;
  - critical edges.
- Briggs et al. (1998): both problems identified. General correctness unclear.
- Sreedhar et al. (1999): correct but
  - handling of complex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

Going to CSSA (conventional SSA)

Conventional SSA (CSSA)

For any ϕ-function $a_0 = \phi(a_1, \ldots, a_n)$, the variables $a_0, \ldots, a_n$ can be safely replaced by a common ressource.

From SSA to CSSA



Going to CSSA (conventional SSA)

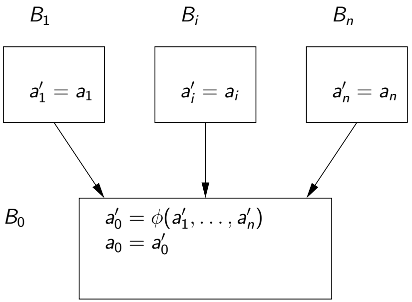
Conventional SSA (CSSA)

For any ϕ-function $a_0 = \phi(a_1, \ldots, a_n)$, the variables $a_0, \ldots, a_n$ can be safely replaced by a common ressource.

Correctness

Add copies with new local variables around every ϕ.
$\Longrightarrow$ CSSA

From SSA to CSSA



Code quality

Removing copies

Useless copies can be removed by standard aggressive coalescing.
$\Rightarrow$ use an accurate notion of interference

Traditional interference

Definition (ultimate interference)

Two variables interfere if they can be simultaneously live while having different values.

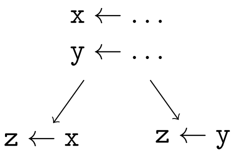
Traditional interference

Definition (ultimate interference)

Two variables interfere if they can be simultaneously live while having different values.

Chaintin's approximation

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



$$\begin{array}{ccc} \mathtt{x} & \leftarrow & \mathtt{z} \\ \mathtt{y} & \leftarrow & \mathtt{z} \\ \ldots & \leftarrow & \mathtt{x} \end{array}$$

In both cases, x interferes with y.

Exploiting SSA: value-based interference

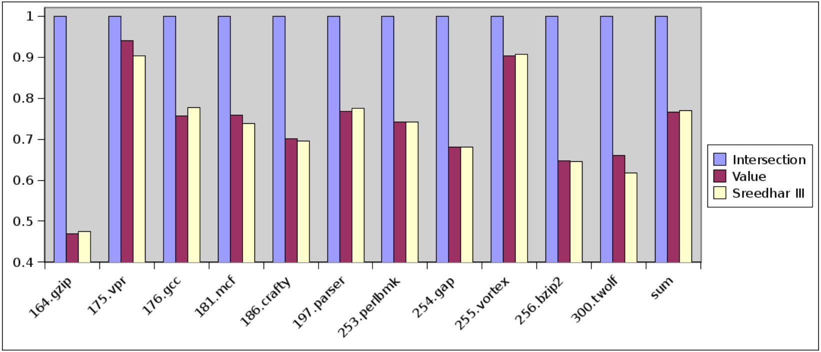
Unique value V of a SSA variable

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

Value-based interference

Two variables $\mathtt{x}$ and $\mathtt{y}$ interfere if $V(\mathtt{x}) \neq V(\mathtt{y})$ and one is live at the definition of the other.

Qualitative experiments with SPEC CINT2000



Number of remaining moves

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.

Fast interference test for a set of variables

Key properties for linear-complexity live range intersection

- 2 SSA variables intersect if one is live at the definition of the other.
- In this case, the first definition dominates the second one.
- Budimlić: If a and b intersect (a dom b), then
  $\forall c$ with a dom c and c dom b: b and c interferes.

$\Longrightarrow$ For each variable, the only test needed is with the "closest" dominating variable

Linear interference test of two congruence classes

Generalization to interference test of two sets

- DFS traversal of a tree
  $\Rightarrow$ Emulate traversal
- Interference inside a set
  $\Rightarrow$ Interference between two sets
- Take values into account

$\Rightarrow$ Test and merge linearly two sets of variables

Linear interference test of two congruence classes

Generalization to interference test of two sets

- DFS traversal of a tree
  ⇒ Emulate traversal
- Interference inside a set
  ⇒ Interference between two sets
- Take values into account

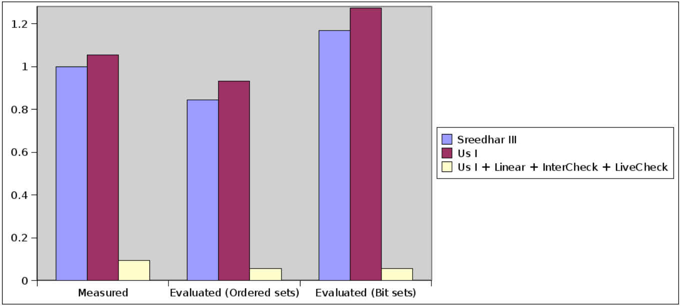
⇒ Test and merge linearly two sets of variables

Fewer intersection tests ⇒ more expensive queries for intersection, avoid interference graph:

- Budimlić intersection test, using liveness sets.
- Liveness checking (presented earlier)

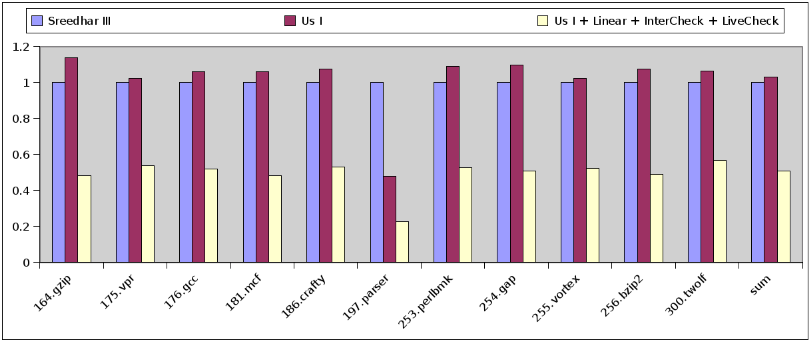
Memory footprint reduction for SPEC CINT2000: x10

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



Max of memory footprint

Speed-up for SPEC CINT2000: x2



Time to go out of SSA (valgrind cycles)
Default: Liveness sets + interference graph

Summary

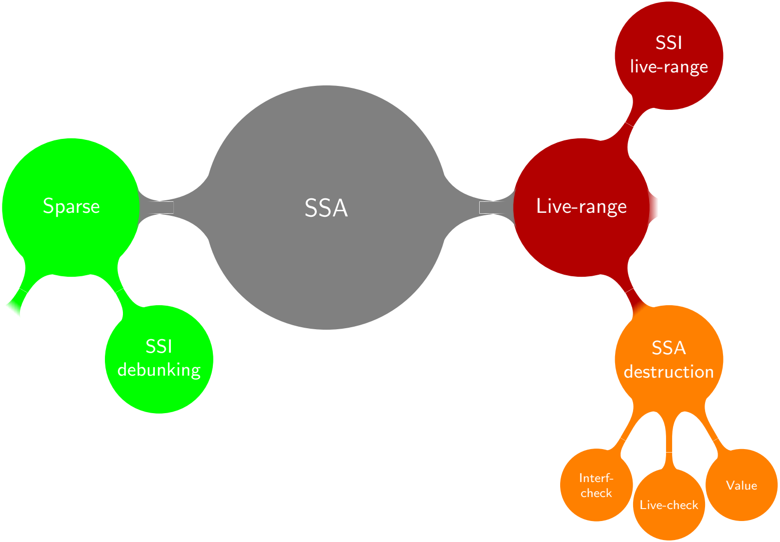
General framework

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

Results

- Value-based interference as good as Sreedhar III
- Fast algorithm: Speed-up x2, memory reduction x10.
- Simple implementation

Conclusion


SSA
Sparse
SSI
debunking
Live-range
SSI
live-range
SSA
destruction
Interf-
check
Live-check
Value

Perspectives